netX Dual-Port Memory Interface Manual

netX Dual-Port Memory Interface for netX based Products

Language: English

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 (0 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 4.7.2.3, 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 7 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 5.4.3 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 Perfomance 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

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

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

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

Table of Content

List of Figures

List of Tables

Page left blank

1 Introduction

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.

1.1 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
Hz	Hertz (1 per Second)
I²C	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)
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 1 - 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.

Windows® 2000/Windows® XP are registered trademarks of Microsoft Corporation.

1.2 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.

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 structure and function of the dual-port memory. The diagram below is an example of how the netX firmware may be organized. The firmware in the diagram is comprised of the system and handshake channel, two netX communication channels and an application 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). A netX can have different independently operating protocol stacks or user applications, 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.

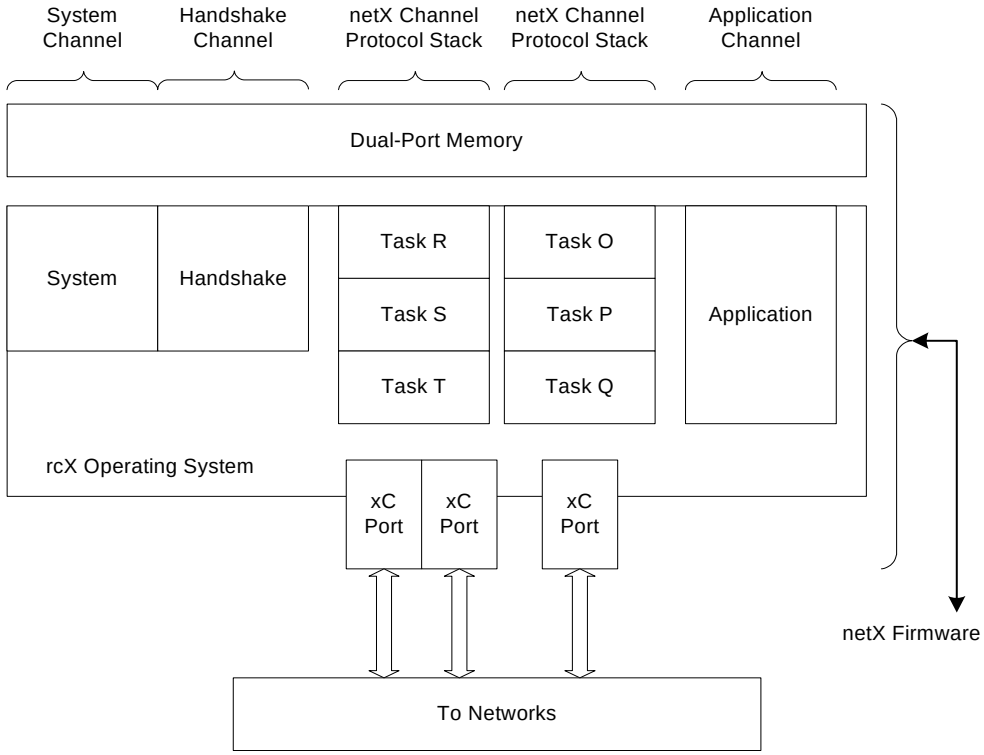


Figure 1 - netX Firmware Block Diagram (Example)

- **Host**
  System with CPU that communicates over the dual-port memory with the netX
- **netX**
  high integrated network controller with ARM CPU, dual-port memory and integrated communication controllers
- **Host Application**
  program that runs on the host controller, typically a PLC program
- **netX Application**
  program which runs on netX, typically a protocol stack
- **Dual-Port Memory**
  interface to the host application in a dual processor system

- **Communication Channel**
  area of the dual-port memory that holds all data structures used to provide communication between host and netX application, typically a communication channel is assigned to one protocol stack
- **Mailbox**
  part of a communication channel to exchange non-cyclic data using handshake cells for synchronization
- **Area**
  process data image or other data structures of a communication channel using handshake cells for synchronization
- **Port**
  serial interface to the network, typically a netX protocol stack that handles one port or, in case of Ethernet with integrated hub / switch functionality, two ports

2.3 Dual-Port Memory Layout

The block diagram below gives an overview of how the netX protocol stacks communicate over certain areas of the dual-port memory, called channels, with the host application. For the default Layout, each of the channels has a fixed length. The system channel and the handshake channel have a fixed length, too. The diagram below shows the default dual-port memory layout for channel 1.

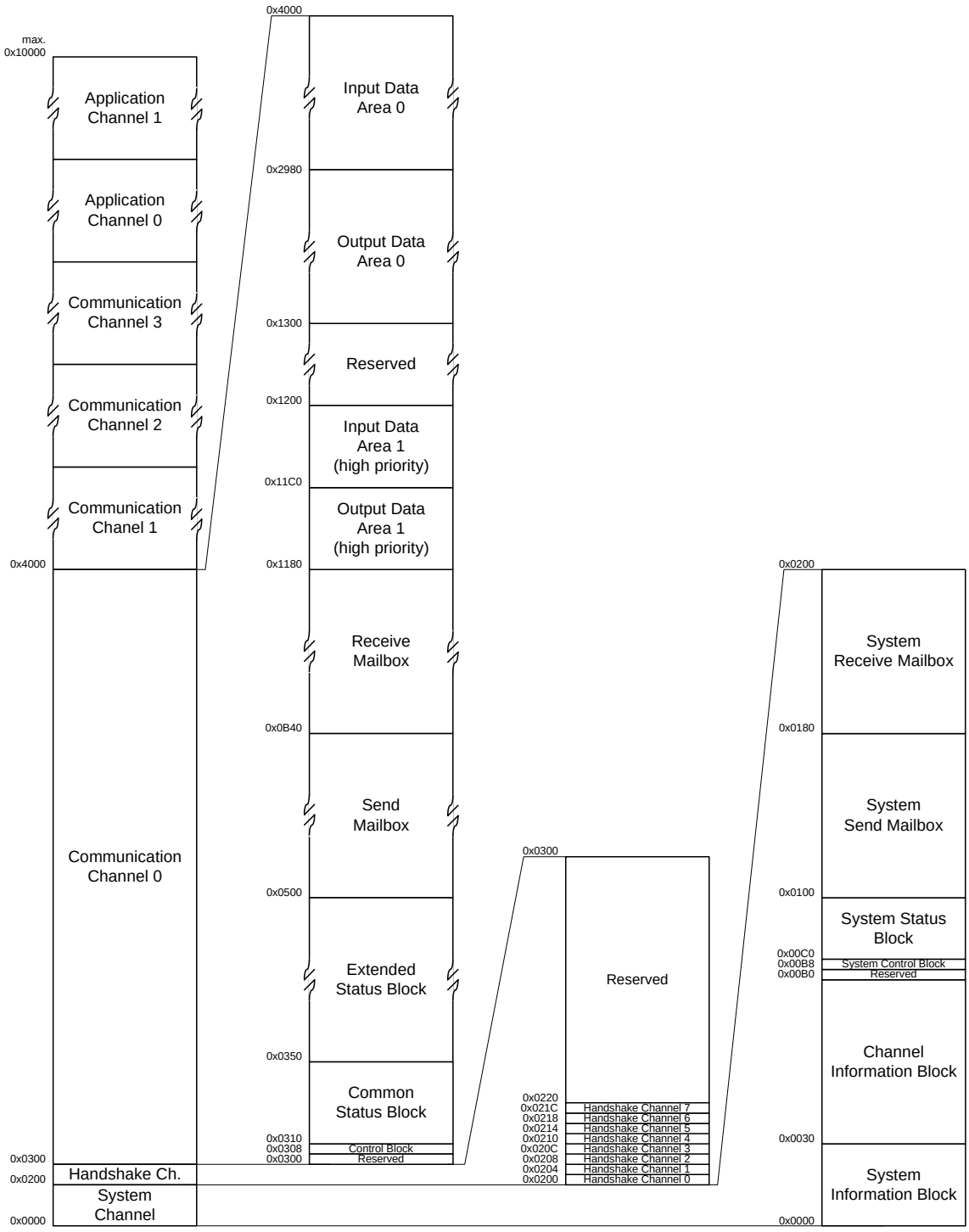


Figure 2 - Block Diagram Default Dual-Port Memory Layout

2.3.1 Default Dual-Port Memory Layout

The netX features a compact layout for small host systems. With the preceding system memory area and handshake channel (see below), its total size is 16 KByte and starts at offset address 0x0300. It supports only one communication channel. The protocol stack will set the *default memory map* flag in the *ulSystemCOS* variable in system status block on page 44 if the default memory layout is used. If the *default memory map* flag is cleared, the layout of the dual-port memory is variable in its size and location.

Default Dual-Port Memory Layout (Communication Channel)			
Block Name	**Offset**	**Size**	**Description**
Reserved	0x0300	8 Bytes	Reserved for Future Use, Set to Zero
Control	0x0308	8 Bytes	Control (see page 52 for details)
Common Status	0x0310	64 Bytes	Status Information Regarding the Protocol Stack (see page 54 for details)
Extended Status	0x0350	432 Bytes	Network Specific Information (see page 60 for details)
Send Mailbox	0x0500	1600 Bytes	Send Mailbox Structure for Non-Cyclic Data Transfer and Diagnostic (see page 60 for details)
Receive Mailbox	0x0B40	1600 Bytes	Receive Mailbox Structure for Non-Cyclic Data Transfer and Diagnostic (see page 60 for details)
Output Data Image 1	0x1180	64 Bytes	High Priority Cyclic Output Data Image (not yet supported)
Input Data Image 1	0x11C0	64 Bytes	High Priority Cyclic Input Data Image (not yet supported)
Reserved	0x1200	256 Bytes	Reserved for Future Use, Set to Zero
Output Data Image 0	0x1300	5760 Bytes	Cyclic Output Data Image for Process Data (see page 63 for details)
Input Data Image 0	0x2980	5760 Bytes	Cyclic Input Data Image for Process Data (see page 63 for details)

Table 2 - Communication Channel (Default Memory Map)

2.3.2 Dual-Port Memory Channels

Typical fieldbus configurations have to one or more xMAC/xPEC ports assigned to them. In addition, the netX dual-port memory concept supports two rcX applications that are not working with xMAC/xPEC ports at all. All of theses applications use memory areas, so called *channels*, which are mapped into the dual-port memory.

Count	Channel used for	Description	Default Size
1	SYSTEM	Access to the netX Operating System	512 Bytes
1	HANDSHAKE	Handshake Flags for Channels	256 Bytes
4	COMMUNICATION	Communication Channel Assigned to a Protocol Stack Using One or More Ports (xMAC/xPEC)	14848 Bytes
2	APPLICATION	Application Channel Using Memory Area for User Tasks Running on the netX Chip	Variable (max 4864)

Table 3 - Memory Blocks

The system channel is always present. It provides information about the structure of the dual-port memory and allows a basic communication via system mailboxes.

The netX firmware is variable in regards of its actual memory layout. Information about the location and size of a certain channels can be obtained via the system mailboxes via messages. The size of a channel is always a multiple of 256 bytes.

Memory Configuration		
Channel	**Size**	**Description**
System Channel	512 Bytes	System Information, Status and Control Blocks, Mailboxes
Handshake Channel	256 Bytes	Handshake Flags for Host and netX, Change of State Mechanism (COS)
Communication Channel 0	Variable m • 256 Bytes	I/O Data, None Cyclic Data Exchange, Diagnostic Data of the Protocol stack Running on Channel 0
Communication Channel 1	Variable n • 256 Bytes	I/O Data, None Cyclic Data Exchange, Diagnostic Data of the Protocol stack Running on Channel 1
Communication Channel 2	Variable p • 256 Bytes	I/O Data, None Cyclic Data Exchange, Diagnostic Data of the Protocol stack Running on Channel 2
Communication Channel 3	Variable q • 256 Bytes	I/O Data, None Cyclic Data Exchange, Diagnostic Data of the Protocol stack Running on Channel 3
Application Channel 0	Variable r • 256 Bytes	Custom Specific (Application)
Application Channel 1	Variable s • 256 Bytes	Custom Specific (Application)

Table 4 - Memory Configuration

The application/driver obtains information regarding the actual structure of the dual-port memory and configures itself accordingly.

2.3.2.1 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 not 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. By default the mailbox structure is 128 bytes in size for the send mailbox and 128 bytes for the receive mailbox.

System Channel		
Block Name	**Size**	**Description**
System Information Block	48 Bytes	System Information Area
Channel Information Block	128 Bytes	Contains Configuration Information About Available Communication and Application Channels and their Data Blocks
Reserved System Handshake Block	8 Bytes	Reserved Handshake Block for Handshake Cells
System Control Block	8 Bytes	System Control and Commands
System Status Block	64 Bytes	System Status Information
System Mailboxes	256 Bytes	System Packet Mailbox Area (Always Located at the End of the System Block, see Table 15)

Table 5 - System Channel

2.3.2.2 Communication Channels

Each of the communication channels can have the following elements:

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

A communication channel follows the preceding channel without gaps. Depending on the implementation, sub areas (or blocks) may or may not be present for a communication channel. This is contrary to the default memory layout as outlined on page 18, where all of the above blocks are mandatory. In the variable layout, the *Control* and *Common Status* blocks are mandatory and always present. Structure and the size of these blocks are fixed. The *Extended Status* block is optional and may or may not be present. The *Send* and *Receive Mailbox* are mandatory and always present, but

variable in its size. Depending on the implementation, *Input* and *Output Data Images* may or may not be present. They have a variable size.

2.3.2.3 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 bits allow 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.

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

2.3.2.4 Communication Channel

The communication channel area in the dual-port memory is used by fieldbus stacks. Those stacks use this area to exchange their cyclic process data with the host application and

If the default memory layout is used, the protocol stack will set the *default memory map* flag in the *ulSystemCOS* variable in system status block. See page 18 for the default dual-port memory layout. When the variable approach of configuring the dual-port memory is used, the rcX operating system calculates the layout of the channel and its blocks based on the configuration database. rcX creates a memory map of the smallest possible size. Individual channel areas follow the previous area without gaps.

Today all functions to obtain the memory map, as described on page 92, are in place. This functionality does not change for an application, independently whether it uses the default memory map as outlined above or it is configurable dynamically in a next step.

2.3.2.5 Application Channels

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

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 features 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 set to true or both set to false) the application has access rights to the memory area or sub-area. If both have a different value, the netX firmware has access rights. The following table illustrates this mechanism.

Host	CMD Flag	ACK Flag	netX
Host System Has Access	**0**	**0**	netX Has NO Access
Host System Has NO Access	**0**	**1**	netX Has Access
Host System Has NO Access	**1**	**0**	netX Has Access
Host System Has Access	**1**	**1**	netX Has NO Access

Table 6 - Command and Acknowledge

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.

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, if host application whishes to lock the configuration settings of a communication channel, 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. See flowchart below.

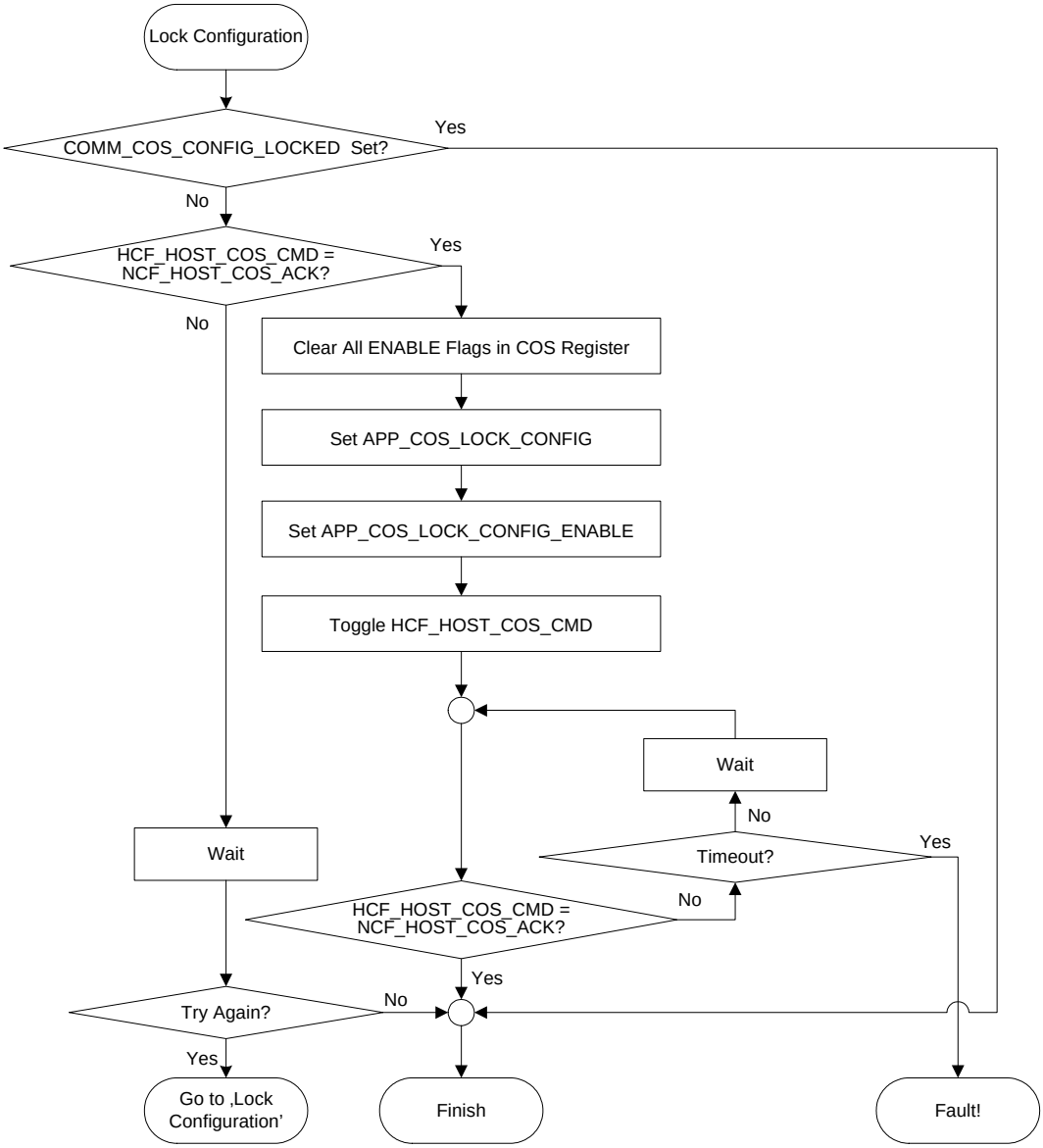


Figure 3 - Lock Configuration (Example Using Enable Flag)

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 whishes 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.

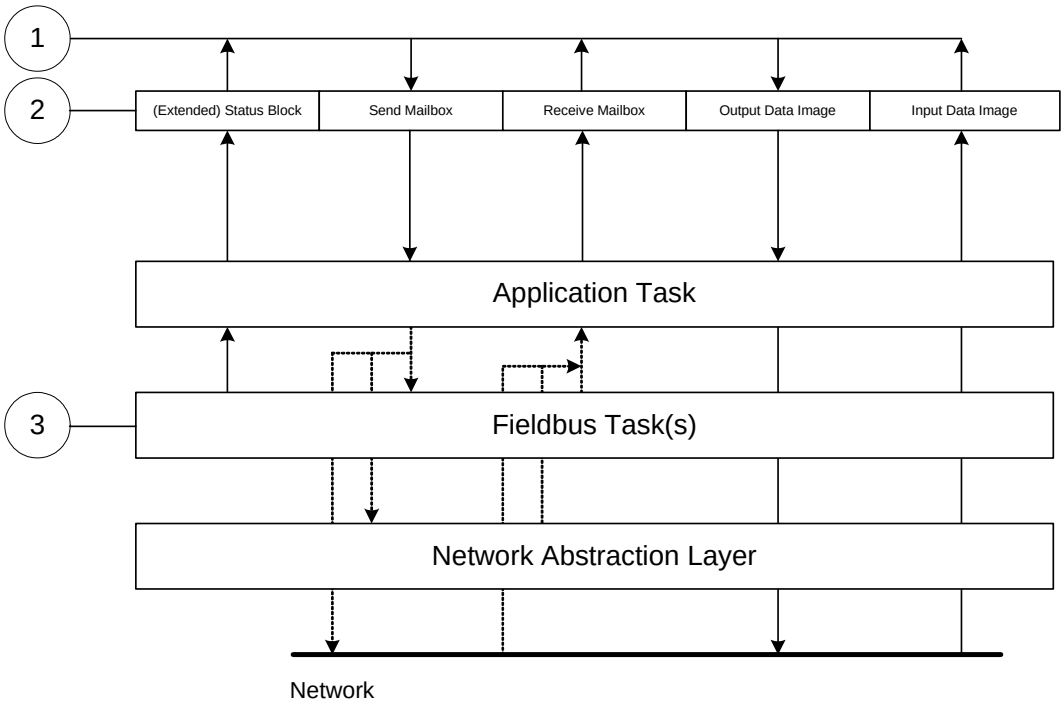


Figure 4 - Accessing a Protocol Stack

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. 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.

System Channel			
Offset	**Type**	**Name**	**Description**
0x0000	Structure	`tSystemInfo`	System Information Block Identifies netX Dual-Port Memory (see page 28)
0x0030	Structure	`tChannelInfo`	Channel Information Block Contains Configuration Information About Available Communication and Application Channel Blocks (see page 35)
0x00B0	Structure	`tReserved`	Handshake Block Handshake Block for Handshake Cells (not used, set to zero)
0x00B8	Structure	`tSystemControl`	System Control Block System Control and Commands (see page 43)
0x00C0	Structure	`tSystemStatus`	System Status Block System Status Information (see page 44)
0x0100	Structure	`tSystemSendMailbox` `tSystemRecvMailbox`	System Mailboxes System Send and Receive Packet Mailbox Area, Always Located at the End of the System Block (see page 46)

Table 7 - System Channel

System Channel Structure Reference

```
typedef struct NETX_SYSTEM_CHANNEL_Ttag
{
  NETX_SYSTEM_INFO_BLOCK_T           tSystemInfo;
  NETX_SYSTEM_CHANNEL_INFO_BLOCK_T   tChannelInfo;
  NETX_SYSTEM_HANDSHAKE_BLOCK_T      tReserved;
  NETX_SYSTEM_CONTROL_BLOCK_T        tSystemControl;
  NETX_SYSTEM_STATUS_BLOCK_T         tSystemStatus;
  NETX_SYSTEM_SEND_MAILBOX_T         tSystemSendMailbox;
  NETX_SYSTEM_RECV_MAILBOX_T         tSystemRecvMailbox;
} NETX_SYSTEM_CHANNEL_T;
```

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 67 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 29)
0x0008	UINT32	`ulDeviceNumber`	Device Number Device Number / Identification (see page 29)
0x000C	UINT32	`ulSerialNumber`	Serial Number Serial Number (see page 29)
0x0010	UINT16	`ausHwOptions[4]`	Hardware Options Hardware Assembly Option (see page 29)
0x0018	UINT16	`usManufacturer`	Manufacturer Manufacturer Code / Manufacturer Location (see page 31)
0x001A	UINT16	`usProductionDate`	Production Date Production Date (see page 31)
0x001C	UINT32	`ulLicenseFlags1`	License Code License Flags 1 (see page 32)
0x0020	UINT32	`ulLicenseFlags2`	License Code License Flags 2 (see page 32)
0x0024	UINT16	`usNetxLicenseID`	License Code netX License Identification (see page 32)
0x0026	UINT16	`usNetxLicenseFlags`	License Code netX License Flags (see page 32)
0x0028	UINT16	`usDeviceClass`	Device Class netX Device Class (see page 33)
0x002A	UINT8	`bHwRevision`	Hardware Revision Hardware Revision Index (see page 34)
0x002B	UINT8	`bHwCompatibility`	Hardware Compatibility Hardware Compatibility Index (not supported yet) (see page 34)
0x002C … 0x002F	UINT16	`ausReserved[2]`	Reserved Set to Zero

Table 8 - System Information Block

System Information Block Structure Reference

```
typedef struct NETX_SYSTEM_INFO_BLOCK_Ttag
{
  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       */
  UINT16  ausReserved[2];
} NETX_SYSTEM_INFO_BLOCK_T;
```

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 staring at address 0x0300.

- BAD MEMORY COOKIE `#define RCX_SYS_BAD_MEMORY_COOKIE 0x0BAD`

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 47).

Device Number, Device Identification

This field holds a device identification or item number.

Example:
A value of 1.234.567.890 (= 0x499602D2) translates into a device 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
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
0xFFF4	I^2C INTERFACE UNKNOWN The xC port is marked I^2C INTERFACE UNKNOWN if the physical interface cannot be determined (e.g. option module is not connected)
0xFFF5	SSI INTERFACE
0xFFF6	SYNC INTERFACE
0xFFFA	TOUCH SCREEN

0xFFFB	I^2C INTERFACE The xC port is marked I^2C 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^2C INTERFACE UNKNOWN (if not found) or the detected hardware assembly is displayed
0xFFFC	I^2C INTERFACE netTAP The xC port is marked I^2C INTERFACE netTAP for an option module on netTAP hardware basis. This value, however, is never shown in the hardware assembly option field
0xFFFD	PROPRIETARY INTERFACE
0xFFFE	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
0xFFFF	RESERVED, DO NOT USE
else	Othere are reserved

Manufacturer

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

- UNDEFINED `#define RCX_MANUFACTURER_UNDEFINED 0x0000`
  The module has not been personalized yet.
- Hilscher Gesellschaft für Systemautomation mbH
  `#define RCX_MANUFACTURER_HILSCHER_GMBH 0x0001`
- Codes ranging from 1 to 255 are reserved for Hilscher Gesellschaft für Systemautomation mbH

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.

High Byte	Low Byte
	Production (Calendar) Week (Range: 01 to 52)
Production Year (Range: 00 to 255)	

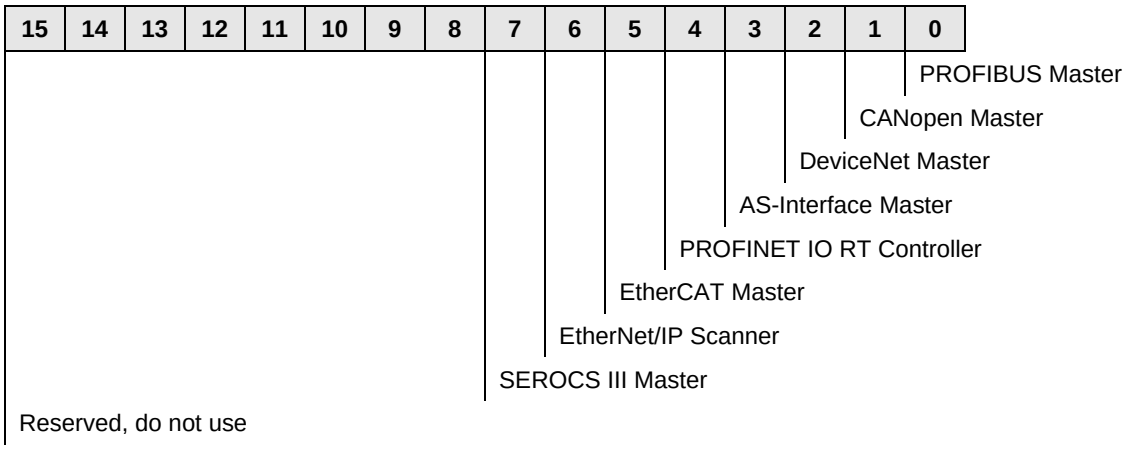
Example:
An *usProductionDate* of 0x062B indicates year 2006 and calendar week 43.

License Code

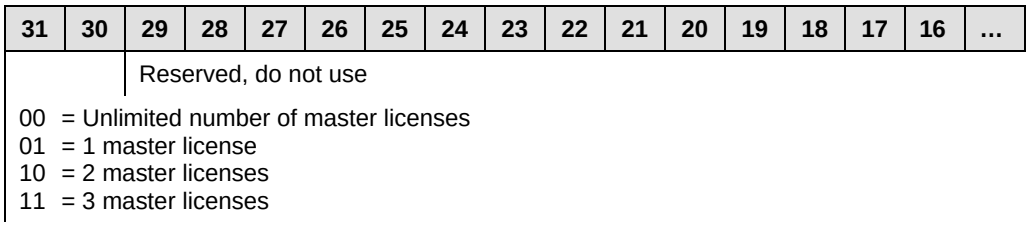
These fields contain licensing information that is available for the netX firmware and tools. All four fields (License Flags 1, License Flags 2, netX License ID & netX License Flags) help identifying available licenses. If the license information fields are equal to zero, a license or license code is 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`

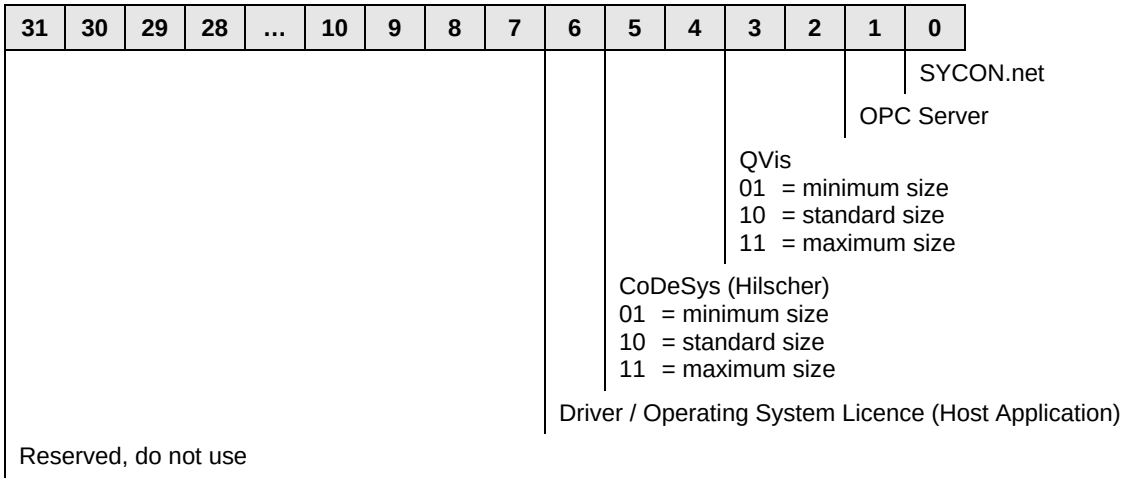


`ulLicenseFlags1` (continued)



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`



netX License ID holds a customer identification number.

netX License Flags are resevered.

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
0x0005	NETX EVALUATION BOARD
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
0x0014	NPLC-C100

0x0015	NPLC-M100
0x0016	NETTAP 50 (Gateway)
0xFFFE	OEM DEVICE
0x0017 … 0xFFFD, 0xFFFF	Othere are reserved

Hardware Revision

This 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

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.

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 66 for details).

Channel Information Block			
Address	**Channel**	**Area Structure**	
		Data Type	**Description**
0x0030	**System**	UINT8	Channel Type = SYSTEM (see page 37)
		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
… 0x003F		UINT8[4]	4 Byte Reserved (set to zero)
		Data Type	**Description**
0x0040	**Handshake**	UINT8	Channel Type = HANDSHAKE (see page 37)
		UINT8[3]	3 Byte Reserved (set to zero)
		UINT32	Channel Size in Bytes
… 0x004F		UINT8[8]	8 Byte Reserved
		Data Type	**Description**
0x0050	**Communication Channel 0**	UINT8	Channel Type = COMMUNICATION (see page 37)
		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	Protocol Class (PROFIBUS, PROFINET....)
		UINT16	Protocol Conformance Class (DPV1, DPV2...)
… 0x005F		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	**Application Channel 0**	UINT8	Channel Type = APPLICATION (see page 37)
		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	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 9 - Channel Information Block

System Channel Information Structure Reference

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

Handshake Channel Information Structure Reference

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

Communication Channel Information Structure Reference

```
typedef struct NETX_COMMUNICATION_CHANNEL_INFO_Ttag
{
  UINT8    bChannelType;
  UINT8    bChannelId;
  UINT8    bSizePositionOfHandshake;
  UINT8    bNumberOfBlocks;
  UINT32   ulSizeOfChannel;
  UINT16   usCommunicationClass;
  UINT16   usProtocolClass;
  UINT16   usConformanceClass;
  UINT16   usReserved;
} NETX_COMMUNICATION_CHANNEL_INFO_T;
```

Application Channel Information Structure Reference

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

Channel Information Block Structure Reference

```
typedef struct NETX_CHANNEL_INFO_BLOCK_Ttag
{
  NETX_SYSTEM_CHANNEL_INFO_T          tSystemChannel;
  NETX_HANDSHAKE_CHANNEL_INFO_T       tHandshakeChannel;
  NETX_COMMUNICATION_CHANNEL_INFO_T   tCommunicationChannel0;
  NETX_COMMUNICATION_CHANNEL_INFO_T   tCommunicationChannel1;
  NETX_COMMUNICATION_CHANNEL_INFO_T   tCommunicationChannel2;
  NETX_COMMUNICATION_CHANNEL_INFO_T   tCommunicationChannel3;
  NETX_APPLICATION_CHANNEL_INFO_T     tApplicationChannel0;
  NETX_APPLICATION_CHANNEL_INFO_T     tApplicationChannel1;
} NETX_CHANNEL_INFO_BLOCK_T;
```

Channel Type

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

■ UNDEFINED	`#define RCX_CHANNEL_TYPE_UNDEFINED`	`0x00`
■ NOT AVAILABLE	`#define RCX_CHANNEL_TYPE_NOT_AVAILABLE`	`0x01`
■ RESERVED	`#define RCX_CHANNEL_TYPE_RESERVED`	`0x02`
■ SYSTEM	`#define RCX_CHANNEL_TYPE_SYSTEM`	`0x03`
■ HANDSHAKE	`#define RCX_CHANNEL_TYPE_HANDSHAKE`	`0x04`
■ COMMUNICATION	`#define RCX_CHANNEL_TYPE_COMMUNICATION`	`0x05`
■ APPLICATION	`#define RCX_CHANNEL_TYPE_APPLICATION`	`0x06`
■ Reserved for future use		`0x07 … 0x7F`
■ User defined		`0x80 … 0xFF`

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.

Size

■ SIZE MASK	`#define RCX_HANDSHAKE_SIZE_MASK`	`0x0F`
■ NOT AVAILABLE	`#define RCX_HANDSHAKE_SIZE_NOT_AVAILABLE`	`0x00`
■ 8 BITS	`#define RCX_HANDSHAKE_SIZE_8_BITS`	`0x01`
■ 16 BITS	`#define RCX_HANDSHAKE_SIZE_16_BITS`	`0x02`

Position

■ POSITION MASK	`#define RCX_HANDSHAKE_POSITION_MASK`	`0xF0`
■ BEGINNING OF CHANNEL	`#define RCX_HANDSHAKE_POSITION_BEGINNING`	`0x00`
■ IN HANDSHAKE CHANNEL	`#define RCX_HANDSHAKE_POSITION_CHANNEL`	`0x10`

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 46 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.

■ UNDEFINED	`#define RCX_COMM_CLASS_UNDEFINED`	`0x0000`
■ UNCLASSIFIABLE	`#define RCX_COMM_CLASS_UNCLASSIFIABLE`	`0x0001`
■ MASTER	`#define RCX_COMM_CLASS_MASTER`	`0x0002`
■ SLAVE	`#define RCX_COMM_CLASS_SLAVE`	`0x0003`
■ SCANNER	`#define RCX_COMM_CLASS_SCANNER`	`0x0004`
■ ADAPTER	`#define RCX_COMM_CLASS_ADAPTER`	`0x0005`
■ MESSAGING	`#define RCX_COMM_CLASS_MESSAGING`	`0x0006`
■ CLIENT	`#define RCX_COMM_CLASS_CLIENT`	`0x0007`
■ SERVER	`#define RCX_COMM_CLASS_SERVER`	`0x0008`

- IO-CONTROLLER `#define RCX_COMM_CLASS_IO_CONTROLLER 0x0009`
- IO-DEVICE `#define RCX_COMM_CLASS_IO_DEVICE 0x000A`
- IO-SUPERVISOR `#define RCX_COMM_CLASS_IO_SUPERVISOR 0x000B`
- GATEWAY `#define RCX_COMM_CLASS_GATEWAY 0x000C`
- MONITOR / ANALYZER `#define RCX_COMM_CLASS_MONITOR 0x000D`
- PRODUCER `#define RCX_COMM_CLASS_PRODUCER 0x000E`
- CONSUMER `#define RCX_COMM_CLASS_CONSUMER 0x000F`
- SWITCH `#define RCX_COMM_CLASS_SWITCH 0x0010`
- HUB `#define RCX_COMM_CLASS_HUB 0x0011`
- COMBINATION FIRMWARE `#define RCX_COMM_CLASS_COMBI 0x0012`
  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.
- MANAGING NODE `#define RCX_COMM_CLASS_MANAGING_NODE 0x0013`
- CONTROLLED NODE `#define RCX_COMM_CLASS_CONTROLLED_NODE 0x0014`
- Others are reserved.

Protocol and Task Class

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

- UNDEFINED `#define RCX_PROT_CLASS_UNDEFINED 0x0000`
- 3964R `#define RCX_PROT_CLASS_3964R 0x0001`
- AS Interface `#define RCX_PROT_CLASS_ASINTERFACE 0x0002`
- ASCII `#define RCX_PROT_CLASS_ASCII 0x0003`
- CANopen `#define RCX_PROT_CLASS_CANOPEN 0x0004`
- CC-Link `#define RCX_PROT_CLASS_CCLINK 0x0005`
- CompoNet `#define RCX_PROT_CLASS_COMPONET 0x0006`
- ControlNet `#define RCX_PROT_CLASS_CONTROLNET 0x0007`
- DeviceNet `#define RCX_PROT_CLASS_DEVICENET 0x0008`
- EtherCAT `#define RCX_PROT_CLASS_ETHERCAT 0x0009`
- EtherNet/IP `#define RCX_PROT_CLASS_ETHERNET_IP 0x000A`
- Foundation Fieldbus `#define RCX_PROT_CLASS_FOUNDATION_FB 0x000B`
- FL Net `#define RCX_PROT_CLASS_FL_NET 0x000C`
- InterBus `#define RCX_PROT_CLASS_INTERBUS 0x000D`
- IO-Link `#define RCX_PROT_CLASS_IO_LINK 0x000E`
- LON `#define RCX_PROT_CLASS_LON 0x000F`
- Modbus Plus `#define RCX_PROT_CLASS_MODBUS_PLUS 0x0010`
- Modbus RTU `#define RCX_PROT_CLASS_MODBUS_RTU 0x0011`
- Open Modbus TCP `#define RCX_PROT_CLASS_OPEN_MODBUS_TCP 0x0012`
- PROFIBUS DP `#define RCX_PROT_CLASS_PROFIBUS_DP 0x0013`
- PROFIBUS MPI `#define RCX_PROT_CLASS_PROFIBUS_MPI 0x0014`

■ PROFINET IO	`#define RCX_PROT_CLASS_PROFINET_IO`	`0x0015`
■ RK512	`#define RCX_PROT_CLASS_RK512`	`0x0016`
■ SERCOS II	`#define RCX_PROT_CLASS_SERCOS_II`	`0x0017`
■ SERCOS III	`#define RCX_PROT_CLASS_SERCOS_III`	`0x0018`
■ TCP/IP, UDP/IP	`#define RCX_PROT_CLASS_TCP_IP_UDP_IP`	`0x0019`
■ Powerlink	`#define RCX_PROT_CLASS_POWERLINK`	`0x001A`
■ HART	`#define RCX_PROT_CLASS_HART`	`0x001B`
■ 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.	`#define RCX_PROT_CLASS_COMBI`	`0x001C`
■ Programmable Gateway The programmable gateway function uses netSCRIPT as programming language.	`#define RCX_PROT_CLASS_PROG_GATEWAY`	`0x001D`
■ Programmable Serial The programmable serial protocol function uses netSCRIPT as programming language.	`#define RCX_PROT_CLASS_PROG_SERIAL`	`0x001E`
■ PLC: CoDeSys	`#define RCX_PROT_CLASS_PLC_CODESYS`	`0x001F`
■ PLC: ProConOS	`#define RCX_PROT_CLASS_PLC_PROCONOS`	`0x0020`
■ PLC: IBH S7	`#define RCX_PROT_CLASS_PLC_IBH_S7`	`0x0021`
■ PLC: ISaGRAF	`#define RCX_PROT_CLASS_PLC_ISAGRAF`	`0x0022`
■ Visualization: QVis	`#define RCX_PROT_CLASS_VISU_QVIS`	`0x0023`
■ Ethernet	`#define RCX_PROT_CLASS_ETHERNET`	`0x0024`
■ OEM, Proprietary	`#define RCX_PROT_CLASS_OEM`	`0xFFF0`
■ Others are reserved		

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 is 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 cells are located in the handshake channel (see page 64). They are used to synchronize data transfer via the system mailbox and to handle the 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.

For the default layout (page 47), the handshake registers are located in the handshake channel.

3.1.3.1 netX System Flags

The netX system register is written by the netX; the host system reads this register.

`bNetxSysFlags` – netX writes, Host reads

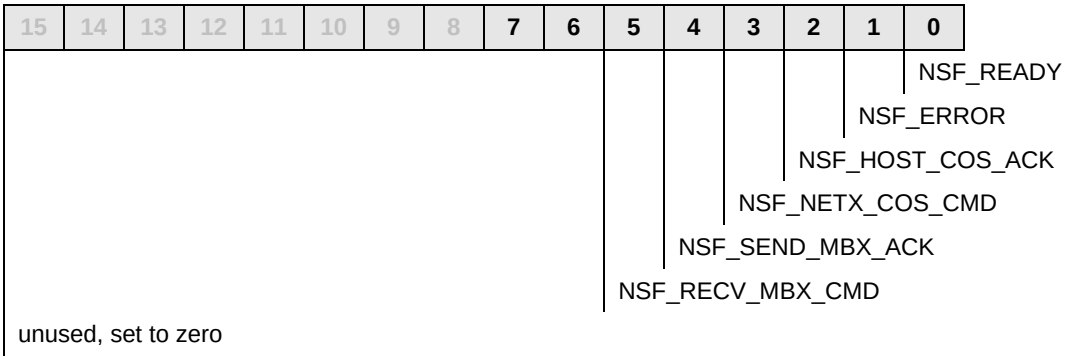


Table 10 - netX System Flags

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

netX System Flags (netX ⇨ Host System)

- READY `#define NSF_READY 0x0001`
  The *Ready* 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.
- ERROR `#define NSF_ERROR 0x0002`
  The *Error* flag is set when the netX has detected an internal error condition. This is considered to be a fatal error. The *Ready* flag is cleared and the operating system is stopped. An error code helping to identify the issue is placed in the *ulSystemError* variable in the system status block (see page 44).
- HOST CHANGE OF STATE ACKNOWLEDGE
  `#define NSF_HOST_COS_ACK 0x0004`
  The *Host Change of State Acknowledge* flag is set when the netX acknowledges a command from the host system. This flag is used together with the *Host Change of State Command* flag in the host system flags on page 42.
- NETX CHANGE OF STATE COMMAND
  `#define NSF_NETX_COS_CMD 0x0010`
  The *netX Change of State Command* 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 *ulSystemCOS* variable in the system control block (see page 43).

- SEND MAILBOX ACKNOWLEDGE
  `#define NSF_SEND_MBX_ACK 0x0020`
  Both the *Send Mailbox Acknowledge* flag and the *Send Mailbox Command* flag are used together to transfer non-cyclic packages between the host system and the netX.
- RECEIVE MAILBOX COMMAND
  `#define NSF_RECV_MBX_CMD 0x0040`
  Both the *Receive Mailbox Command* flag and the *Receive Mailbox Acknowledge* flag are used together to transfer non-cyclic packages between the netX and the host.

Detail on the process data handshake mechanism can be found on page 78.

3.1.3.2 Host System Flags

The host system flags are written by the host system; the netX reads these flags.

`bHostSysFlags` – Host writes, netX reads

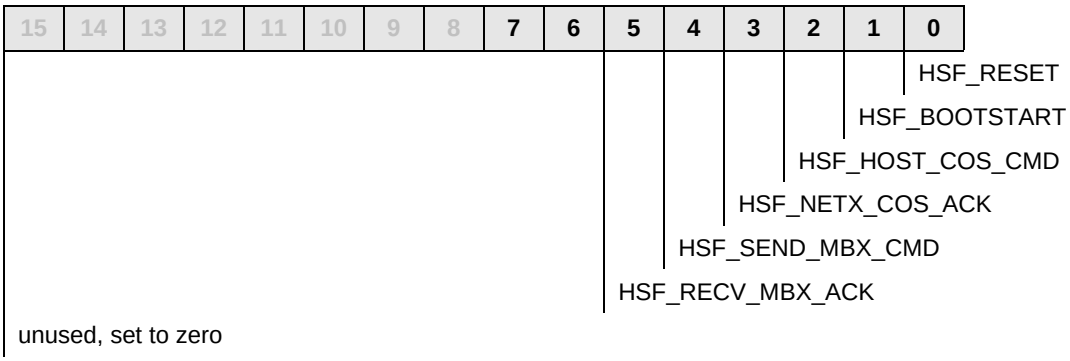


Table 11 - netX System Flags

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

Host System Flags (Host ⇨ netX System)

- RESET `#define HSF_RESET 0x0001`
  The *Reset* 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.
- BOOT START `#define HSF_BOOTSTART 0x0002`
  If set during reset, the *Boot-Start* 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.
- HOST CHANGE OF STATE COMMAND
  `#define HSF_HOST_COS_CMD 0x0004`
  The *Host Change of State Command* 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 *ulSystemCommandCOS* variable in the system control block (see page 43).
- NETX CHANGE OF STATE ACKNOWLEDGE
  `#define HSF_NETX_COS_ACK 0x0008`
  The *netX Change of State Acknowledge* flag is set by the host system to acknowledge the new state of the netX. This flag is used together with the n*etX Change of State Command* flag in the netX system flags on page 41.

- SEND MAILBOX COMMAND `#define HSF_SEND_MBX_CMD 0x0010`
  Both the *Send Mailbox Command* flag and the *Send Mailbox Acknowledge* flag are used together to transfer non-cyclic packages between the host system and the netX.
- RECEIVE MAILBOX ACKNOWLEDGE
  `#define HSF_RECV_MBX_ACK 0x0020`
  Both the *Receive Mailbox Acknowledge* flag and the *Receive Mailbox Command* flag are used together to transfer non-cyclic packages between the netX and the host system.

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 12 -System Handshake Block

System Handshake Block Structure Reference

```
typedef struct NETX_SYSTEM_HANDSHAKE_BLOCK_Ttag
{
  UINT8    abReserved[8];
} NETX_SYSTEM_HANDSHAKE_BLOCK_T;
```

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 67 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 13 - System Control Block

System Control Block Structure Reference

```
typedef struct NETX_SYSTEM_CONTROL_BLOCK_Ttag
{
  UINT32   ulSystemCommandCOS;
  UINT32   ulReserved;
} NETX_SYSTEM_CONTROL_BLOCK_T;
```

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 42). 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 67 for details).

System Status Block			
Offset	**Type**	**Name**	**Description**
0x00C0	UINT32	`ulSystemCOS`	System Change Of State DEFAULT LAYOUT (see page 44)
0x00C4	UINT32	`ulSystemStatus`	System Status (not supported yet)
0x00C8	UINT32	`ulSystemError`	System Error Indicates Success or an Error Code
0x00CC	UINT32	`ulReserved`	Reserved Set To 0
0x00D0	UINT32	`ulTimeSinceStart`	Time Since Startup Time Elapsed Since Startup (POR) in s
0x00D4	UINT16	`usCpuLoad`	CPU Load CPU Load in 0.01% Units
0x00D6 … 0x00FF	UINT8	`abReserved[42]`	Reserved Set to 0

Table 14 - System Status Block

System Status Block Structure Reference

```
typedef struct NETX_SYSTEM_STATUS_BLOCK_Ttag
{
  UINT32   ulSystemCOS;
  UINT32   ulSystemStatus;
  UINT32   ulSystemError;
  UINT32   ulReserved;
  UINT32   ulTimeSinceStart;
  UINT16   usCpuLoad;
  UINT8    abReserved[42];
} NETX_SYSTEM_STATUS_BLOCK_T;
```

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.

- UNDEFINED `#define RCX_SYS_COS_UNDEFINED 0x00000000`
- DEFAULT MEMORY MAP `#define RCX_SYS_COS_DEFAULT_MEMORY 0x80000000`
  If set, the default dual-port memory layout as outlined on page 47 is applied. This bit is set once after power up or reset and changes only after reconfiguration.
- Others are reserved.

System Status

The system status field holds information regarding netX operating system rcX. The value indicates the current state the rcX is in. Currently not supported and set to 0.

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 7 on page 203 for error codes.

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. A value of 10000 corresponds to 100%. Therefore the resolution is 0.01%.

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 66 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 48 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 15 - System Mailbox

System Mailbox Structure Reference

```
typedef struct NETX_SYSTEM_SEND_MAILBOX_Ttag
{
  UINT16   usPackagesAccepted;
  UINT16   usReserved;
  UINT8    abSendMbx[124];
} NETX_SYSTEM_SEND_MAILBOX_T;

typedef struct NETX_SYSTEM_RECV_MAILBOX_Ttag
{
  UINT16   usWaitingPackages;
  UINT16   usReserved;
  UINT8    abRecvMbx[124];
} NETX_SYSTEM_RECV_MAILBOX_T;
```

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 44. If the *default memory map* flag is cleared, the layout of the dual-port memory is variable in its size and location.

Default Communication Channel Layout			
Offset	**Type**	**Name**	**Description**
0x0300	Structure	tReserved	Reserved See Page 51 for Details
0x0308	Structure	tControl	Control See Page 52 for Details
0x0310	Structure	tCommonStatus	Common Status Block See Page 54 for Details
0x0350	Structure	tExtendedStatus	Extended Status Block See Page 60 for Details
0x0500	Structure	tSendMbx	Send Mailbox See Page 60 for Details
0x0B40	Structure	tRecvMbx	Receive Mailbox See Page 60 for Details
0x1180	UINT8	abPd1Output[64]	High Priority Output Data Image 1 Not Yet Supported, Set to 0
0x11C0	UINT8	abPd1Input[64]	High Priority Input Data Image 1 Not Yet Supported, Set to 0
0x1200	UINT8	abReserved1[256]	Reserved Set to 0
0x1300	UINT8	abPd0Output[5760]	Output Data Image 0 See Page 63 for Details
0x2980	UINT8	abPd0Input[5760]	Input Data Image 0 See Page 63 for Details

Table 16 - Default Communication Channel Layout

Default Communication Channel Structure Reference

```
typedef struct NETX_DEFAULT_COMM_CHANNEL_Ttag
{
  NETX_HANDSHAKE_BLOCK_T         tReserved;
  NETX_CONTROL_BLOCK_T           tControl;
  NETX_COMMON_STATUS_BLCOK_T     tCommonStatus;
  NETX_EXTENDED_STATUS_BLOCK_T   tExtendedStatus;
  NETX_SEND_MAILBOX_BLOCK_T      tSendMbx;
  NETX_RECV_MAILBOX_BLOCK_T      tRecvMbx;
  UINT8                          abPd1Output[64];
  UINT8                          abPd1Input[64];
  UINT8                          abReserved1[256];
  UINT8                          abPd0Output[5760];
  UINT8                          abPd0Input[5760];
} NETX_DEFAULT_COMM_CHANNEL_T;
```

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 for instance). The process data handshake flags are used only if the buffered handshake mode is configured (see page 78 for details). The mailbox flags are used to send and receive non-cyclic messages via the channel mailboxes. See page 60 for details to the command/acknowledge mechanism.

For the default layout, (see page 47) the handshake registers are located in the handshake channel (see page 64).

3.2.2.1 netX Communication Flags

The netX protocol stack writes the register; the register is read by the host system.

`usNetxCommFlags` - netX writes, Host reads

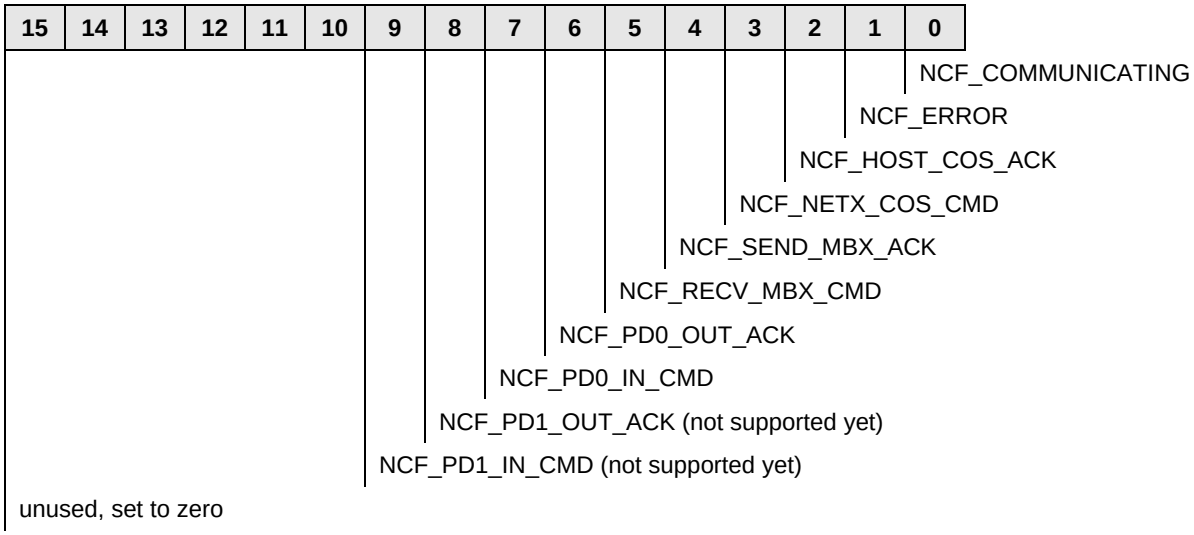


Table 17 - netX Communication Channel Flags

netX Communication Flags (netX ⇨ Application)

- COMMUNICATING `#define NCF_COMMUNICATING 0x0001`
  The *Communicating* 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.
- ERROR `#define NCF_ERROR 0x0002`
  The *Error* 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 *ulCommunicationError* variable in the common status block (see page 54). At initialization time, this flag is cleared.
- HOST CHANGE OF STATE ACKNOWLEDGE `#define NCF_HOST_COS_ACK 0x0004`
  The *Host Change of State Acknowledge* flag is used by the protocol stack indicate that the new state of the host application has been read. At initialization time, this flag is cleared.

- NETX CHANGE OF STATE COMMAND

  ```
  #define NCF_NETX_COS_CMD                0x0008
  ```

  The n*etX Change of State Command* flag signals a change in the state of the protocol stack. The new state can be found in the *ulCommunicationCOS* variable in the common status block (see page 54). The host application then toggles the n*etX Change of State Acknowledge* flag in the host communication flags back acknowledging that the new protocol state has been read. At initialization time, this flag is cleared.

- SEND MAILBOX ACKNOWLEDGE

  ```
  #define NCF_SEND_MBX_ACK                0x0010
  ```

  Both the *Send Mailbox Acknowledge* flag and the *Send Mailbox Command* flag are used together to transfer non-cyclic packages between the protocol stack and the application. At initialization time, this flag is cleared.

- RECEIVE MAILBOX COMMAND

  ```
  #define NCF_RECV_MBX_CMD                0x0020
  ```

  Both the *Receive Mailbox Command* flag and the *Receive Mailbox Acknowledge* flag are used together to transfer non-cyclic packages between the application and the protocol stack. At initialization time, this flag is cleared.

- PROCESS DATA OUT ACKNOWLEDGE

  ```
  #define NCF_PD0_OUT_ACK                 0x0040
  ```

  (not supported yet: `#define RCX_NCF_PD1_OUT_ACK 0x0100`)

  Both the *PDx OUT Acknowledge* flag and the *PDx OUT Command* flag are used together to transfer cyclic data between the application and the protocol stack. ('x' indicates the number of the output/input area.) At initialization time, this flag may be set, depending on the data exchanged mode.

- PROCESS DATA IN COMMAND

  ```
  #define NCF_PD0_IN_CMD                  0x0080
  ```

  (not supported yet: `#define NCF_PD1_IN_CMD 0x0200`)

  Both the *PDx IN Command* flag and the *PDx IN Acknowledge* flag are used together to transfer cyclic data between the protocol stack and the application. ('x' indicates the number of the output/input area.) At initialization time, this flag may be set, depending on the data exchanged mode.

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

Detail on the process data handshake mechanism can be found on page 78.

3.2.2.2 Host Communication Flags

The register is written by the host system; the register is read by the netX protocol stack.

`usHostCommFlags` - Host writes, netX reads

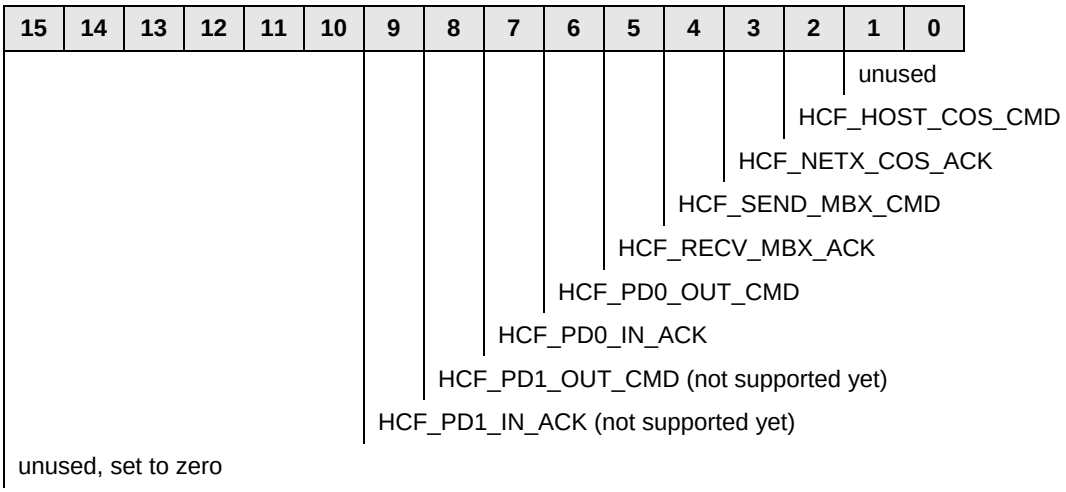


Table 18 - Communication Channel Flags

Host Communication Flags (Application ⇨ netX System)

- HOST CHANGE OF STATE COMMAND

  `#define HCF_HOST_COS_CMD 0x0004`

  The *Host Change of State Command* (COS) flag signals a change in the state of the host application. A new state is set in the *ulApplicationCOS* variable in the communication control block (see page 52). The protocol stack on the netX then toggles the *Host Change of State Acknowledge* flag in the netX communication flags back acknowledging that the new state has been read. At initialization time, this flag is cleared.

- NETX CHANGE OF STATE ACKNOWLEDGE

  `#define HCF_NETX_COS_ACK 0x0008`

  The *netX Change of State Acknowledge* (COS) 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.

- SEND MAILBOX COMMAND

  `#define HCF_SEND_MBX_CMD 0x0010`

  Both the *Send Mailbox Command* flag and the *Send Mailbox Acknowledge* flag are used together to transfer non-cyclic packages between the application and the protocol stack. At initialization time, this flag is cleared.

- RECEIVE MAILBOX ACKNOWLEDGE

  `#define HCF_RECV_MBX_CMD 0x0020`

  Both the *Receive Mailbox Acknowledge* flag and the *Receive Mailbox Command* flag are used together to transfer non-cyclic packages between the protocol stack and the application. At initialization time, this flag is cleared.

- PROCESS DATA OUT COMMAND

  `#define HCF_PD0_OUT_CMD 0x0040`

  (not supported yet: `#define HCF_PD1_OUT_CMD 0x0100`)

  Both the *PDx OUT Command* flag and the *PDx OUT Acknowledge* flag are used together to transfer cyclic data between the application and the protocol stack. ('x' indicates the number of the output/input area.) At initialization time, this flag may be set, depending on the data exchanged mode.

- PROCESS DATA IN ACKNOWLEDGE

  ```
  #define HCF_PD0_IN_ACK                0x0080
  #define HCF_PD1_IN_ACK                0x0200
  ```
  (not supported yet:)

  Both the *PDx IN Acknowledge* flag and the *PDx IN Command* flag are used together to transfer cyclic data between the protocol stack and the application. ('x' indicates the number of the output/input area.) At initialization time, this flag may be set, depending on the data exchanged mode.

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

Detail on the process data handshake mechanism can be found on page 78.

3.2.3 Handshake Block

If required, the handshake register can be moved from the handshake channel to the beginning of the communication channel area. This handshake block is not yet supported and therefore set to zero. It is always available in the default memory map (see page 47). Locating the handshake block in the channel section of the dual-port memory is not supported yet.

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

Table 19 - Communication Handshake Block

Communication Handshake Block Structure Reference

```
typedef struct NETX_HANDSHAKE_BLOCK_Ttag
{
  UINT8    abReserved[8];
} NETX_HANDSHAKE_BLOCK_T;
```

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 67 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 52)
0x000C	UINT32	`ulDeviceWatchdog`	Device Watchdog Host System Writes, Protocol Stack Reads (see page 53)

Table 20 - Communication Control Block

Communication Control Block Structure Reference

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

Application Change of State

Using this state field the application can send commands to the communication channel.

`ulApplicationCOS` - Host writes, netX reads

31	30	…	12	11	10	9	8	7	6	5	4	3	2	1	0
															RCX_APP_COS_APP_READY
														RCX_APP_COS_BUS_ON	
													RCX_APP_COS_BUS_ON_ENABLE		
												RCX_APP_COS_INIT			
											RCX_APP_COS_INIT_ENABLE				
										RCX_APP_COS_LOCK_CONFIG					
									RCX_APP_COS_LOCK_CONFIG_ENABLE						
unused, set to zero															

Table 21 - Application Change of State

Application Change of State Flags (Application ⇨ netX System)

- APPLICATION READY `#define RCX_APP_COS_APP_READY 0x00000001`
  If set, the host application indicates to the protocol stack that its state is *Ready*. Not supported yet.
- BUS ON `#define RCX_APP_COS_BUS_ON 0x00000002`
  Using the *Bus On* flag, the host application allows or disallows the firmware to open network connections. This flag is used together *Bus On Enable* flag below. If set, the netX firmware tries to open network connections; if cleared, no connections are allowed and open connections are closed.
- BUS ON ENABLE `#define RCX_APP_COS_BUS_ON_ENABLE 0x00000004`
  The *Bus On Enable* flag is used together with the *Bus On* flag above. If set, this flag enables the execution of the Bus On command in the netX firmware (for details on the *Enable* mechanism see page 23).
- INITIALIZATION `#define RCX_APP_COS_INIT 0x00000008`
  Setting the *Initialization* 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.
- INITIALIZATION ENABLE `#define RCX_APP_COS_INIT_ENABLE 0x00000010`
  The *Initialization Enable* flag is used together with the *Initialization* flag above. If set, this flag enables the execution of the Initialization command in the netX firmware (for details on the *Enable* mechanism see page 23).
- LOCK CONFIGURATION `#define RCX_APP_COS_LOCK_CONFIG 0x00000020`
  If set, the host system does not allow the firmware to reconfigure the communication channel. The database will be locked. The *Configuration Locked* flag in the channel status block (see page 54) shows if the current database has been locked.
- LOCK CONFIGURATION ENABLE `#define RCX_APP_COS_LOCK_CONFIG_ENABLE 0x00000040`
  The *Lock Configuration Enable* flag is used together with the *Lock Configuration* flag above. If set, this flag enables the execution of the Lock Configuration command in the netX firmware (for details on the *Enable* mechanism see page 23).
- Others are reserved.

Changing flags in the above register requires the application also to toggle the *Host Change of State Command* flag in the *Host Communication Flags* register (see page 50). Only then, the netX protocol stack recognizes the change.

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 54) to the device watchdog location (page 52), 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 153.

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 67 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 55)
0x0014	UINT32	`ulCommunicationState`	Communication State OFFLINE, STOP, IDLE, OPERATE (see page 56)
0x0018	UINT32	`ulCommunicationError`	Communication Error Unique Error Number According to Protocol Stack (see page 56)
0x001C	UINT16	`usVersion`	Version Version Number of this Diagnosis Structure (see page 56)
0x001E	UINT16	`usWatchdogTime`	Watchdog Time Configured Watchdog Time (see page 56)
0x0020	UINT16	`ausReserved[2]`	Reserved Former version Protocol Class, set to 0
0x0024	UINT32	`ulHostWatchdog`	Host Watchdog Joint Supervision Mechanism Protocol Stack Writes, Host System Reads (see page 57)
0x0028	UINT32	`ulErrorCount`	Error Count Total Number of Detected Error Since Power-Up or Reset (see page 57)
0x002C	UINT32	`ulErrorLogInd`	Error Log Indicator Total Number Of Entries In The Error Log Structure (not supported yet) (see page 57)
0x0030	UINT32	`aulReserved[2]`	Reserved Set to 0

Table 22 - Common Status Block

Common Status Block Structure Reference

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

Communication Change of State (All Implementations)

The communication change of state register contains information about the current operating status of the communication channel and its firmware. Every time the status changes, the netX protocol stack toggles the n*etX Change of State Command* flag in the netX communication flags register (see page 48). The application then has to toggle the *netX Change of State Acknowledge* flag back acknowledging the new state.

`ulCommunicationCOS` - netX writes, Host reads

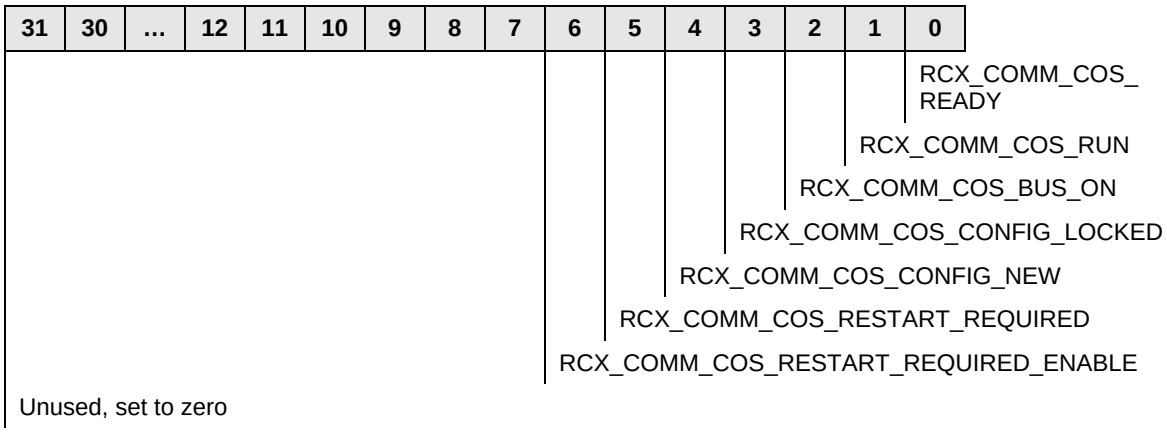


Table 23 - Communication State of Change

Communication Change of State Flags (netX System ⇨ Application)

- READY `#define RCX_COMM_COS_READY 0x00000001`
  The *Ready* flag is set as soon as the protocol stack is started properly. Then the protocol stack is awaiting a configuration. As soon as the protocol stack is configured properly, the *Running* flag is set, too.
- RUNNING `#define RCX_COMM_COS_RUN 0x00000002`
  The *Running* flag is set when the protocol stack has been configured properly. Then the protocol stack is awaiting a network connection. Now both the *Ready* flag and the *Running* flag are set.

- BUS ON `#define RCX_COMM_COS_BUS_ON 0x00000004`
  The *Bus On* flag is set to indicate to the host system whether or not the protocol stack has the permission to open network connections. If set, the protocol stack has the permission to communicate on the network; if cleared, the permission was denied and the protocol stack will not open network connections.
- CONFIGURATION LOCKED `#define RCX_COMM_COS_CONFIG_LOCKED 0x00000008`
  The *Configuration Locked* flag is set, if the communication channel firmware has locked the configuration database against being overwritten. Re-initializing the channel is not allowed in this state. To unlock the database, the application has to clear the *Lock Configuration* flag in the control block (see page 52).
- CONFIGURATION NEW `#define RCX_COMM_COS_CONFIG_NEW 0x00000010`
  The *Configuration New* flag is set by the protocol stack to indicate that a new configuration became available, which has not been activated. This flag may be set together with the *Restart Required* flag.
- RESTART REQUIRED
  `#define RCX_COMM_COS_RESTART_REQUIRED 0x00000020`
  The *Restart Required* flag is set when the channel firmware requests to be restarted. This flag is used together with the *Restart Required Enable* flag below. Restarting the channel firmware may become necessary, if a new configuration was downloaded from the host application or if a configuration upload via the network took place.
- RESTART REQUIRED ENABLE
  `#define RCX_COMM_COS_RESTART_REQUIRED_ENABLE 0x00000040`
  The *Restart Required Enable* flag is used together with the *Restart Required* flag above. If set, this flag enables the execution of the Restart Required command in the netX firmware (for details on the *Enable* mechanism see 23).
- Other are reserved.

Communication State (All Implementations)

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

- UNKNOWN `#define RCX_COMM_STATE_UNKNOWN 0x00000000`
- OFFLINE `#define RCX_COMM_STATE_OFFLINE 0x00000001`
- STOP `#define RCX_COMM_STATE_STOP 0x00000002`
- IDLE `#define RCX_COMM_STATE_IDLE 0x00000003`
- OPERATE `#define RCX_COMM_STATE_OPERATE 0x00000004`

Communication Channel Error (All Implementations)

This field holds the current error code of the communication channel. If the cause of error is resolved, the communication error field is set to zero (= `RCX_S_OK`) again. Not all of the error codes are supported in every implementation. Protocol stacks may use a subset of the error codes outlined in section 7 on page 203.

Version (All Implementations)

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

- STRUCTURE VERSION `#define RCX_STATUS_BLOCK_VERSION 0x0001`

Watchdog Timeout (All Implementations)

This field holds the configured watchdog timeout value in milliseconds. The application may set its watchdog trigger interval accordingly. If the application fails to copy the value from the host watchdog location to the device watchdog location, the protocol stack will interrupt all network connections immediately regardless of their current state. For details, see page 153.

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 52), 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 54.

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.

3.2.5.2 Master Implementation

In addition to the common status block as outlined on page 54, a master firmware maintains the following structure.

Master Status			
Start Offset	**Type**	**Name**	**Description**
0x0010	Structure	See common structure in Table 22	
0x0038	UINT32	`ulSlaveState`	Slave State OK, FAILED (At Least One Slave) (see page 59)
0x003C	UINT32	`ulSlaveErrLogInd`	Slave Error Log Indicator Slave Diagnosis Data Available: EMPTY, AVAILABLE (see page 59)
0x0040	UINT32	`ulNumOfConfigSlaves`	Configured Slaves Number of Configured Slaves On The Network (see page 59)
0x0044	UINT32	`ulNumOfActiveSlaves`	Active Slaves Number of Slaves Running Without Problems (see page 59)
0x0048	UINT32	`ulNumOfDiagSlaves`	Faulted Slaves Number of Slaves Reporting Diagnostic Issues (see page 59)
0x004C	UINT32	`ulReserved`	Reserved Set to 0

Table 24 - Master Status

Master Status Structure Reference

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

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.

■ UNDEFINED	`#define RCX_SLAVE_STATE_UNDEFINED`	`0x00000000`
■ OK	`#define RCX_SLAVE_STATE_OK`	`0x00000001`
■ FAILED (at least one slave)	`#define RCX_SLAVE_STATE_FAILED`	`0x00000002`
■ WARNING (at least one slave)	`#define RCX_SLAVE_STATE_WARNING`	`0x00000003`
■ Others are reserved		

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 (see 6.1). This field holds the number of configured slaves.

Number of Active Slaves

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

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

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 54 only.

3.2.6 Extended Status Block (Protocol Specific)

The content of the channel specific status block is specific to the implementation and is defined in a separate manual. Depending on the protocol, a status area may or may not be used. It is always available in the default memory map (see page 47). This block can also be read using the mailbox interface (see page 67 for details).

Extended Status Block (Channel Specific)			
Offset	**Type**	**Name**	**Description**
0x0050	UINT8	`abExtendedStatus[432]`	Extended Status Area Protocol Stack Specific Status Area

Table 25 - Extended Status Block

Extended Status Block Structure Reference

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

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 66 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 66 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 **To** The Network or **To** the Protocol Stack
0x0840	UINT16	usWaitingPackages	Packages Waiting Counter of Packages that are Waiting to be Processed
0x0842	UINT16	usReserved	Reserved Set to 0
0x0844	UINT8	abRecvMbx[1596]	Receive Mailbox Non Cyclic Data **From** the Network or **From** the Protocol Stack

Table 26 - Channel Mailboxes

Channel Mailboxes Structure Reference

```
typedef struct NETX_SEND_MAILBOX_BLOCK_Ttag
{
  UINT16   usPackagesAccepted;
  UINT16   usReserved;
  UINT8    abSendMbx[1596];
} NETX_SEND_MAILBOX_BLOCK_T;

typedef struct NETX_RECV_MAILBOX_BLOCK_Ttag
{
  UINT16   usWaitingPackages;
  UINT16   usReserved;
  UINT8    abRecvMbx[1596];
} NETX_RECV_MAILBOX_BLOCK_T;
```

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 47). This block can also be read using the mailbox interface (see page 67 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 **To** The Network
0x0EC0	UINT8	`abPd1Input[64]`	High Priority Input Data Image High Priority Cyclic Data **From** The Network

Table 27 - 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 48). 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 47).

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

Table 28 - 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 47).

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

Table 29 - 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 48). In this case the input data should not be evaluated.

This block can also be read using the mailbox interface (see page 67 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 47).

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

Handshake Channel			
DPM Address	**Assigned Block**	**netX Register**	**Host Register**
0x0200	System Channel, set to 0	0x0200, 0x0201	
	System Channel	0x0202	0x0203
	Handshake Channel	0x0204	0x0206
	Communication Channel 0	0x0208	0x020A
	Communication Channel 1	0x020C	0x020E
	Communication Channel 2	0x0210	0x0212
	Communication Channel 3	0x0214	0x0216
	Application Channel 0	0x0218	0x021A
	Application Channel 1	0x021C	0x021E
	Future Registers, set to 0 (8 x UINT16)	0x0220	0x023F
… 0x02FF	Reserved, set to 0	0x0240 ... 0x02FF	

Table 30 - Handshake Channel

For compatibility reasons, the cells for the handshake block itself (offset 0x0204 and 0x0206) are present but not used and set to zero. Channel 6 and 7 are not supported yet and set to zero.

Handshake Channel Structure Reference

```
typedef union NETX_HANDSHAKE_REG_Ttag
{
  struct
  {
    UINT8   abData[2];
    UINT8   bNetxFlags;               /* netX writes                    */
    UINT8   bHostFlags;               /* host writes                    */
  } t8Bit;
  struct
  {
    UINT16  usNetxFlags;              /* netX writes                    */
    UINT16  usHostFlags;              /* host writes                    */
  } t16Bit;
  UINT32    ulReg;
} NETX_HANDSHAKE_REG_T;

typedef struct NETX_HANDSHAKE_CHANNEL_Ttag
{
  NETX_HANDSHAKE_REG_T       tSysFlags;   /* system handshake flags      */
  NETX_HANDSHAKE_REG_T       tHskFlags;   /* not used                    */
  NETX_HANDSHAKE_REG_T       tCommFlags0; /* channel 0 handshake flags   */
  NETX_HANDSHAKE_REG_T       tCommFlags1; /* channel 1 handshake flags   */
  NETX_HANDSHAKE_REG_T       tCommFlags2; /* channel 2 handshake flags   */
  NETX_HANDSHAKE_REG_T       tCommFlags3; /* channel 3 handshake flags   */
  NETX_HANDSHAKE_REG_T       tAppFlags0;  /* not supported yet           */
  NETX_HANDSHAKE_REG_T       tAppFlags1;  /* not supported yet           */
  NETX_HANDSHAKE_REG_T       tFutureRegs[8];  /* set to 0                */
  UINT16                     ulReserved[96];
} NETX_HANDSHAKE_CHANNEL_T;
```

- LENGTH OF HANDSHAKE BLOCK IN BYTES

```
#define NETX_HANDSHAKE_CHANNEL_SIZE          256
```

- NUMBER OF POSSIBLE HANDSHAKE PAIRS

```
#define NETX_HANDSHAKE_PAIRS                  16
```

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 each 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 itself 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 46 respectively on page 60.

NOTE Each mailbox can hold one packet at a time. The netX firmware stores packets that are not retrieved by the host application in a packet queue. This queue has limited space and may fill up so new packets maybe lost. To avoid these 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
	ulSta	UINT32		Status / Error Code
	ulCmd	UINT32		Command / Response
	ulExt	UINT32		Reserved
	ulRout	UINT32		Routing Information
tData	Structure Information			
	…	…		User Data Specific To The Command

Table 31 - 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 203.

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_HEADER_Ttag
{
  UINT32  ulDest;      /* Destination Queue Handler */
  UINT32  ulSrc;       /* Source Queue Handler      */
  UINT32  ulDestId;    /* Destination Identifier    */
  UINT32  ulSrcId;     /* Source Identifier         */
  UINT32  ulLen;       /* Length of Data Field      */
  UINT32  ulId;        /* Packet Identifier         */
  UINT32  ulSta;       /* Status / Error Code       */
  UINT32  ulCmd;       /* Command / Response        */
  UINT32  ulExt;       /* Extension Field           */
  UINT32  ulRout;      /* Routing Information       */
} RCX_PACKET_HEADER_T;
```

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.

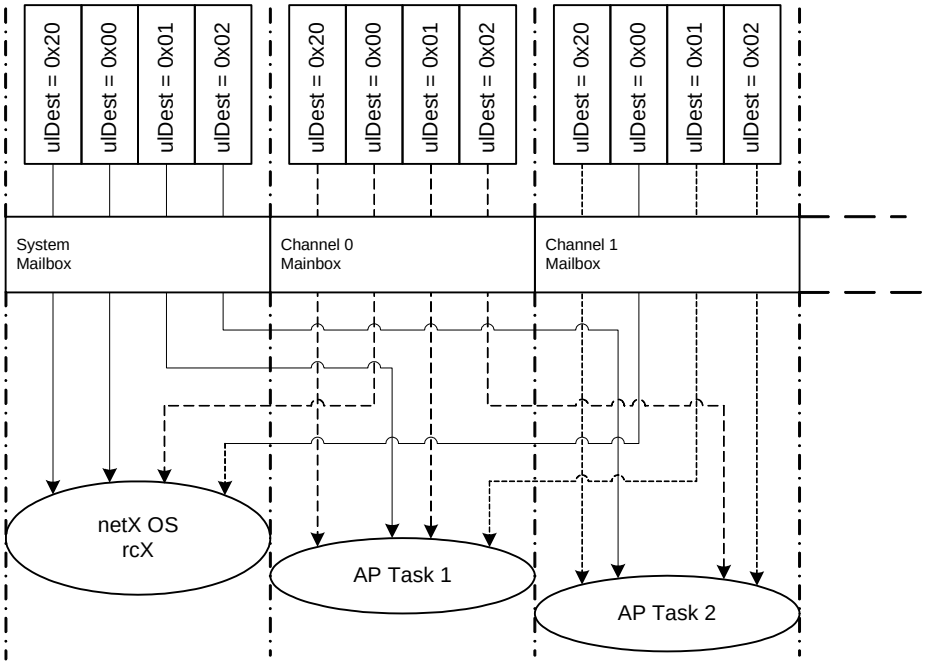


Figure 5 - Use of ulDest *in Channel and System Mailbox*

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 32 - 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 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:

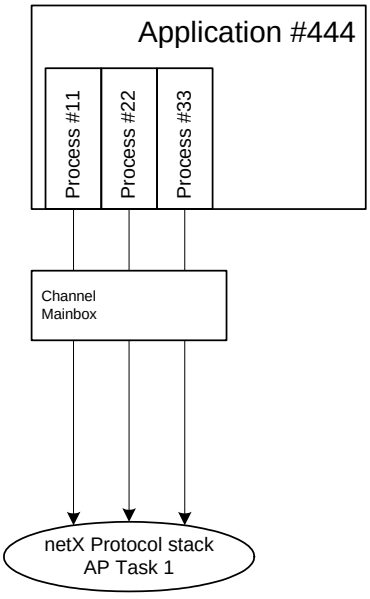


Figure 6 - Using ulSrc *and* ulSrcId

Example:

This example applies to command messages imitated 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).

4.1.4 How to Route rcX Packets

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

4.1.5 Client/Server Mechanism

4.1.5.1 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).

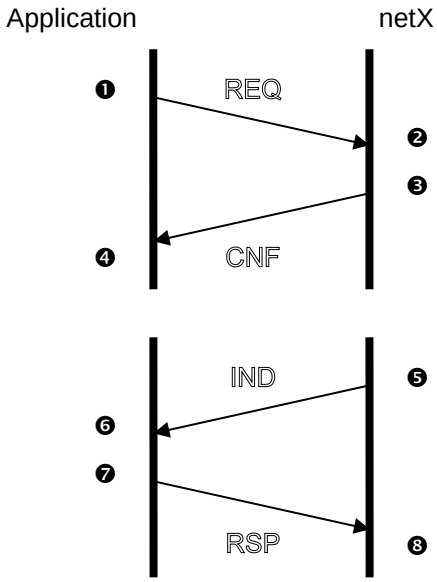


Figure 7 - Transition Chart Application as Client

❶ ❷ The host application sends request packets to the netX firmware.

❸ ❹ The netX firmware sends a confirmation packet in return.

❺ ❻ The host application receives indication packets from the netX firmware.

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

REQ	Request	CNF	Confirmation
IND	Indication	RSP	Response

4.1.5.2 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).

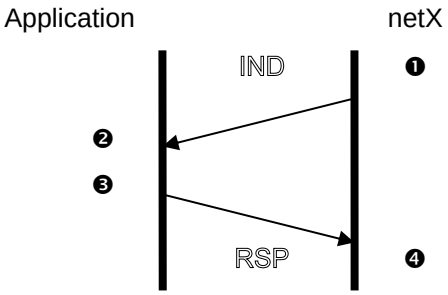


Figure 8 - Transition Chart Application as Server

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

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

IND Indication RSP Response

4.1.6 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.

4.1.6.1 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.

- NO SEQUENCED PACKET `#define RCX_PACKET_SEQ_NONE 0x00000000`
- FIRST PACKET OF SEQUENCE `#define RCX_PACKET_SEQ_FIRST 0x00000080`
- SEQUENCED PACKET `#define RCX_PACKET_SEQ_MIDDLE 0x000000C0`
- LAST PACKET OF SEQUENCE `#define RCX_PACKET_SEQ_LAST 0x00000040`

The other important field is the identifier field *ulId*. 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.

4.1.6.2 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 33 - 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 34 - Upload Request (CMD = upload command; N = None; F = First; M = Middle; L = Last)

4.1.6.3 Abort Fragmented Packets Request

A data block transfer should be aborted when a sequence number in the identifier field *ulId* is skipped or used twice. Failure in handling the extension flags in *ulExt* 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: *ulCmd* = CMD, CMD is fieldbus dependent) at any time during the process. Additionally, the length field *ulLen* is set to zero and the extension field *ulExt* 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				
Type: Request				
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	0	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	0xC02B0024 0xC0000010	Status Packet out of Sequence Out Of Memory
	ulCmd	UINT32	CMD	Command
	ulExt	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`

4.1.6.4 Abort Fragmented Packet Confirmation

The receiver of the abort request returns the following packet.

Structure Information				
Type: Confirmation				
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

The offset address in the IO data image of each input and output point or device is set by the SYCON.net network configuration tool or can be configured online via an application program.

Each of the inputs or outputs in the IO block can have additional information regarding its status, indicating whether its data is valid or not. This status information field (4 bytes) is optional and can be configured by SYCON.net or online via a network specific description table (see page 200 for details).

Depending on the implementation, an output and input data image may or may not be present.

4.2.1 Process Data Transfer Synchronization

The dual-port memory features buffered data transfer mode to ensure data consistency over the entire process data image individually for each input and output image. Therefore, the protocol stack maintains buffers internally that hold a copy of the process data image for each direction. Data to be sent to the network is taken from those buffers; data to be received from the network is stored in those buffers.

In the "controlled" mode, the protocol stack synchronizes the exchange of data between these buffers and the process data image in the dual-port memory with the application via a handshake mechanism. Once copied from/into the input/output area, the protocol stack gives control over the dual-port memory to the application. When the application has finished copying, the control is given back to the protocol stack, and so on. If no update of data happened, the protocol stack overwrites the input buffer with data received from the network. If the application is much faster than the network cycle, it might be possible that data in the output buffers is overwritten without ever being sent to the network.

The above handshake mechanism applies to input and output data areas that have the *IN* respectively *OUT flag* set in their *direction* field; blocks with the *direction* field indicating *IN - OUT* (bi-directional) use always the *Uncontrolled, Not Buffered* mode (see below).

4.2.2 Process Data Handshake Modes

The process data handshake is carried out individually for each input and output image, respectively for each protocol stack. The protocol stack allows controlling the transfer of data independently for inputs and outputs. The handshake cells are located in the handshake channel (see pages 64 and 48 for details). The internal buffers have the same size like the input and output data images in the dual-port memory for the assigned memory block. The following data exchange modes are supported.

Mode	Controlled by	Consistency	Supported by
Not Buffered	No Control	None	Master & Slave FW
Buffered	Host (Application/Driver)	Yes	Master & Slave FW

Table 35 - Process Data Handshake Modes

The configuration of the protocol stacks in terms of their process data handshake mode is carried out by SYCON.net (see page 200 for details).

The following sections are intended to illustrate the procedure of transferring data from the host application to the netX dual-port memory and vice versa. The step-by-step section shows the mechanism in a block diagram and details the data flow. The second approach provides a different view of the procedure and adds a time component to the mechanism. Input and output data is transferred independently and therefore use an own pair of handshake bits.

4.2.2.1 Not Buffered, Uncontrolled Mode

For each valid bus cycle the protocol stack updates the process data in the input and output data image in the dual-port memory. No handshake bits are evaluated and no buffers are used. The application can read or write process data at any given time without obeying any synchronization mechanism otherwise carried out via handshake location. This handshake mode is the simplest method of transferring process data between the protocol stack and the application.

NOTE This mode can only guarantee data consistency over a byte.

Step-by-Step Procedure

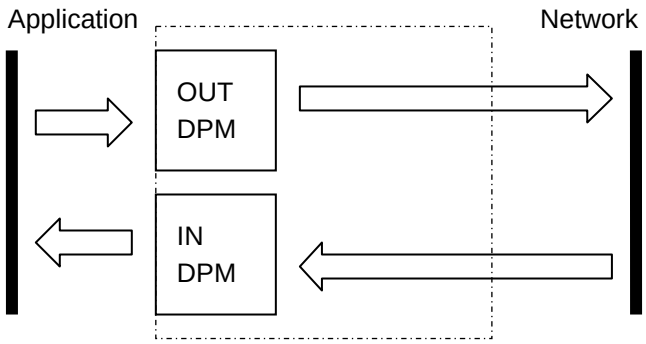


Figure 9 - Step-by-Step: Not Buffered, Uncontrolled Mode

The protocol stack receives network data directly into the input data image and sends data to the network from the output data image of the dual-port memory. There is no handshaking necessary and therefore no guaranty for constancy of input or output data.

Time Related View

- Output/Input Data Exchange

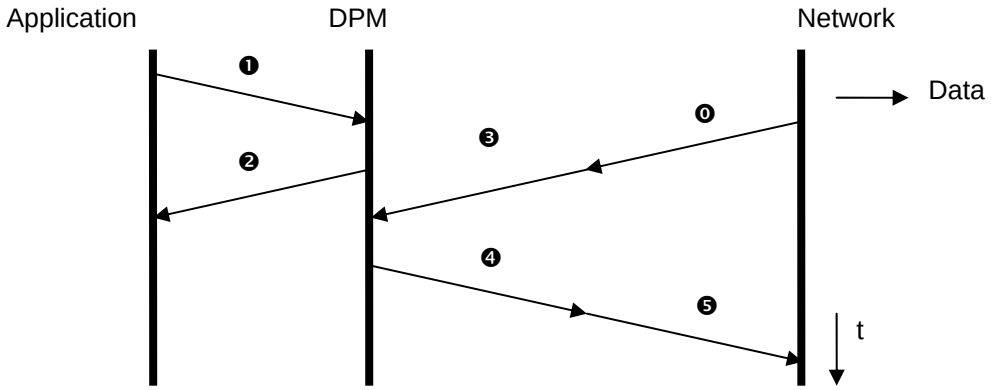


Figure 10 - Time Related: Not Buffered, Uncontrolled Mode

❻❺ The protocol stack constantly sends and receives data to/from the network.

❶ The application may copy data into output data image of the dual-port memory at any time.

❷ The application copies data from the input data area of the dual-port memory at any time.

❸ As soon as a new telegram is available from the network, the protocol stack copies it's data directly into the input data image of the dual-port memory.

❹ When a new telegram has to be sent to the network, the protocol stack takes its data directly from the output data image of the dual-port memory.

❺❻ The protocol stack sends data from the output data image. Once updated, the protocol stack uses the new data from the buffer to send it to the network. The cycle starts over with step 1.

NOTE In case of a network fault (e.g. disconnected network cable), a netX 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 48); the input data should not be evaluated anymore.

4.2.2.2 Buffered, Controlled Mode

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

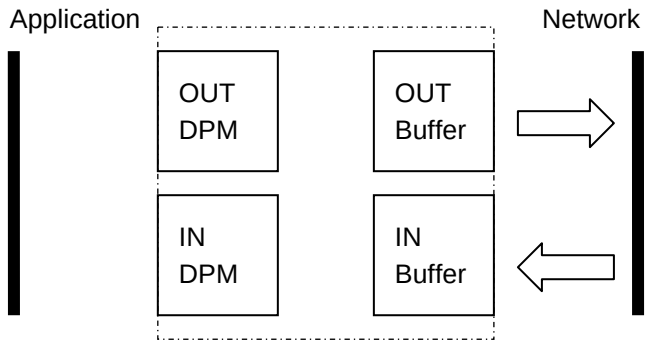


Figure 11 - Step 1: Buffered, Controlled Mode

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.

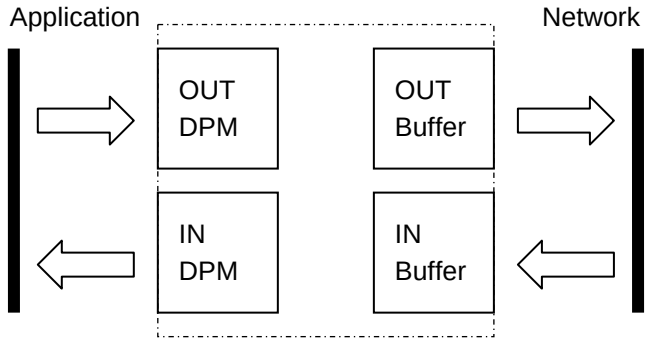


Figure 12 - Step 2: Buffered, Controlled Mode

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.

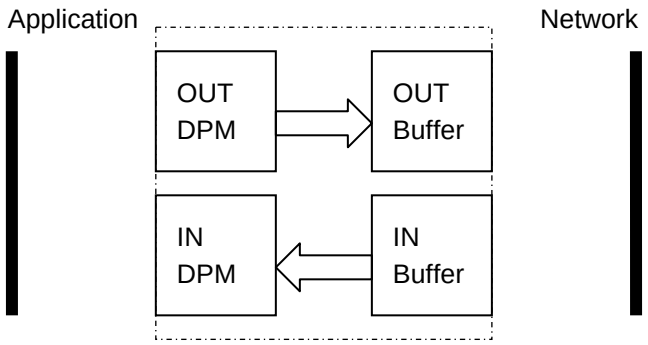


Figure 13 - Step 3: Buffered, Controlled Mode

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.

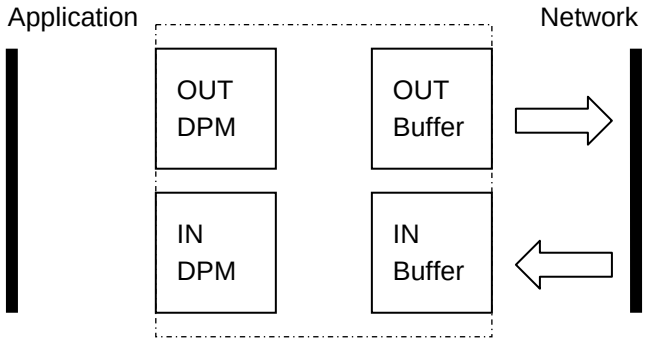


Figure 14 - Step 4: Buffered, Controlled Mode

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

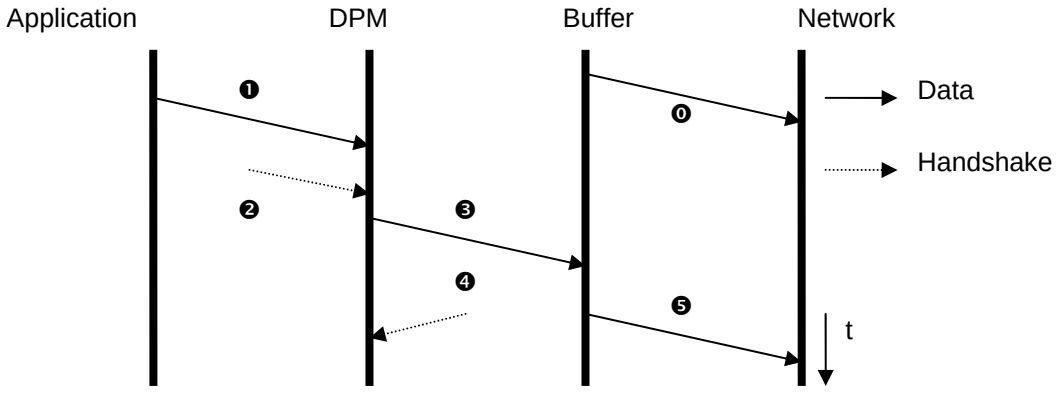


Figure 15 - Time Related: Buffered, Controlled, Output Data

❶ The protocol stack constantly transmits data from the buffer to the network.

❶ The application has control over the dual-port memory and can copy data to the output data image.

❷ The application then toggles the handshake bits, giving control over the dual-port memory to the protocol stack.

❸ The protocol stack copies the content of the output data image into the internal OUT buffer.

❹ The protocol stack toggles the handshake bits, giving control back to the application.

❺ 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

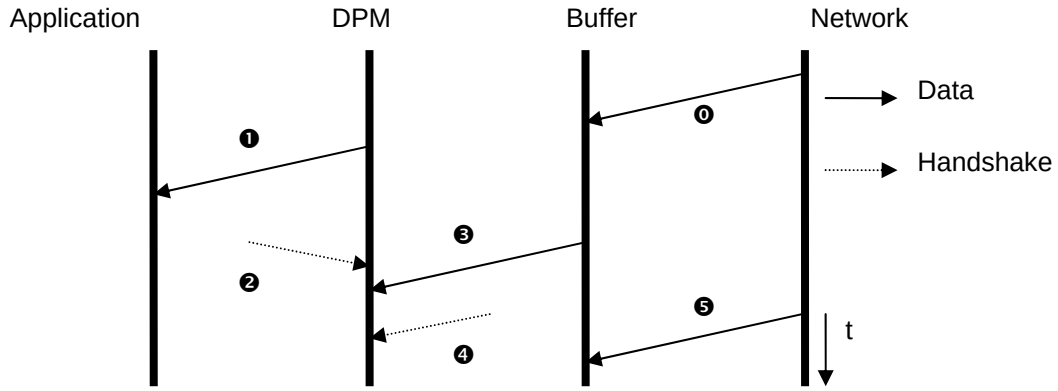


Figure 16 - Time Related: Buffered, Controlled, Input Data

❻❺ The protocol stack constantly receives data from the network into the buffer.

❶ 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.

❷ The application then toggles the handshake bits, giving control over the dual-port memory to the netX protocol stack.

❸ The protocol stack copies the latest content of the internal IN buffer to the input data image of the dual-port memory.

❹ The protocol stack then toggles the handshake bits, giving control back to the application.

❺❻ 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 48); 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*.

The provider status indicates whether the data is valid (Good, Bad).

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.

The following common status flags are defined:

- DATA STATE (Good, Bad) `#define RCX_IODS_DATA_STATE_GOOD 0x0080`
- PROVIDER STATE (Run, Stop) `#define RCX_IODS_PROVIDER_RUN 0x0040`
- GENERATED (Locally, Remote) `#define RCX_IODS_GENERATED_LOCALLY 0x0020`
- FIELDBUS MASK `#define RCX_IODS_FIELDBUS_MASK 0x00F0`
- Others are reserved.

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.

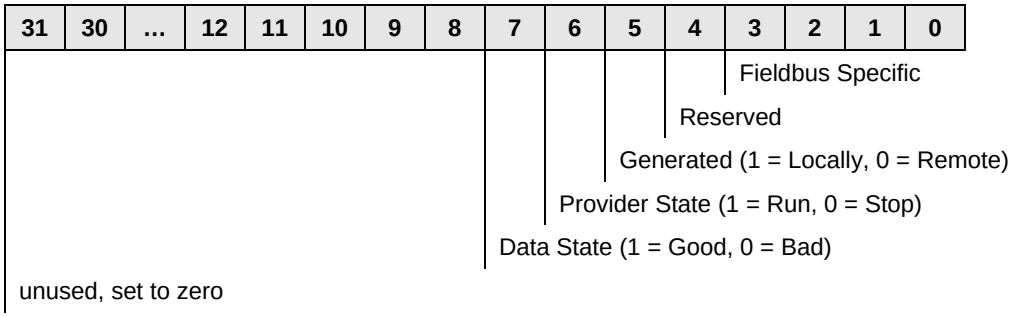


Table 36 - Input Data Status

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.

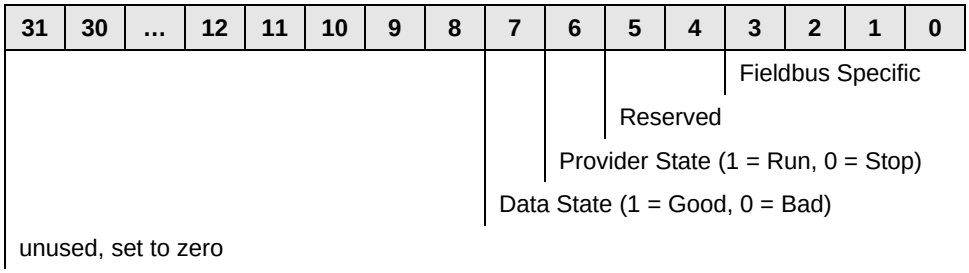


Table 37 - Output Data Status

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 52). 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 52). 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 48).

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 52). 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 48.

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 52.

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				
Type: Request				
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

- DESTINATION: CHANNEL `#define RCX_PKT_COMM_CHANNEL_TOKEN 0x00000020`
- START / STOP COMMUNICATION REQUEST
  `#define RCX_START_STOP_COMM_REQ 0x00002F30`

Packet Structure Reference

```
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_T              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				
Type: Confirmation				
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 7
	ulCmd	UINT32	0x00002F31	Confirmation Start / Stop Communication
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

- START / STOP COMMUNICATION CONFIRMATION

```
#define RCX_START_STOP_COMM_CNF          RCX_START_STOP_COMM_REQ+1
```

Packet Structure Reference

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

Data Field

There is no data field returned in the start / stop confirmation packet.

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 explicity mentioned in the documentation of the protocol stack.

4.5.1 Lock Configuration through Dual-Port Memory

If the host application whishes to lock the configuration settings, it sets the *Lock Configuration* flag in the control block (see page 52). As a result, the channel firmware sets the *Configuration Locked* flag in the status block (see page 54), 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 52). 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				
Type: Request				
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

- DESTINATION: CHANNEL `#define RCX_PKT_COMM_CHANNEL_TOKEN 0x00000020`
- LOCK / UNLOCK CONFIGURATION REQUEST
  `#define RCX_LOCK_UNLOCK_CONFIG_REQ 0x00002F32`

Packet Structure Reference

```
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_T                 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				
Type: Confirmation				
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 7
	ulCmd	UINT32	0x00002F33	Confirmation Lock/Unlock Configuration
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

- LOCK / UNLOCK CONFIGURATION CONFIRMATION

```
#define RCX_LOCK_UNLOCK_CONFIG_CNF     RCX_LOCK_UNLOCK_CONFIG_REQ+1
```

Packet Structure Reference

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

Data Field

There is no data field returned in the lock / unlock confirmation packet.

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 28). 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.

The layout of the dual-port memory may change when the configurations 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 44, indicating that the default memory layout is used (see page 47). 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 35). 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**	
		Data Type	**Description**
0x0060	**Communication Channel 1**	UINT8	Channel Type = COMMUNICATION (see page 37)
		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…)
… 0x006F		UINT8[2]	Reserved

Table 38 - 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 35, 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 35) 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				
Type: Request				
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)

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- GET BLOCK INFORMATION REQUEST
  `#define RCX_DPM_GET_BLOCK_INFO_REQ 0x00001EF8`

Packet Structure Reference

```
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_T
{
  RCX_PACKET_HEADER_T               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 35.

4.6.3.4 Get Block Information Confirmation

The firmware replies with the following message.

Structure Information				
Type: Confirmation				
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	28 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 7
	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

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- GET BLOCK INFORMATION CONFIRMATION
  `#define RCX_DPM_GET_BLOCK_INFO_CNF RCX_DPM_GET_BLOCK_INFO_REQ+1`

Packet Structure Reference

```
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 within the area */
  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  tData;   /* packet data             */
} RCX_DPM_GET_BLOCK_INFO_CNF_T;
```

Sub Block 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 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.

■ UNDEFINED	`#define RCX_BLOCK_UNDEFINED`	`0x0000`
■ UNKNOWN	`#define RCX_BLOCK_UNKNOWN`	`0x0001`
■ PROCESS DATA IMAGE	`#define RCX_BLOCK_DATA_IMAGE`	`0x0002`
■ HIGH PRIORITY DATA IMAGE	`#define RCX_BLOCK_DATA_IMAGE_HI_PRIO`	`0x0003`
■ MAILBOX	`#define RCX_BLOCK_MAILBOX`	`0x0004`
■ CONTROL	`#define RCX_BLOCK_CNTRL_PARAM`	`0x0005`
■ COMMON STATUS	`#define RCX_BLOCK_COMMON_STATE`	`0x0006`
■ EXTENDED STATUS	`#define RCX_BLOCK_EXTENDED_STATE`	`0x0007`
■ USER	`#define RCX_BLOCK_USER`	`0x0008`
■ RESERVED	`#define RCX_BLOCK_RESERVED`	`0x0009`
■ Others are reserved		`0x000A … 0xFFFF`

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.

Flags

The flags field holds information regarding the data transfer direction from the view point of the application. The following flags are defined.

- DIRECTION MASK `#define RCX_DIRECTION_MASK 0x000F`
- UNDEFINED `#define RCX_DIRECTION_UNDEFINED 0x0000`
- IN (netX to Host System) `#define RCX_DIRECTION_IN 0x0001`
- OUT (Host System to netX) `#define RCX_DIRECTION_OUT 0x0002`
- IN - OUT (Bi-Directional) `#define RCX_DIRECTION_IN_OUT 0x0003`
- Others are reserved

The transmission type field in the flags location holds the type of how to exchange data with this sub block. The choices are:

- TRANSMISSION MASK `#define RCX_TRANSMISSION_TYPE_MASK 0x00F0`
- UNDEFINED `#define RCX_TRANSMISSION_TYPE_UNDEFINED 0x0000`
- DPM (Dual-Port Memory) `#define RCX_TRANSMISSION_TYPE_DPM 0x0010`
- DMA (Direct Memory Access) `#define RCX_TRANSMISSION_TYPE_DMA 0x0020`
- Others are reserved

Handshake Mode

The handshake mode is defined only for IO data images. The handshake modes are the same as defined on page 78.

- IO MODE MASK `#define RCX_IO_MODE_MASK 0x000F`
- UNKNOWN `#define RCX_IO_MODE_UNKNOWN 0x0000`
- UNCONTROLLED `#define RCX_IO_MODE_UNCONTROLLED 0x0003`
- BUFFERED, HOST CONTROLLED `#define RCX_IO_MODE_BUFF_HST_CTRL 0x0004`
- Others are reserved `0x0001, 0x0002, 0x0005 … 0xFFFF`

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 64 and 48 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 writeable 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 writeable.

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

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 writeable.

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 105 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				
Type: Request				
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

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- READ SECURITY EEPROM REQUEST
`#define RCX_SECURITY_EEPROM_READ_REQ 0x00001EBC`
- ZONE 1 `#define RCX_SECURITY_EEPROM_ZONE_1 0x00000001`
- ZONE 2 `#define RCX_SECURITY_EEPROM_ZONE_2 0x00000002`
- ZONE 3 `#define RCX_SECURITY_EEPROM_ZONE_3 0x00000003`

Packet Structure Reference

```
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_T                  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				
Type: Confirmation				
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	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 7
	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**

- READ SECURITY EEPROM CONFIRMATION
  `#define RCX_SECURITY_EEPROM_READ_CNF RCX_SECURITY_EEPROM_READ_REQ+1`

Packet Structure Reference

```
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_T                 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				
Type: Request				
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

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- WRITE SECURITY EEPROM REQUEST
  `#define RCX_SECURITY_EEPROM_WRITE_REQ 0x00001EBE`
- ZONE 1 `#define RCX_SECURITY_EEPROM_ZONE_1 0x00000001`
- ZONE 2 `#define RCX_SECURITY_EEPROM_ZONE_2 0x00000002`
- ZONE 3 `#define RCX_SECURITY_EEPROM_ZONE_3 0x00000003`

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

Packet Structure Reference

```
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_T                  tHead;   /* packet header         */
  RCX_SECURITY_EEPROM_WRITE_REQ_DATA_T tData;   /* packet data           */
} RCX_SECURITY_EEPROM_WRITE_REQ_T;
```

4.7.1.4 Security Memory Write Confirmation

The netX operating system returns the following packet.

Structure Information				
Type: Confirmation				
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 7
	ulCmd	UINT32	0x00001EBF	Confirmation Write Security EEPROM
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

- WRITE SECURITY EEPROM CONFIRMATION
  `#define RCX_SECURITY_EEPROM_WRITE_CNF  RCX_SECURITY_EEPROM_WRITE_REQ+1`

Packet Structure Reference

```
typedef struct RCX_SECURITY_EEPROM_WRITE_CNF_Ttag
{
  RCX_PACKET_HEADER_T  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 39 - 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 40 - PCI System and OS Setting (Zone 2)

Zone 3 - User Specific Zone

Offset	Type	Name	Description
0-31	UINT8[32]	(*User Specific*)	Reserved, 32 Byte

Table 41 - User Specific Zone (Zone 3)

Memory Zones Structure Reference

```
typedef struct RCX_SECURITY_MEMORY_ZONE1_Ttag
{
  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_T;

typedef struct RCX_SECURITY_MEMORY_ZONE2_Ttag
{
  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_T;

typedef struct RCX_SECURITY_MEMORY_ZONE3_Ttag
{
  UINT8    UserSpecific[32];       /* user specific area                  */
} RCX_SECURITY_MEMORY_ZONE3_T;
```

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 28).

- **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				
Type: Request				
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

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- IDENTIFY FIRMWARE REQUEST
  `#define RCX_HW_IDENTIFY_REQ 0x00001EB8`

Packet Structure Reference

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

4.7.2.2 Identify Hardware Confirmation

The channel firmware returns the following packet.

Structure Information				
Type: Confirmation				
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	88 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 7
	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 Device Number / Identification (see page 29)
	ulSerial Number	UINT32	0 … 0xFFFFFFFF	Serial Number Serial Number (see page 29)
	ausHw Options[4]	UINT16	0 … 0xFFFF	Hardware Options Hardware Assembly Option (see page 29)
	usDeviceClass	UINT16	0 … 0xFFFF	Device Class netX Device Class (see page 33)
	bHwRevision	UINT8	0 … 0xFF	Hardware Revision Hardware Revision Index (see page 34)
	bHw Compatibility	UINT8	0 … 0xFF	Hardware Compatibility Hardware Compatibility Index (see page 34)
	ulBootType	UINT32	0 … 6	Hardware Boot Type See below

- HARDWARE IDENTIFY CONFIRMATION

```
#define RCX_HW_IDENTIFY_CNF          RCX_HW_IDENTIFY_REQ+1
```

Packet Structure Reference

```
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         */
} RCX_HW_IDENTIFY_CNF_DATA_T;

typedef struct RCX_HW_IDENTIFY_CNF_Ttag
{
  RCX_PACKET_HEADER_T         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.

Hardware Boot Type

This field indicates how the netX operating system was started. It is more or less for informational purposes only.

- PARALLEL FLASH (SRAM Bus) `#define RCX_BOOT_TYPE_PFLASH_SRAMBUS 0x00000000`
- PARALLEL FLASH (Extension Bus) `#define RCX_BOOT_TYPE_PFLASH_EXTBUS 0x00000001`
- DUAL-PORT MEMORY `#define RCX_BOOT_TYPE_DUALPORT 0x00000002`
- PCI INTERFACE `#define RCX_BOOT_TYPE_PCI 0x00000003`
- MULTIMEDIA CARD `#define RCX_BOOT_TYPE_MMC 0x00000004`
- I²C BUS `#define RCX_BOOT_TYPE_I2C 0x00000005`
- SERIAL FLASH `#define RCX_BOOT_TYPE_SFLASH 0x00000006`
- Others are reserved `0x00000007 through 0xFFFFFFFF`

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				
Type: Request				
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

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- OBTAIN LICENSE INFORMATION REQUEST
  `#define RCX_HW_LICENSE_INFO_REQ 0x00001EF4`

Packet Structure Reference

```
typedef struct RCX_HW_LICENSE_INFO_REQ_Ttag
{
  RCX_PACKET_HEADER_T   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				
Type: Confirmation				
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	12 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 7
	ulCmd	UINT32	0x00001EF5	Confirmation License Information
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use
tData	Structure Information			
	ulLicense Flags1	UINT32	0 … 0xFFFFFFFF	License Code License Flags 1
	ulLicense Flags2	UINT32	0 … 0xFFFFFFFF	License Code License Flags 2
	usNetx LicenseID	UINT16	0 … 0xFFFF	License Code netX License Identification
	usNetxLicense Flags	UINT16	0 … 0xFFFF	License Code netX License Flags

- OBTAIN LICENSE INFORMATION CONFIRMATION

```
#define RCX_HW_LICENSE_INFO_CNF          RCX_HW_LICENSE_INFO_REQ+1
```

Packet Structure Reference

```
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_CNF_Ttag
{
  RCX_PACKET_HEADER_T              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 32 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				
Type: Request				
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

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- READ HARDWARE INFORMATION REQUEST
  `#define RCX_HW_HARDWARE_INFO_REQ 0x00001EF6`

Packet Structure Reference

```
typedef struct RCX_HW_HARDWARE_INFO_REQ_Ttag
{
  RCX_PACKET_HEADER_T   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				
Type: Confirmation				
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	44 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 7
	ulCmd	UINT32	0x00001EF7	Confirmation 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 Device Number / Identification (see page 29)
	ulSerial Number	UINT32	0 … 0xFFFFFFFF	Serial Number Serial Number (see page 29)
	ausHw Options[4]	Array of UINT16	0 … 0xFFFF	Hardware Options Hardware Assembly Option (see page 29)
	usManu-facturer	UINT16	0 … 0xFFFF	Manufacturer Manufacturer Code / Manufacturer Location (see page 31)
	usProduction Date	UINT16	0 … 0xFFFF	Production Date Production Date (see page 31)
	ulLicense Flags1	UINT32	0 … 0xFFFFFFFF	License Code License Flags 1 (see page 32)
	ulLicense Flags2	UINT32	0 … 0xFFFFFFFF	License Code License Flags 2 (see page 32)
	usNetx LicenseID	UINT16	0 … 0xFFFF	License Code netX License Identification (see page 32)
	usNetxLicense Flags	UINT16	0 … 0xFFFF	License Code netX License Flags (see page 32)

continued next page

usDeviceClass	UINT16	0 … 0xFFFF	Device Class netX Device Class (see page 33)
bHwRevision	UINT8	0 …0xFFFF	Hardware Revision Hardware Revision Index (see page 34)
bHw Compatibility	UINT8	0	Hardware Compatibility Hardware Compatibility Index (see page 34)
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

- READ HARDWARE INFORMATION CONFIRMATION

```
#define RCX_HW_HARDWARE_INFO_CNF        RCX_HW_HARDWARE_INFO_REQ+1
```

Packet Structure Reference

```
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_T               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				
Type: Request				
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 *ulDest* = CHANNEL Communication Channel Firmware System Channel

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- DESTINATION: CHANNEL `#define RCX_PKT_COMM_CHANNEL_TOKEN 0x00000020`
- IDENTIFY FIRMWARE REQUEST `#define RCX_FIRMWARE_IDENTIFY_REQ 0x00001EB6`
- SYSTEM CHANNEL `#define RCX_SYSTEM_CHANNEL 0xFFFFFFFF`
- COMMUNICATION CHANNEL 0 `#define RCX_COMM_CHANNEL_0 0x00000000`
- COMMUNICATION CHANNEL 1 `#define RCX_COMM_CHANNEL_1 0x00000001`
- COMMUNICATION CHANNEL 2 `#define RCX_COMM_CHANNEL_2 0x00000002`
- COMMUNICATION CHANNEL 3 `#define RCX_COMM_CHANNEL_3 0x00000003`

Packet Structure Reference

```
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_T               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				
Type: Confirmation				
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	76 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 7
	ulCmd	UINT32	0x00001EB7	Confirmation Identify Channel Firmware
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use
tData	Structure Information			
	tFwVersion	Structure		Firmware Version see below
	tFwName	Structure		Firmware Name see below
	tFwDate	Structure		Firmware Date see below

- IDENTIFY FIRMWARE CONFIRMATION

```
#define RCX_FIRMWARE_IDENTIFY_CNF       RCX_FIRMWARE_IDENTIFY_REQ+1
```

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

```
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_T                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 186.

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.

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 43). Then the *HSF_RESET* flag in *bHostSysFlags* (see page 42) 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 *bNetxSysFlags* (page 41), 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 *bNetxSysFlags* is set and an error code is being written in *ulSystemError* in the system status block (see page 44) that helps identifying possible problems.

- SYSTEM RESET COOKIE `#define RCX_SYS_RESET_COOKIE 0x55AA55AA`

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.

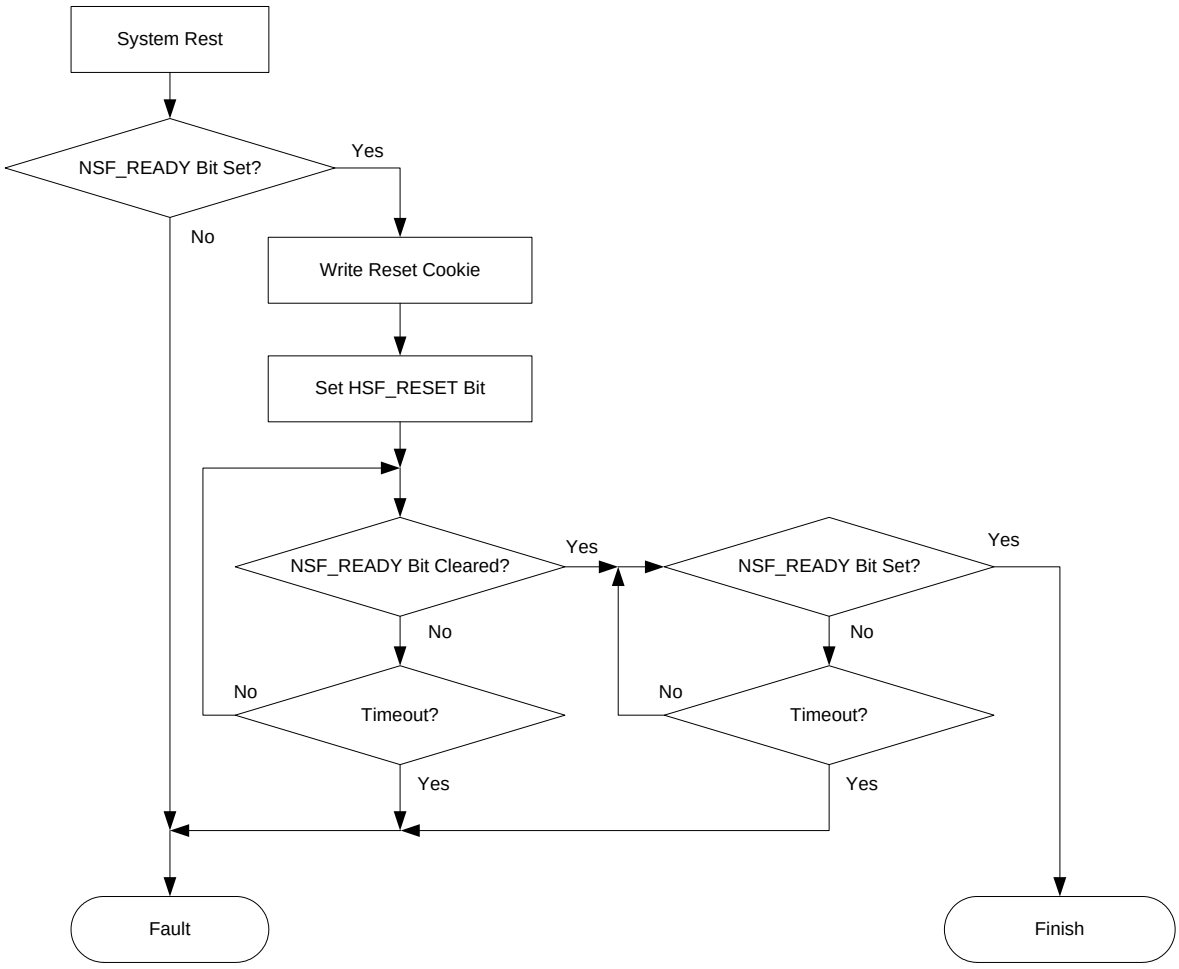


Figure 17 - System Reset Flowchart

Timing

The duration of the reset outlined above, depends on the firmware. Typically the NSF_READY flag is cleard 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 52 and 54).

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 50). Only then, the netX protocol stack recognizes the reset command.

4.9.2.3 Boot Start

The *Boot Start* feature uses a flag from the *bHostSysFlags* register on page 42 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 the *bHostSysFlags* register.
2. Write the system reset cookie into the *ulSystemCommandCOS* variable in the system control block.
3. Set the *HSF_RESET* flag in *bHostSysFlags* 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 52 and 54).

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				
Type: Request				
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

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- CHANNEL RESET REQUEST `#define RCX_FIRMWARE_RESET_REQ 0x00001E00`

Packet Structure Reference

```
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_REQ_Ttag
{
  RCX_PACKET_HEADER_T           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				
Type: Confirmation				
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 7
	ulCmd	UINT32	0x00001E01	Confirmation System Reset
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

- CHANNEL RESET CONFIRMATION

```
#define RCX_CHANNEL_RESET_CNF     RCX_CHANNEL_RESET_REQ+1
```

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

Data Field

There is no data field returned in the Reset confirmation packet.

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				
Type: Request				
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

- DESTINATION: CHANNEL `#define RCX_PKT_COMM_CHANNEL_TOKEN 0x00000020`
- CHANNEL INITIALIZATION REQUEST
  `#define RCX_CHANNEL_INIT_REQ 0x00002F80`

Packet Structure Reference

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

4.9.3.4 Channel Initialization Confirmation

The channel firmware returns the following packet.

Structure Information				
Type: Confirmation				
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 7
	ulCmd	UINT32	0x00002F81	Confirmation Channel Initialization
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

- CHANNEL INITIALIZATION CONFIRMATION

```
#define RCX_CHANNEL_INIT_CNF                    RCX_CHANNEL_INIT_REQ+1
```

Packet Structure Reference

```
typedef struct RCX_CHANNEL_INIT_CNF_Ttag
{
  RCX_PACKET_HEADER_T              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 which are 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 *ulId* 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.

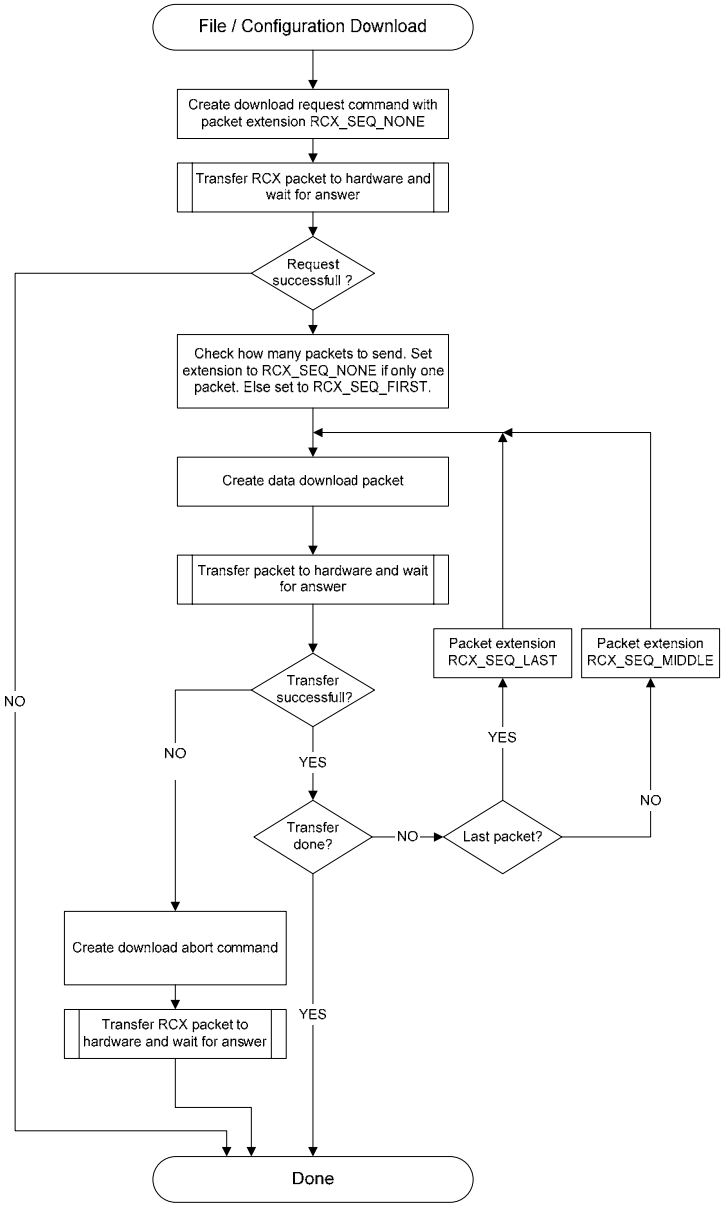


Figure 18 - Flowchart Download

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				
Type: Request				
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**

■ DESTINATION: SYSTEM	`#define`	`RCX_PACKET_DEST_SYSTEM`	`0x00000000`
■ 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`
■ SYSTEM CHANNEL	`#define`	`RCX_SYSTEM_CHANNEL`	`0xFFFFFFFF`
■ COMMUNICATION CHANNEL 0	`#define`	`RCX_COMM_CHANNEL_0`	`0x00000000`
■ COMMUNICATION CHANNEL 1	`#define`	`RCX_COMM_CHANNEL_1`	`0x00000001`
■ COMMUNICATION CHANNEL 2	`#define`	`RCX_COMM_CHANNEL_2`	`0x00000002`
■ COMMUNICATION CHANNEL 3	`#define`	`RCX_COMM_CHANNEL_3`	`0x00000003`

Packet Structure Reference

```
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_T            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				
Type: Confirmation				
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 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 7
	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

- FILE DOWNLOAD CONFIRMATION

```
#define RCX_FILE_DOWNLOAD_CNF     RCX_FILE_DOWNLOAD_REQ+1
```

Packet Structure Reference

```
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_T            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				
Type: Request				
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
	ulChksum	UINT32	S	Checksum CRC32 Polynomial
	abData[n]	UINT8	0 … 0xFF	File Data Block (Size is n)

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- FILE DATA DOWNLOAD `#define RCX_FILE_DATA_DOWNLOAD_REQ 0x00001E64`
- NO SEQUENCED PACKET `#define RCX_PACKET_SEQ_NONE 0x00000000`
- FIRST PACKET OF SEQUENCE `#define RCX_PACKET_SEQ_FIRST 0x00000080`
- SEQUENCED PACKET `#define RCX_PACKET_SEQ_MIDDLE 0x000000C0`
- LAST PACKET OF SEQUENCE `#define RCX_PACKET_SEQ_LAST 0x00000040`

Packet Structure Reference

```
typedef struct RCX_FILE_DOWNLOAD_DATA_REQ_DATA_Ttag
{
  UINT32  ulBlockNo;               /* block number                      */
  UINT32  ulChksum;                /* 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_T              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 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.

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 132) 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				
Type: Confirmation				
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 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 7
	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			
	ulExpected Crc32	UINT32	S	Checksum Expected CRC32 Polynomial

- FILE DATA DOWNLOAD CONFIRMATION
  `#define RCX_FILE_DATA_DOWNLOAD_CNF RCX_FILE_DATA_DOWNLOAD_REQ+1`
- NO SEQUENCED PACKET
  `#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_T               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				
Type: Request				
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

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- ABORT DOWNLOAD REQUEST
  `#define RCX_FILE_ABORT_REQ 0x00001E66`

```
typedef struct RCX_FILE_DOWNLOAD_ABORT_REQ_Ttag
{
  RCX_PACKET_HEADER_T   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				
Type: Confirmation				
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 7
	ulCmd	UINT32	0x00001E67	Confirmation Abort Download Confirmation
	ulExt	UINT32	0x00000000	Extension: None
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

- ABORT DOWNLOAD REQUEST

```
#define RCX_FILE_ABORT_CNF          RCX_FILE_ABORT_REQ+1
```

Packet Structure Reference

```
typedef struct RCX_FILE_DOWNLOAD_ABORT_CNF_Ttag
{
  RCX_PACKET_HEADER_T   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 *ulId* 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.

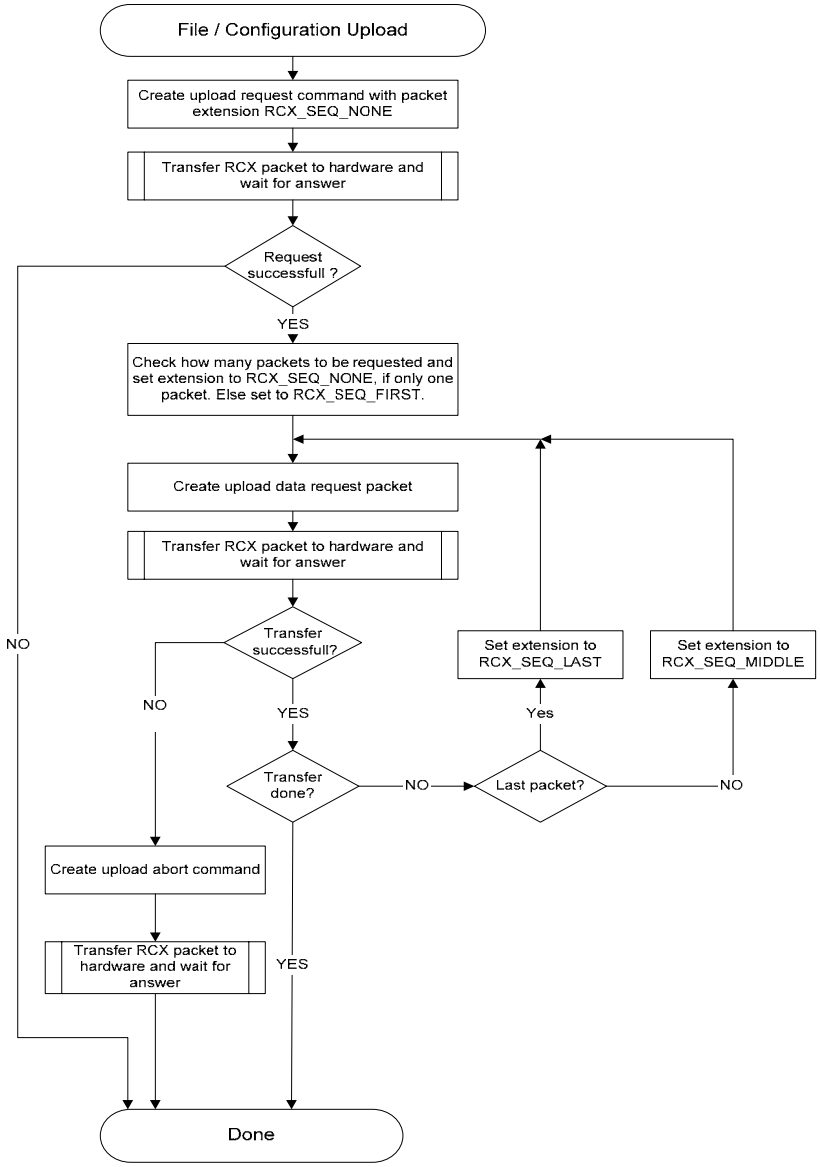


Figure 19 - Flowchart upload

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				
Type: Request				
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**)

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- FILE UPLOAD COMMAND `#define RCX_FILE_UPLOAD_REQ 0x00001E60`
- NO SEQUENCED PACKET `#define RCX_PACKET_SEQ_NONE 0x00000000`
- TRANSFER TYPE `#define RCX_FILE_XFER 0x00000001`
- SYSTEM CHANNEL `#define RCX_SYSTEM_CHANNEL 0xFFFFFFFF`
- COMMUNICATION CHANNEL 0 `#define RCX_COMM_CHANNEL_0 0x00000000`
- COMMUNICATION CHANNEL 1 `#define RCX_COMM_CHANNEL_1 0x00000001`
- COMMUNICATION CHANNEL 2 `#define RCX_COMM_CHANNEL_2 0x00000002`
- COMMUNICATION CHANNEL 3 `#define RCX_COMM_CHANNEL_3 0x00000003`

Packet Structure Reference

```
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_T         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				
Type: Confirmation				
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 7
	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)

- FILE UPLOAD CONFIRMATION
  `#define RCX_FILE_UPLOAD_CNF RCX_FILE_UPLOAD_REQ+1`
- NO SEQUENCED PACKET `#define RCX_PACKET_SEQ_NONE 0x00000000`
- TRANSFER TYPE `#define RCX_FILE_XFER 0x00000001`

Packet Structure Reference

```
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_T         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				
Type: Request				
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

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- FILE DATA UPLOAD REQUEST `#define RCX_FILE_DATA_UPLOAD_REQ 0x00001E6E`
- NO SEQUENCED PACKET `#define RCX_PACKET_SEQ_NONE 0x00000000`
- FIRST PACKET OF SEQUENCE `#define RCX_PACKET_SEQ_FIRST 0x00000080`
- SEQUENCED PACKET `#define RCX_PACKET_SEQ_MIDDLE 0x000000C0`
- LAST PACKET OF SEQUENCE `#define RCX_PACKET_SEQ_LAST 0x00000040`
- TRANSFER TYPE `#define RCX_FILE_XFER 0x00000001`

Packet Structure Reference

```
typedef struct RCX_FILE_UPLOAD_DATA_REQ_Ttag
{
  PACKET_HEADER_T   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				
Type: Confirmation				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure Information			
	ulDest	UINT32	Form 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 7
	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)

- FILE DATA UPLOAD CONFIRMATION

```
#define RCX_FILE_DATA_UPLOAD_CNF          RCX_FILE_DATA_UPLOAD_REQ+1
```

- NO SEQUENCED PACKET

```
#define RCX_PACKET_SEQ_NONE           0x00000000
```

Packet Structure Reference

```
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_T               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				
Type: Request				
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	0x00001E5E	Command Abort Upload Request
	ulExt	UINT32	0x00000000	Extension: None
	ulRout	UINT32	0x00000000	Routing Information, Not Used

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- FILE ABORT UPLOAD REQUEST
  `#define RCX_FILE_ABORT_REQ 0x00001E5E`

Packet Structure Reference

```
typedef struct RCX_FILE_UPLOAD_ABORT_REQ_Ttag
{
  RCX_PACKET_HEADER_T   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				
Type: Confirmation				
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 7
	ulCmd	UINT32	0x00001E5F	Confirmation Abort Upload
	ulExt	UINT32	0x00000000	Extension: None
	ulRout	UINT32	z	Routing Information, Don't Care, Don't Use

- FILE ABORT UPLOAD CONFIRMATION

```
#define RCX_FILE_ABORT_CNF          RCX_FILE_ABORT_REQ+1
```

Packet Structure Reference

```
typedef struct RCX_FILE_UPLOAD_ABORT_CNF_Ttag
{
  RCX_PACKET_HEADER_T   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, 0x1adad47dUL,
  0x6ddde4ebUL, 0xf4d4b551UL, 0x83d385c7UL, 0x136c9856UL, 0x646ba8c0UL, 0xfd62f97aUL, 0x8a65c9ecUL,
  0x14015c4fUL, 0x63066cd9UL, 0xfa0f3d63UL, 0x8d080df5UL, 0x3b6e20c8UL, 0x4c69105eUL, 0xd56041e4UL,
  0xa2677172UL, 0x3c03e4d1UL, 0x4b04d447UL, 0xd20d85fdUL, 0xa50ab56bUL, 0x35b5a8faUL, 0x42b2986cUL,
  0xdbbbc9d6UL, 0xacbcf940UL, 0x32d86ce3UL, 0x45df5c75UL, 0xdcd60dcfUL, 0xabd13d59UL, 0x26d930acUL,
  0x51de003aUL, 0xc8d75180UL, 0xbfd06116UL, 0x21b4f4b5UL, 0x56b3c423UL, 0xcfba9599UL, 0xb8bda50fUL,
  0x2802b89eUL, 0x5f058808UL, 0xc60cd9b2UL, 0xb10be924UL, 0x2f6f7c87UL, 0x58684c11UL, 0xc1611dabUL,
  0xb6662d3dUL, 0x76dc4190UL, 0x01db7106UL, 0x98d220bcUL, 0xefd5102aUL, 0x71b18589UL, 0x06b6b51fUL,
  0x9fbfe4a5UL, 0xe8b8d433UL, 0x7807c9a2UL, 0x0f00f934UL, 0x9609a88eUL, 0xe10e9818UL, 0x7f6a0dbbUL,
  0x086d3d2dUL, 0x91646c97UL, 0xe6635c01UL, 0x6b6b51f4UL, 0x1c6c6162UL, 0x856530d8UL, 0xf262004eUL,
  0x6c0695edUL, 0x1b01a57bUL, 0x8208f4c1UL, 0xf50fc457UL, 0x65b0d9c6UL, 0x12b7e950UL, 0x8bbeb8eaUL,
  0xfcb9887cUL, 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, 0x5edef90eUL, 0x29d9c998UL, 0xb0d09822UL, 0xc7d7a8b4UL, 0x59b33d17UL, 0x2eb40d81UL,
  0xb7bd5c3bUL, 0xc0ba6cadUL, 0xedb88320UL, 0x9abfb3b6UL, 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, 0x8ebeeff9UL, 0x17b7be43UL, 0x60b08ed5UL, 0xd6d6a3e8UL,
  0xa1d1937eUL, 0x38d8c2c4UL, 0x4fdff252UL, 0xd1bb67f1UL, 0xa6bc5767UL, 0x3fb506ddUL, 0x48b2364bUL,
  0xd80d2bdaUL, 0xaf0a1b4cUL, 0x36034af6UL, 0x41047a60UL, 0xdf60efc3UL, 0xa867df55UL, 0x316e8eefUL,
  0x4669be79UL, 0xcb61b38cUL, 0xbc66831aUL, 0x256fd2a0UL, 0x5268e236UL, 0xcc0c7795UL, 0xbb0b4703UL,
  0x220216b9UL, 0x5505262fUL, 0xc5ba3bbeUL, 0xb2bd0b28UL, 0x2bb45a92UL, 0x5cb36a04UL, 0xc2d7ffa7UL,
  0xb5d0cf31UL, 0x2cd99e8bUL, 0x5bdeae1dUL, 0x9b64c2b0UL, 0xec63f226UL, 0x756aa39cUL, 0x026d930aUL,
  0x9c0906a9UL, 0xeb0e363fUL, 0x72076785UL, 0x05005713UL, 0x95bf4a82UL, 0xe2b87a14UL, 0x7bb12baeUL,
  0x0cb61b38UL, 0x92d28e9bUL, 0xe5d5be0dUL, 0x7cdcefb7UL, 0x0bdbdf21UL, 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, 0xdebb9ec5UL,
  0x47b2cf7fUL, 0x30b5ffe9UL, 0xbdbdf21cUL, 0xcabac28aUL, 0x53b39330UL, 0x24b4a3a6UL, 0xbad03605UL,
  0xcdd70693UL, 0x54de5729UL, 0x23d967bfUL, 0xb3667a2eUL, 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				
Type: Request				
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
	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**

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- REQUEST MD5 FILE CHECKSUM REQUEST
  `#define RCX_FILE_GET_MD5_REQ 0x00001E68`

Packet Structure Reference

```
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_T                 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				
Type: Confirmation				
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 7
	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

- REQUEST MD5 FILE CHECKSUM REQUEST
  `#define RCX_FILE_GET_MD5_CNF` `RCX_FILE_GET_MD5_REQ+1`

Packet Structure Reference

```
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_CNF_Ttag
{
  RCX_PACKET_HEADER_T            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				
Type: Request				
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
	usFileName Length	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**

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- FILE DELETE REQUEST `#define RCX_FILE_DELETE_REQ 0x00001E6A`
- COMMUNICATION CHANNEL 0 `#define RCX_COMM_CHANNEL_0 0x00000000`
- COMMUNICATION CHANNEL 1 `#define RCX_COMM_CHANNEL_1 0x00000001`
- COMMUNICATION CHANNEL 2 `#define RCX_COMM_CHANNEL_2 0x00000002`
- COMMUNICATION CHANNEL 3 `#define RCX_COMM_CHANNEL_3 0x00000003`
- SYSTEM CHANNEL `#define RCX_SYSTEM_CHANNEL 0xFFFFFFFF`

Packet Structure Reference

```
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_T          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				
Type: Confirmation				
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 7
	ulCmd	UINT32	0x00001E6B	Command File Delete Confirmation
	ulExt	UINT32	0x00000000	Extension
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

- FILE DELETE REQUEST

```
#define RCX_FILE_DELETE_CNF                    RCX_FILE_DELETE_REQ+1
```

Packet Structure Reference

```
typedef struct RCX_FILE_DELETE_CNF_Ttag
{
  RCX_PACKET_HEADER_T     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				
Type: Request				
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	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
	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**

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- DIRECTORY LIST REQUEST `#define RCX_DIR_LIST_REQ 0x00001E70`
- SYSTEM CHANNEL `#define RCX_SYSTEM_CHANNEL 0xFFFFFFFF`
- COMMUNICATION CHANNEL 0 `#define RCX_COMM_CHANNEL_0 0x00000000`
- COMMUNICATION CHANNEL 1 `#define RCX_COMM_CHANNEL_1 0x00000001`
- COMMUNICATION CHANNEL 2 `#define RCX_COMM_CHANNEL_2 0x00000002`
- COMMUNICATION CHANNEL 3 `#define RCX_COMM_CHANNEL_3 0x00000003`

Packet Structure Reference

```
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_T         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				
Type: Confirmation				
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	6 + n	Packet Data Length (in Bytes)
	ulId	UINT32	Any	Packet Identification as Unique Number
	ulSta	UINT32	See Below	Status / Error Code, see Section 7
	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	<u>File Type</u> 1 2	 Directory File
	bReserved	UINT8	0	Reserved, unused
	usReserved2	UINT16	0	Reserved, unused

- 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
```

Packet Structure Reference

```
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_T     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 52 and 54 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.

- WATCHDOG OFF `#define RCX_WD_OFF 0x00000000`

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 commincation channel, the packet is send through the channel mailbox.

Structure Information				
Type: Request				
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

- DESTINATION: CHANNEL `#define RCX_PKT_COMM_CHANNEL_TOKEN 0x00000020`
- GET WATCHDOG TIME REQUEST
  `#define RCX_GET_WATCHDOG_TIME_REQ 0x00002F02`

Packet Structure Reference

```
typedef struct RCX_GET_WATCHDOG_TIME_REQ_Ttag
{
  RCX_PACKET_HEADER_T              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				
Type: Confirmation				
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 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 7
	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

- GET WATCHDOG TIME CONFIRMATION

```
#define RCX_GET_WATCHDOG_TIME_CNF       RCX_GET_WATCHDOG_TIME_REQ+1
```

Packet Structure Reference

```
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_T                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				
Type: Request				
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

- DESTINATION: CHANNEL `#define RCX_PKT_COMM_CHANNEL_TOKEN 0x00000020`
- SET WATCHDOG TIME REQUEST
  `#define RCX_SET_WATCHDOG_TIME_REQ 0x00002F04`

Packet Structure Reference

```
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_REQ_Ttag
{
  RCX_PACKET_HEADER_T              tHead;   /* packet header             */
  RCX_SET_WATCHDOG_TIME_REQ_DATA_T tData;   /* packet data               */
} RCX_SET_WATCHDOG_TIME_REQ_T;
```

4.15.5 Set Watchdog Time Confirmation

The system channel returns the following packet.

Structure Information				
Type: Confirmation				
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 7
	ulCmd	UINT32	0x00002F05	Confirmation Set Watchdog Time
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

- SET WATCHDOG TIME CONFIRMATION

```
#define RCX_SET_WATCHDOG_TIME_CNF       RCX_SET_WATCHDOG_TIME_REQ+1
```

Packet Structure Reference

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

4.16 Set MAC Address

The Set MAC Address function can be used to store a MAC address into the Security Memory. The existing MAC address will be overwritten. If no Security Memory is available, the return packet indicates that the address could not be stored persistently. In that case, the MAC address is stored temporarily and is lost after power-on-reset (POR).

NOTE The existing MAC address, which is stored in the Security Memory, will be overwritten.

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. So one netX chip uses up to 4 MAC addresses base on the initial MAC address stored in the Security Memory.

4.16.1 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				
Type: Request				
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	0x00000001 else	Parameter Bit Field Store Address Persistently Reserved
	abMacAddr[6]	UINT8		MAC Address
	abPad[2]	UINT8	0x00	Pad Bytes, Set to Zero

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- SET MAC ADDRESS REQUEST
  `#define RCX_SET_MAC_ADDR_REQ 0x00001EEE`
- STORE MAC ADDRESS PERSISTENTLY
  `#define RCX_STORE_MAC_ADDRESS 0x00000001`
- FORCE MAC ADDRESS `#define RCX_FORCE_MAC_ADDRESS 0x00000002`

Packet Structure Reference

```
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_T             tHead;       /* packet header          */
  RCX_SET_MAC_ADDR_REQ_DATA_T     tData;       /* packet data            */
} RCX_SET_MAC_ADDR_REQ_T;
```

4.16.2 Set MAC Address Confirmation

The system channel returns the following packet.

Structure Information				
Type: Confirmation				
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 7
	ulCmd	UINT32	0x00001EEF	Confirmation Set MAC Address
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

- SET MAC ADDRESS CONFIRMATION

```
#define RCX_SET_MAC_ADDR_CNF                    RCX_SET_MAC_ADDR_REQ+1
```

Packet Structure Reference

```
typedef struct RCX_SET_MAC_ADDR_CNF_Ttag
{
  RCX_PACKET_HEADER_T             tHead;       /* packet header          */
} typedef struct 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				
Type: Request				
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

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM` `0x00000000`
- INSTANTIATE FIRMWARE REQUEST
  `#define RCX_CHANNEL_INSTANTIATE_REQ` `0x00001EC4`
- COMMUNICATION CHANNEL 0 `#define RCX_COMM_CHANNEL_0` `0x00000000`
- COMMUNICATION CHANNEL 1 `#define RCX_COMM_CHANNEL_1` `0x00000001`
- COMMUNICATION CHANNEL 2 `#define RCX_COMM_CHANNEL_2` `0x00000002`
- COMMUNICATION CHANNEL 3 `#define RCX_COMM_CHANNEL_3` `0x00000003`

Packet Structure Reference

```
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_T                   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				
Type: Confirmation				
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 7
	ulCmd	UINT32	0x00001EC5	Confirmation Instantiate Firmware
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

- INSTANTIATE FIRMWARE CONFIRMATION

```
#define RCX_CHANNEL_INSTANTIATE_CNF     RCX_CHANNEL_INSTANTIATE_REQ+1
```

Packet Structure Reference

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

Data Field

There is no data field returned in the Start Firmware confirmation packet.

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				
Type: Request				
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	0x00002F10	Command Register Application
	`ulExt`	UINT32	0x00000000	Reserved
	`ulRout`	UINT32	0x00000000	Routing Information

- DESTINATION: CHANNEL `#define RCX_PKT_COMM_CHANNEL_TOKEN 0x00000020`
- REGISTER APPLICATION REQUEST
  `#define RCX_REGISTER_APP_REQ 0x00002F10`

Packet Structure Reference

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

4.18.2 Register Application Confirmation

The system channel returns the following packet.

Structure Information				
Type: Confirmation				
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 7
	ulCmd	UINT32	0x00002F11	Confirmation Register Application
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

- REGISTER APPLICATION CONFIRMATION

```
#define RCX_REGISTER_APP_CNF       RCX_REGISTER_APP_REQ+1
```

Packet Structure Reference

```
typedef struct RCX_REGISTER_APP_CNF_Ttag
{
  RCX_PACKET_HEADER_T              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				
Type: Request				
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

- DESTINATION: CHANNEL `#define RCX_PKT_COMM_CHANNEL_TOKEN 0x00000020`
- UNREGISTER APPLICATION REQUEST
  `#define RCX_UNREGISTER_APP_REQ 0x00002F12`

Packet Structure Reference

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

4.18.4 Unregister Application Confirmation

The system channel returns the following packet.

Structure Information				
Type: Confirmation				
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 7
	ulCmd	UINT32	0x00002F13	Confirmation Unregister Application
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

- UNREGISTER APPLICATION CONFIRMATION

```
#define RCX_UNREGISTER_APP_CNF          RCX_UNREGISTER_APP_REQ+1
```

Packet Structure Reference

```
typedef struct RCX_UNREGISTER_APP_CNF_Ttag
{
  RCX_PACKET_HEADER_T              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 is 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 147 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 send through the channel mailbox to the netX operating system.

Structure Information				
Type: Request				
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

- DESTINATION: CHANNEL `#define RCX_PKT_COMM_CHANNEL_TOKEN 0x00000020`
- DELETE CONFIGURATION REQUEST
  `#define RCX_DELETE_CONFIG_REQ 0x00002F14`

Packet Structure Reference

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

4.19.2 Delete Configuration Confirmation

The system channel returns the following packet.

Structure Information				
Type: Confirmation				
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 7
	ulCmd	UINT32	0x00002F15	Confirmation Delete Configuration
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

- DELETE CONFIGURATION CONFIRMATION

```
#define RCX_DELETE_CONFIG_CNF                RCX_DELETE_CONFIG_REQ+1
```

Packet Structure Reference

```
typedef struct RCX_DELETE_CONFIG_CNF_Ttag
{
  RCX_PACKET_HEADER_T              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 74 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 28.

Structure Information				
Type: Request				
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

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- READ SYSTEM INFORMATION BLOCK REQUEST
  `#define RCX_SYSTEM_INFORMATION_BLOCK_REQ 0x00001E32`

Packet Structure Reference

```
typedef struct RCX_READ_SYS_INFO_BLOCK_REQ_Ttag
{
  RCX_PACKET_HEADER_T              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				
Type: Confirmation				
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	48 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 7
	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 28 for details)

- READ SYSTEM INFORMATION BLOCK CONFIRMATION

```
#define RCX_SYSTEM_INFORMATION_BLOCK_CNF
                                    RCX_SYSTEM_INFORMATION_BLOCK_REQ+1
```

Packet Structure Reference

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

typedef struct RCX_READ_SYS_INFO_BLOCK_CNF_Ttag
{
  RCX_PACKET_HEADER_T                 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 35. Using channel ID, the application can request one block per packet.

Structure Information				
Type: Request				
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

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- READ CHANNEL INFORMATION BLOCK REQUEST
  `#define RCX_CHANNEL_INFORMATION_BLOCK_REQ 0x00001E34`

Packet Structure Reference

```
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_T                 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				
Type: Confirmation				
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 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 7
	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 35 for details)

- READ CHANNEL INFORMATION BLOCK CONFIRMATION

```
#define RCX_CHANNEL_INFORMATION_BLOCK_CNF
                                    RCX_CHANNEL_INFORMATION_BLOCK_REQ+1
```

Packet 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;

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_T                  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 43.

Structure Information				
Type: Request				
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

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- READ SYSTEM CONTROL BLOCK REQUEST
  `#define RCX_SYSTEM_CONTROL_BLOCK_REQ 0x00001E36`

Packet Structure Reference

```
typedef struct RCX_READ_SYS_CNTRL_BLOCK_REQ_Ttag
{
  RCX_PACKET_HEADER_T               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 3.1.5.

Structure Information				
Type: Confirmation				
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 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 7
	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			
	tSystem Control	Structure		System Control Block (see page 43 for details)

- READ SYSTEM CONTROL BLOCK CONFIRMATION

```
#define RCX_SYSTEM_CONTROL_BLOCK_CNF
                                        RCX_SYSTEM_CONTROL_BLOCK_REQ+1
```

Packet Structure Reference

```
typedef struct RCX_READ_SYS_CNTRL_BLOCK_CNF_DATA_Ttag
{
  NETX_SYSTEM_CONTROL_BLOCK_T       tSystemControl;
} RCX_READ_SYS_CNTRL_BLOCK_CNF_DATA_T;

typedef struct RCX_READ_SYS_CNTRL_BLOCK_CNF_Ttag
{
  RCX_PACKET_HEADER_T                   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 44.

Structure Information				
Type: Request				
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

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- READ SYSTEM STATUS BLOCK REQUEST
  `#define RCX_SYSTEM_CONTROL_BLOCK_REQ 0x00001E38`

Packet Structure Reference

```
typedef struct RCX_READ_SYS_STATUS_BLOCK_REQ_Ttag
{
  RCX_PACKET_HEADER_T               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				
Type: Confirmation				
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	64 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 7
	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 44 for details)

- READ SYSTEM STATUS BLOCK CONFIRMATION

```
#define RCX_SYSTEM_CONTROL_BLOCK_CNF
                                        RCX_SYSTEM_CONTROL_BLOCK_REQ+1
```

Packet Structure Reference

```
typedef struct RCX_READ_SYS_STATUS_BLOCK_CNF_DATA_Ttag
{
  NETX_SYSTEM_STATUS_BLOCK_T           tSystemState;
} RCX_READ_SYS_STATUS_BLOCK_CNF_DATA_T;

typedef struct RCX_READ_SYS_STATUS_BLOCK_CNF_Ttag
{
  RCX_PACKET_HEADER_T                  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 74 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 52. 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				
Type: Request				
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

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- DESTINATION: CHANNEL `#define RCX_PKT_COMM_CHANNEL_TOKEN 0x00000020`
- READ COMMUNICATION CONTROL BLOCK REQUEST
  `#define RCX_CONTROL_BLOCK_REQ 0x00001E3A`

Packet Structure Reference

```
typedef struct RCX_READ_COMM_CNTRL_BLOCK_REQ_DATA_Ttag
{
  UINT32  ulChannelId;                        /* channel id              */
} RCX_READ_COMM_CNTRL_BLOCK_REQ_DATA_T;

typedef struct RCX_READ_COMM_CNTRL_BLOCK_REQ_Ttag
{
  RCX_PACKET_HEADER_T                   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 0.

Structure Information				
Type: Confirmation				
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 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 7
	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 52 for details)

- READ COMMUNICATION CONTROL BLOCK CONFIRMATION

```
#define RCX_CONTROL_BLOCK_CNF                RCX_CONTROL_BLOCK_REQ+1
```

Packet Structure Reference

```
typedef struct RCX_READ_COMM_CNTRL_BLOCK_CNF_DATA_Ttag
{
  NETX_CONTROL_BLOCK_T                  tControl;  /* control block        */
} RCX_READ_COMM_CNTRL_BLOCK_CNF_DATA_T;

typedef struct RCX_READ_COMM_CNTRL_BLOCK_CNF_Ttag
{
  RCX_PACKET_HEADER_T                   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 54. 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				
Type: Request				
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	0x00001EFC	Command Read Common Status Block
	`ulExt`	UINT32	0x00000000	Reserved
	`ulRout`	UINT32	0x00000000	Routing Information
`tData`	Structure Information			
	`ulChannelId`	UINT32	0 … 7	Channel Identifier Port Number, Channel Number

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- DESTINATION: CHANNEL `#define RCX_PKT_COMM_CHANNEL_TOKEN 0x00000020`
- READ COMMON STATUS BLOCK REQUEST
  `#define RCX_DPM_GET_COMMON_STATE_REQ 0x00001EFC`

Packet Structure Reference

```
typedef struct RCX_READ_COMMON_STS_BLOCK_REQ_DATA_Ttag
{
  UINT32 ulChannelId;                          /* channel id               */
} RCX_READ_COMMON_STS_BLOCK_REQ_DATA_T;

typedef struct RCX_READ_COMMON_STS_BLOCK_REQ_Ttag
{
  RCX_PACKET_HEADER_T                   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				
Type: Confirmation				
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	64 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 7
	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 54 for details)

- READ COMMON STATUS BLOCK CONFIRMATION

```
#define RCX_DPM_GET_COMMON_STATE_CNF                RCX_DPM_GET_COMMON_STATE_REQ+1
```

Packet Structure Reference

```
typedef struct RCX_READ_COMMON_STS_BLOCK_CNF_DATA_Ttag
{
  NETX_COMMON_STATUS_BLOCK_T           tCommonStatus;  /* common status   */
} RCX_READ_COMMON_STS_BLOCK_CNF_DATA_T;

typedef struct RCX_READ_COMMON_STS_BLOCK_CNF_Ttag
{
  RCX_PACKET_HEADER_T                  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 60. 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				
Type: Request				
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	0x00001EFE	Command Read Extended Status Block
	`ulExt`	UINT32	0x00000000	Reserved
	`ulRout`	UINT32	0x00000000	Routing Information
`tData`	Structure Information			
	`ulChannelId`	UINT32	0 … 7	Channel Identifier Port Number, Channel Number

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- DESTINATION: CHANNEL `#define RCX_PKT_COMM_CHANNEL_TOKEN 0x00000020`
- READ EXTENDED STATUS BLOCK REQUEST
  `#define RCX_DPM_GET_EXTENDED_STATE_REQ 0x00001EFE`

Packet Structure Reference

```
typedef struct RCX_READ_EXT_STS_BLOCK_REQ_DATA_Ttag
{
  UINT32 ulChannelId;                          /* channel id              */
} RCX_READ_EXT_STS_BLOCK_REQ_DATA_T;

typedef struct RCX_READ_EXT_STS_BLOCK_REQ_Ttag
{
  RCX_PACKET_HEADER_T                  tHead;     /* packet header        */
  RCX_READ_EXT_STS_BLOCK_REQ_DATA_T    tData;     /* packet data          */
} RCX_READ_EXT_STS_BLOCK_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				
Type: Confirmation				
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 7
	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			
	tExtended Status	Structure		Extended Status Block (see page 60 for details)

- READ EXTENDED STATUS BLOCK CONFIRMATION

```
#define RCX_DPM_GET_EXTENDED_STATE_CNF
                                        RCX_DPM_GET_EXTENDED_STATE_REQ+1
```

Packet Structure Reference

```
typedef struct RCX_READ_EXT_STS_BLOCK_CNF_DATA_Ttag
{
  NETX_EXTENDED_STATUS_BLOCK_T         tExtendedStatus;
} RCX_READ_EXT_STS_BLOCK_CNF_DATA_T;

typedef struct RCX_READ_EXT_STS_BLOCK_CNF_Ttag
{
  RCX_PACKET_HEADER_T                  tHead;     /* packet header        */
  RCX_READ_EXT_STS_BLOCK_CNF_DATA_T    tData;     /* packet data          */
} RCX_READ_EXT_STS_BLOCK_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				
Type: Request				
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	

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- READ PERFORMANCE COUNTER REQUEST
  `#define RCX_GET_PERF_COUNTERS_REQ 0x00001ED4`

Packet Structure Reference

```
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_T                     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				
Type: Confirmation				
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 x (1+n)) 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 7
	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[0]	Structure		RCX_PERF_COUNTER_DATA_T for structure definition see below
	…	…		…
	tPerfSystemUptime[n-1]	Structure		RCX_PERF_COUNTER_DATA_T for structure definition see below

- READ PERFORMANCE COUNTER CONFIRMATION

```
#define RCX_GET_PERF_COUNTERS_CNF        RCX_GET_PERF_COUNTERS_REQ+1
```

Packet Structure Reference

```
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[n-1]; /*                 */
  /* dynamic array, length is given indirectly by ulLen               */
} RCX_GET_PERF_COUNTERS_CNF_DATA_T;

typedef struct RCX_GET_PERF_COUNTERS_CNF_Ttag
{
  RCX_PACKET_HEADER_T                   tHead;   /* packet header      */
  RCX_GET_PERF_COUNTERS_CNF_DATA_T      tData;   /* packet data        */
} RCX_GET_PERF_COUNTERS_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				
Type: Request				
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

- DESTINATION: CHANNEL `#define RCX_PKT_COMM_CHANNEL_TOKEN 0x00000020`
- GET SLAVE HANDLE REQUEST `#define RCX_GET_SLAVE_HANDLE_REQ 0x00002F08`
- LIST OF CONFIGURED SLAVES `#define RCX_LIST_CONF_SLAVES 0x00000001`
- LIST OF ACTIVATED SLAVES `#define RCX_LIST_ACTV_SLAVES 0x00000002`
- LIST OF FAULTED SLAVES `#define RCX_LIST_FAULTED_SLAVES 0x00000003`

Packet Structure Reference

```
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				
Type: Confirmation				
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 7
	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**

- GET SLAVE HANDLE CONFIRMATION

```
#define RCX_GET_SLAVE_HANDLE_CNF          RCX_GET_SLAVE_HANDLE_REQ+1
```

Packet Structure Reference

```
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_RACKET_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				
Type: Request				
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

- DESTINATION: CHANNEL `#define RCX_PKT_COMM_CHANNEL_TOKEN 0x00000020`
- SLAVE CONNECTION INFORMATION REQUEST
  `#define RCX_GET_SLAVE_CONN_INFO_REQ 0x00002F0A`

Packet Structure Reference

```
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				
Type: Confirmation				
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 7
	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)

- GET SLAVE CONNECTION INFORMATION CONFIRMATION

```
#define RCX_GET_SLAVE_CONN_INFO_CNF   RCX_GET_SLAVE_CONN_INFO_REQ+1
```

Packet Structure Reference

```
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 IO 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				
Type: Request				
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

- DESTINATION: CHANNEL `#define RCX_PKT_COMM_CHANNEL_TOKEN 0x00000020`
- GET DPM I/O INFORMATION REQUEST
  `#define RCX_GET_DPM_IO_INFO_REQ 0x00002F0C`

Packet Structure Reference

```
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 applicatction may receive the packet in a sequenced manner. So the *ulExt* field has to be evaluated!

Structure Information				
Type: Confirmation				
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 7
	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			
	ulBlockNum	UINT32	0 … 10	Number **n** of Block Definitions Below
	tIoBlock[n]	Array of Structrue		I/O Block Definition Structure(s) RCX_IO_BLOCK_INFO_T

- GET DPM I/O INFORMATION CONFIRMATION

```
#define RCX_GET_DPM_IO_INFO_CNF          RCX_GET_DPM_IO_INFO_REQ+1
```

Packet Structure Reference

```
typedef struct RCX_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 of I/O data in bytes             */
  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                    ulBlockNum; /* number of definitions follow */
/*RCX_DPM_IO_BLOCK_INFO_T     tIoBlock[n];  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 96.

Sub Block Type

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

■ UNDEFINED	`#define RCX_BLOCK_UNDEFINED`	`0x0000`
■ UNKNOWN	`#define RCX_BLOCK_UNKNOWN`	`0x0001`
■ PROCESS DATA IMAGE	`#define RCX_BLOCK_DATA_IMAGE`	`0x0002`
■ HIGH PRIORITY DATA IMAGE	`#define RCX_BLOCK_DATA_IMAGE_HI_PRIO`	`0x0003`
■ MAILBOX	`#define RCX_BLOCK_MAILBOX`	`0x0004`
■ CONTROL	`#define RCX_BLOCK_CNTRL_PARAM`	`0x0005`
■ COMMON STATUS	`#define RCX_BLOCK_COMMON_STATE`	`0x0006`
■ EXTENDED STATUS	`#define RCX_BLOCK_EXTENDED_STATE`	`0x0007`
■ USER	`#define RCX_BLOCK_USER`	`0x0008`
■ RESERVED	`#define RCX_BLOCK_RESERVED`	`0x0009`
■ Others are reserved		`0x000A … 0xFFFF`

Flags

The flags field holds information regarding the data transfer direction from the view point of the application. The following flags are defined.

- DIRECTION MASK `#define RCX_DIRECTION_MASK 0x000F`
- UNDEFINED `#define RCX_DIRECTION_UNDEFINED 0x0000`
- IN (netX to Host System) `#define RCX_DIRECTION_IN 0x0001`
- OUT (Host System to netX) `#define RCX_DIRECTION_OUT 0x0002`
- IN - OUT (Bi-Directional) `#define RCX_DIRECTION_IN_OUT 0x0003`
- Others are reserved

The transmission type field in the flags location holds the type of how to exchange data with this block. The choices are:

- TRANSMISSION MASK `#define RCX_TRANSMISSION_TYPE_MASK 0x00F0`
- UNDEFINED `#define RCX_TRANSMISSION_TYPE_UNDEFINED 0x0000`
- DPM (Dual-Port Memory) `#define RCX_TRANSMISSION_TYPE_DPM 0x0010`
- DMA (Direct Memory Access) `#define RCX_TRANSMISSION_TYPE_DMA 0x0020`
- Others are reserved

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 42 - SYS LED

5.4.2 Communication Channel LEDs

The meaning of the communication channel LEDs (COM LED) depends on the actual implementation and is describted in a separate manual.

5.4.3 Force LED Flashing

The following packets are used for diagnosis purposes. It allows the user to identify a communication module or PC card. The firmware flashes a LED for 10 s at 0.25 Hz. There is one packtet that forces the firmware to flash the System LED (SYS LED) and another that forces the firmware to flash the protocol LED (COM LED).

5.4.3.1 Force LED Flashing Request

If the packet outlined below is send to the RCX operating system / system channel (`ulDest = 0x00000000`) the module flashes the System LED. If the packet is send to a communication channel the protocol LED is being flashed.

Structure Information				
Type: Request				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	Structure Information			
	ulDest	UINT32	0x00000000 0x00000001 0x00000002 0x00000003 0x00000004 0x00000020	Destination Queue Handle SYSTEM Communication Channel 0 Communication Channel 1 Communication Channel 2 Communication Channel 3 'Local' 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	0x00002F90	Command Force LED Flashing
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	0x00000000	Routing Information

- DESTINATION: SYSTEM `#define RCX_PACKET_DEST_SYSTEM 0x00000000`
- COMMUNICATION CHANNEL 0 `#define RCX_COMM_CHANNEL_0 0x00000000`
- COMMUNICATION CHANNEL 1 `#define RCX_COMM_CHANNEL_1 0x00000001`
- COMMUNICATION CHANNEL 2 `#define RCX_COMM_CHANNEL_2 0x00000002`
- COMMUNICATION CHANNEL 3 `#define RCX_COMM_CHANNEL_3 0x00000003`
- DESTINATION: CHANNEL `#define RCX_PKT_COMM_CHANNEL_TOKEN 0x00000020`
- FORCE LED FLASHING REQUEST
  `#define RCX_FORCE_LED_FLASH_REQ 0x00002F90`

Packet Structure Reference

```
typedef struct RCX_FORCE_LED_FLASH_REQ_Ttag
{
  RCX_PACKET_HEADER_T              tHead;    /* packet header            */
} RCX_FORCE_LED_FLASH_REQ_T;
```

5.4.3.2 Force LED Flashing Confirmation

The system channel returns the following packet.

Structure Information				
Type: Confirmation				
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 7
	ulCmd	UINT32	0x00002F91	Confirmation Force LED Flashing
	ulExt	UINT32	0x00000000	Reserved
	ulRout	UINT32	Z	Routing Information, Don't Care, Don't Use

- FORCE LED FLASHING CONFIRMATION

```
#define RCX_FORCE_LED_FLASH_CNF          RCX_FORCE_LED_FLASH_REQ+1
```

Packet Structure Reference

```
typedef struct RCX_FORCE_LED_FLASH_CNF_Ttag
{
  RCX_PACKET_HEADER_T             tHead;          /* packet header     */
} RCX_FORCE_LED_FLASH_CNF_T;
```

Data Field

There is no data field returned in the Start Firmware confirmation packet.

6 Configuration

6.1 SYCON.net

SYCON.net used the FDT / DTM model with its ActiveX interfaces (see below).

Configuration in terms of a network setup is carried out with SYCON.net. From the viewpoint of a master device on a network, SYCON.net creates a list of slaves, to which the master shall open a connection. This scan list contains information regarding network slaves as well as network parameters such as baud rate and handshake mode between the netX DPM and the host system. Slave devices are assigned a position (or offset address) within the output data block and input data block of the dual-port memory, if the slave has output and input data. A slave device occupies a certain amount of memory in the master's output and input data area.

SYCON.net creates a proprietary file, which contains configuration information. This file then is downloaded to the netX chip and evaluated by the firmware. Refer to page 124 of how to download a configuration file.

Offset addresses are usually assigned during the configuration process. Configuration in terms of a network setup is carried out with a configuration tool.

6.2 FDT / DTM Concept

FDT stands for *Field Device Type*. The FDT standard was introduced by the FDT Joint Interest Group (www.fdt-jig.org). FDT defines interfaces between so-called the *Device Type Manager* (or DTM) and the engineering tool of the control system manufacturer. The FDT concept focuses on engineering, commissioning, diagnostic and documentation of the fieldbus portion of the control systems. FDT specifies interfaces in ActiveX technology.

A DTM is executed in the context of a FDT container. It uses ActiveX interfaces as defined by the FDT specification. Device manufacturers supply DTMs as a means to configure their devices. The DTM contains manufacturer specific configuration dialogs and generates all settings and parameters to setup the devices. It can read and write parameter from/to the device and it provides customized diagnostic functions. The DTM is independent from the engineering tool. A DTM is installed as a component within the FDT framework.

A good example for the FDT / DTM model is a printer driver. An operating system does not need to know all details of a certain printer; a driver provides a set of standardized function to the operating system to control all functions of this printer. The FDT / DTM concept functions similar: The DTM provides a set of function to the control system and "translates" commands into a language a fieldbus device understands.

The engineering tool and the DTM exchange data by the means of XML files (XML = Extensible Markup Language). The XML file contains information necessary for the engineering tool of the control system to understand the network layout in terms of offset addresses of configured network slaves and their data length in the dual-port memory.

6.3 Online Data Manager ODM

The ODM provides a standardized access to Hilscher communication interface cards. It is based on COM technology (Component Object Model).

Functional Overview

- COM interface, serving multiple applications per device connection
- Support of an unlimited number of hardware devices
- Executable as system service or out-proc server
- Device addressing via human readable strings
- Support of RCS and rcX operating system and user defined data packets
- Device communication via a separate communication layer
- Access to the communication drivers via COM (in-proc server) using a pre-defined interface to allow OEM driver development and implementation
- Automated loading of communication drivers via a defined COM category
- Driver configuration through client application (ActiveX)
- Supports asynchronous and I/O image data transfer
- Support of unsolicited messages for registered server applications
- Flexible trace mechanisms for debugging and monitoring operation
- Supported operating systems: Windows 2000/XP

Currently available ODM drivers

- cifX Device Driver, to communicate to netX based devices, running the rcX operating system, via dual port memory
- CIF Device Driver, to communicate to Hilscher devices, running the RCS operating system, via dual port memory
- 3964R Serial Driver to communicate to Hilscher devices, running the RCS operating system, via the serial diagnostics interface
- TCP/IP Driver to communicate to Hilscher devices, running the RCS operating system, offering a Hilscher TCP/IP server

More details to the Online Data Manager can be found in a separate document.

6.4 Other Configuration Tools

6.4.1 Configuration without SYCON.net

In this section the term *Configuration* is used to describe a method to transfer a set of parameter from the host system to the protocol stack that allows the system to communicate on the network. For example, configuration parameters include among other things baud rate, station name or address and length of input and output data.

The master station opens and maintains connection to network slaves. For that reason the master requires extensive slave parameter settings. Usually, a master stores one parameter set for each of the slaves. So the master firmware has more sophisticated requirements in terms of configuration than a slave firmware.

6.4.1.1 Slave Firmware

Most of the netX slave protocol stacks support configuration via packets. Details of the packet are outlined in the corresponding documentation for the fieldbus firmware. The general mechanism is described here.

A slave protocol stack can be commissioned without downloading a SYCON.net configuration database. Without such a database, the slave stack expects to receive its configuration settings via the mailbox. Therefore the application compiles a packet with appropriate configuration parameter and sends it to the slave protocol stack. The slave stack stores these parameters into its RAM. Then the slave expects the application to execute the *Channel Initialization* procedure (see page 118 for details). Until the initialization procedure is not performed, the slave stack remains in a non-configured mode (Offline mode).

After installation, the slave stack allows network connection to be opened automatically or application request, depending on the setting for *Automatic / Controlled Start of Communication* (see page 86). If the initialization procedure is not performed, the application may overwrite the last configuration settings stored in RAM. The slave firmware will not use them until the initialization procedure is performed.

NOTE During the initialization procedure the protocol stack shuts down all network connections immediately regardless of their current state.

If the current configuration settings are locked (see page 89 for details), the slave will not accept the initialization command. However, it will accept downloading configuration parameter.

NOTE After *System Reset* (see page 117) or power-on reset (POR) all configuration settings stored in RAM are lost and need to be downloaded again.

6.4.1.2 Master Firmware

Details are to be determined.

6.5 Address Table

Details are to be determined.

7 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 specifc protocol stack.

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

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
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

System Error

The system error in the system status block field (see page 44) 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 41 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
else	Others are reserved

8 Appendix

A – 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	A comX 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 100	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 100	The netTAP 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

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	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
NPLC-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
NPLC-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
OEM Device	Original Equipment Manufacturer (OEM) Device, no further information available	0xFFFE
Reserved	Reserved for further use.	0x0009, 0x000A 0x0017-0xFFFD 0xFFFF

Table 43 – Device Class

9 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 | 41 | 48 52 | 55
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 | 19 | 27
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	24 | 18 | 60 |
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 | 41 | 72 48 | 78
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 | 18 | 20 47 | 92
Confirmation	Mechanism used to transfer data via the mailboxes from/to the netX chip: Request ⇨ Indication | Response ⇨ **Confirmation**	72
Data Status	Additional information regarding the state of input and output process data in the IO data image	84
Default Memory Layout (DPM)	Small sized dual-port memory layout with is 16 KByte (one system channel, one handshake channel & one communication channel)	18 | 27
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 | 17 | 18 27 | 92
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.	23 | 52 | 54
FDT/DTM	Field Device Tool / Data Type Manager; Hilscher network configuration tool (⇨ SYCON.net) utilizes FDT/DTM	200
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	125 | 135
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 | 41 | 48 | 64
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 ⇨ **Indication** | Response ⇨ Confirmation	72
Initialization Channel Reset	Reset function that affects one communication channel only, as apposed to the system-wide reset that affects the entire chip	117 | 117 | 118 120
Lock Configuration	Function to protect the configuration settings against being overwritten or otherwise changed	48 | 89
ODM	Online Data Manager; high-level interface to a hardware driver; used by the Hilscher configuration tool (⇨ SYCON.net); open to third-party application	201
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	67
Process Data Image	Cyclic input and output data exchanged with nodes on the network;	25 | 17 | 78
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 | 47
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	72
Reset System Reset	Reset function that affects the entire system including the operating system (⇨ rcX) and all communication channels	117
Response	Mechanism used to transfer data via the mailboxes from/to the netX chip: Request ⇨ Indication | **Response** ⇨ Confirmation	72
Security Memory Security EEPROM	Use to store certain hardware and product related information to help identifying a netX hardware	99
SYCON.net	FDT/DTM (see there) based network configuration tool; used for creating a configuration database that is sent to the netX firmware; database holds information about network settings like baud rate and slaves that are assigned to a master firmware	200
SYS LED	System LED	197
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 | 19 | 27
System Mailbox	Used for a non-cyclic data exchange or to provide access to the firmware running on the netX (send and receive mailbox)	24 | 18 | 46 |
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	52 | 54 | 153
xC Port	Serial communication interface to a fieldbus or network, integrated processor on the netX chip	15 | 29

Table 44 - Glossary

10 Contact

Headquarter

Germany
Hilscher Gesellschaft für
Systemautomation mbH
Rheinstrasse 15
65795 Hattersheim
Phone: +49 (0) 6190 9907-0
Fax: +49 (0) 6190 9907-50
E-Mail: info@hilscher.com

Support
Phone: +49 (0) 6190 9907-99
E-Mail: de.support@hilscher.com

Subsidiaries

China
Hilscher Ges.f.Systemaut. mbH
Shanghai Representative Office
200010 Shanghai
Phone: +86 (0) 21-6355-5161
E-Mail: info@hilscher.cn

Support
Phone: +86 (0) 21-6355-5161
E-Mail: cn.support@hilscher.com

France
Hilscher France S.a.r.l.
69500 Bron
Phone: +33 (0) 4 72 37 98 40
E-Mail: info@hilscher.fr

Support
Phone: +33 (0) 4 72 37 98 40
E-Mail: fr.support@hilscher.com

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

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

Support
Phone: +39/02 25007068
E-Mail: it.support@hilscher.com

Japan
Hilscher Japan KK
Tokyo, 160-0022
Phone: +81 (0) 3-5362-0521
E-Mail: info@hilscher.jp

Support
Phone: +81 (0) 3-5362-0521
E-Mail: jp.support@hilscher.com

Switzerland
Hilscher Swiss GmbH
4500 Solothurn
Phone: +41 (0) 32 623 6633
E-Mail: info@hilscher.ch

Support
Phone: +49 (0) 6190 9907-99
E-Mail: ch.support@hilscher.com

USA
Hilscher North America, Inc.
Lisle, IL 60532
Phone: +1 630-505-5301
E-Mail: info@hilscher.us

Support
Phone: +1 630-505-5301
E-Mail: us.support@hilscher.com