Dual-Port Memory Interface Manual

netX Dual-Port Memory Interface

Hilscher Gesellchaft für Systemautomation mbH

www.hilscher.com

DOC060302DPM13EN | Revision 13 | English | 2017-11 | Released | Public

Table of Content

1 Introduction

1.1 About this Document

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

In a dual-processor system, the netX dual-port memory (DPM) is the interface between a host (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 it is 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 up to 8 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 of the individual memory channels depends on the type of the channel. System channel and handshake channel using a fixed structure and location, while a communication channel can provide some variable areas. The system channel can be used to obtain information regarding type, offset and length of the variable areas.

1.2 List of Revisions

Rev	Date	Name	Revisions
13	2017-11-27	RM	Complete Rework of the document based on revision 12 ▪ Packet-based functions moved to an own manual ▪ Firmware functions and handling moved into a programming reference guide ▪ FLASH Device label information added ▪ Added information about Sync register and flags

Table 1: List of Revisions

1.3 Considerations / Prerequisites and Limitations

The dual-port memory (DPM) and the containing structures and definitions apply to Hilscher netX-based products only. The dual-port memory documented here is not compatible to Hilscher AMD or EC1-based products.

Whenever the term “netX firmware” / "protocol stack" is used throughout this manual, it refers to ready-made firmware provided by Hilscher.

- *Little Endian* Data Representation

  The netX CPU kernel is ARM based and uses the *Little Endian* data representation (LSB/MSB, known as *'Intel format*'), therefore all variables, parameters and data used in this manual corresponding to this representation.

- **C99-Standard-based data types**

  Data which are transferred between different CPU systems (host system / netX system) needs to be fixed in sizes, therefore the C99 standard is used to define the following fixed data types.

Data width	Signed definition	Unsigned definition
8 bit	int8_t	uint8_t
16 bit	int16_t	uint16_t
32 bit	int32_t	uint32_t
64 bit	int64_t	uint64_t

Table 2: Data Types

  Within the rcX operating system alternative names for these data types may be used.

Standard Types	General Definition	TLR Types
int8_t	INT8	TLR_INT8
uint8_t	UINT8	TLR_UINT8
int16_t	INT16	TLR_INT16
uint16_t	UINT16	TLR_UINT16
int32_t	INT32	TLR_INT32
uint32_t	UINT32	TLR_UINT32
int64_t	INT64	TLR_INT64
uint64_t	UINT64	TLR_UINT64

- **Data Packing / Data Alignment**

  Most of the values in the DPM and the non-cyclic command packets are given as C data structures. These C structures are partly byte packed (not always *Natural Aligned*) to safe space in the DPM and the non-cyclic functions. If byte packing is not supported by the host CPU or development environment, some of the structures are not usable and data has to be processed manually to meet this specification and corresponding C structure cannot be used is this case.

- **Additional Terms**

  The terms *Host*, *Host System*, *Application, Host* Application and *Driver* are used interchangeably to identify an external process interfacing the netX via its dual-port memory (DPM).

- **Individual Implementations**

  A netX firmware or protocol stack may support only a subset of the structures and functions described in this document.

- **Host Controlled Mode**
  In *Host Controlled Mode*, the host application initially has access to the I/O data areas in the DPM and can be the first to read and write data before starting a transfer between the netX firmware and the DPM.

1.4 Terms, Abbreviations and Definitions

Term	Description
ACK	Acknowledge
ASCII	American Standard Code of Information Interchange
CMD	Command
COS	Change of State
DMA	Direct Memory Access
DPM	Dual-Port Memory
DRAM	Dynamic Random Access Memory
EC1	80186 based Micro Controller
EEPROM	Electrically Erasable Programmable Read Only Memory
FW	Firmware
FIFO	“First in, first out”, Storage Mechanism
GPIO	General Purpose Input/Output Pins
HMI	Human Machine Interface
Hz	Hertz (1 per Second)
I²C	Inter-Integrated Circuit
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
rcX	Real Time Operating System on netX
RTC	Real Time Clock
s	Second, 1/60 of a Minute
SRAM	Static Random Access Memory
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
WORD	2 Bytes, 16 Bit Entity
xC	Communications Channel on the netX Chip (short form)
xPEC, xMAC	Communications Channel on the netX Chip

Table 3: Terms, Abbreviations and Definitions

1.5 References to documents

[1] Hilscher Gesellschaft für Systemautomation mbH: Packet API, netX Dual-Port Memory, Packet-based services, Revision 1, English.

[2] Hilscher Gesellschaft für Systemautomation mbH: Programming reference guide, netX Dual-Port Memory, Revision 1, English.

[3] Hilscher Gesellschaft für Systemautomation mbH: Function Description, Second Stage Boot Loader, netX 10/50/51/52/100/500, V1.4, Revision 14, English.

1.6 Information and data security

Please take all the usual measures for information and data security, in particular for devices with Ethernet technology. Hilscher explicitly points out that a device with access to a public network (Internet) must be installed behind a firewall or only be accessible via a secure connection such as an encrypted VPN connection. Otherwise the integrity of the device, its data, the application or system section is not safeguarded.

Hilscher can assume no warranty and no liability for damages due to neglected security measures or incorrect installation.

2 Dual-Port Memory Structure

The Dual-Port Memory (DPM) is a structured memory address space in the internal SRAM (INTRAM) of the netX chip. It is the general access to functions and data of a netX firmware. It can be accessed from two sides, the host side and the netX side and provides mechanisms for communication, control, and synchronization.

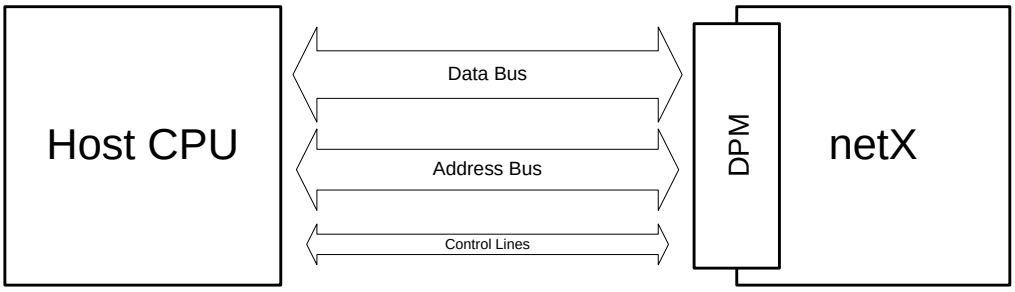


Figure 1: DPM Structure: DPM Connection to netX

The DPM structure is based on the general functionality of a netX firmware.

It is organized in *'Channels'* and various *'Data Blocks'* inside the channels. Each type of channel provides specific functions and information necessary for working with the hardware and firmware.

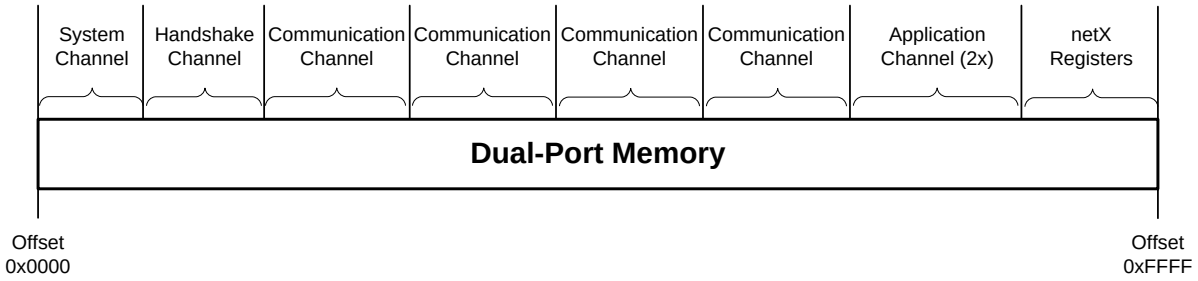


Figure 2: DPM Structure: Overview of DPM Memory Layout

- **System Channel**

  The system channel provides information and functions affecting the entire netX hardware like the version of the operating system or the structure of the dual-port memory and allows basic communication via a mailbox system.

- **Handshake Channel**

  The handshake channel provides synchronization mechanism to ensure data consistency and data access synchronization between a host system and the netX firmware. The synchronization is based on so called *'Handshake Register'* (e.g. bit toggle mechanism via handshake registers) explained later in this manual. In the default layout, the handshake registers from system, communication and application channels are packed together in this channel.

- **Communication Channel / Application Channel**

  System and handshake channel are followed by one or more communication and/or application channels. A *'Communication Channel'* provides access to a fieldbus protocol or network and contains areas for cyclic and acyclic communication data and information. An *'Application Channel'* can be used for any functionality that may be executed in the context of the netX firmware and which does not correspond to a communication channel.

In the example below, two netX communication channels and one application channel are defined where the communication channel corresponds to a protocol stack like PROFINET or DeviceNet. In the example, one of the protocol stacks uses two xMAC/xPEC ports (xC ports).

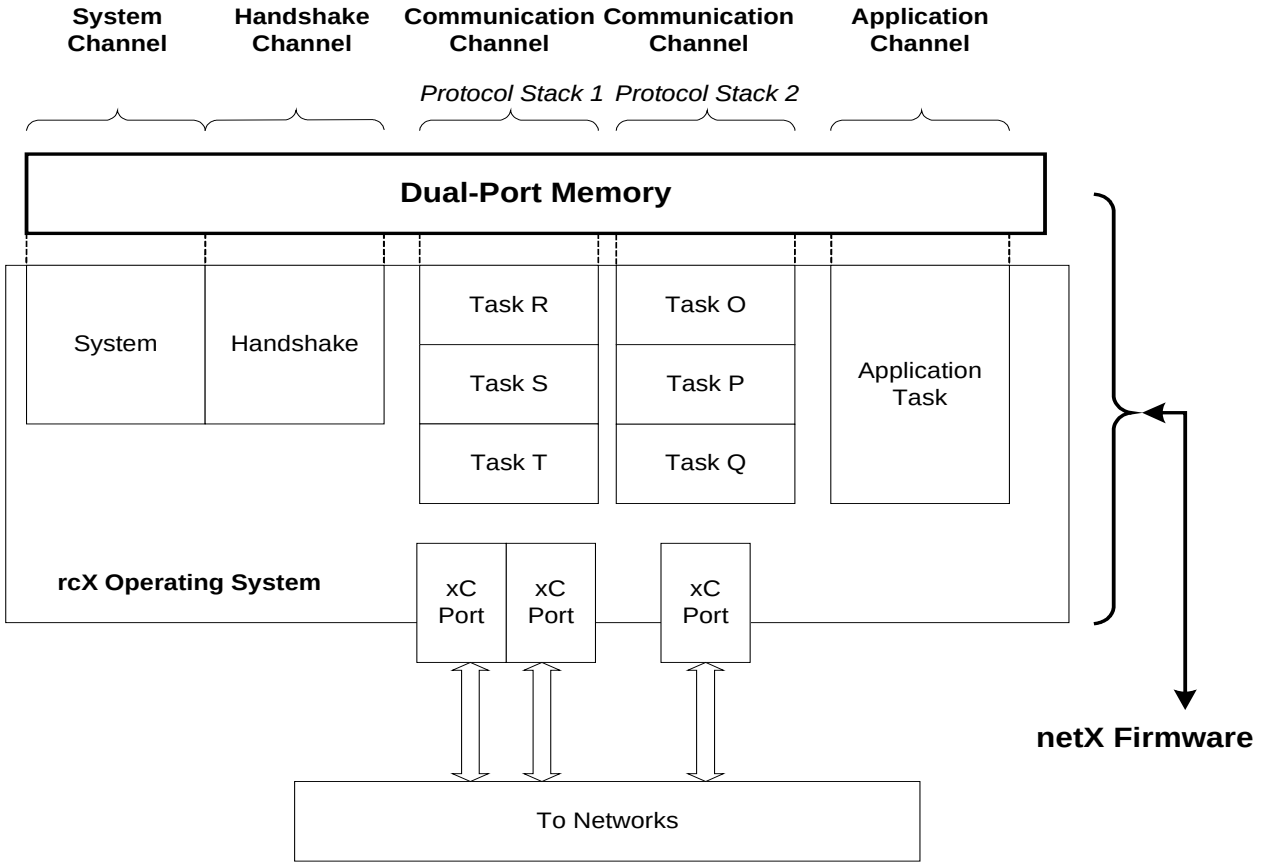


Figure 3: DPM Structure: netX Firmware Block Diagram

2.1 Block Diagram of the default Dual-Port Memory

The block diagram below gives an overview on how the netX firmware organizes the dual-port memory if channels with default configurations are used.

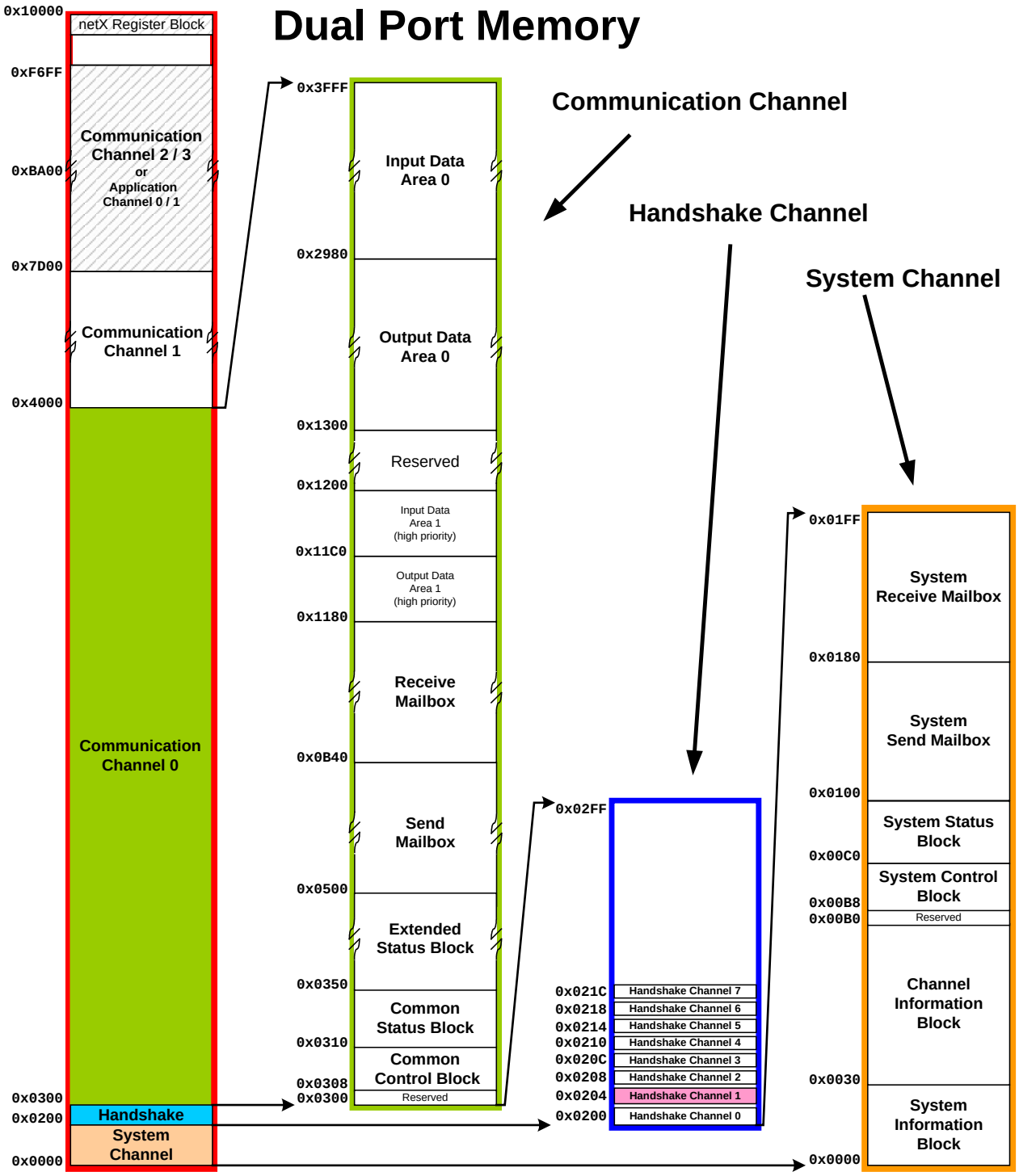


Figure 4: DPM Structure: Block Diagram Default Dual-Port Memory Layout

2.2 Dual-port memory layout and sizes

The DPM address space is available in different variants:

- 64 Kbyte address space, which is considered as the default layout and size (used on PCI-based hardware like CIFX 50 and CIFX 50E PC cards).
- 16 Kbyte address space (used by COMX modules).
- 8 Kbyte address space (used by COMX 10 modules).

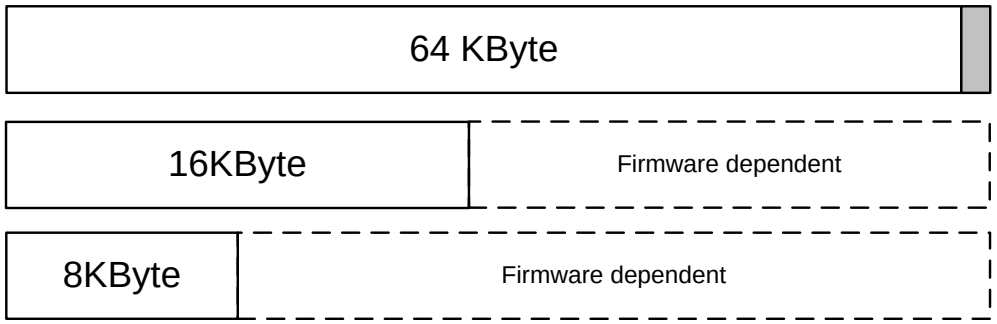


Figure 5: DPM Structure: DPM Address Spaces

Note: If not mentioned otherwise, this document refers to the *64 Kbyte layout* as the '*default memory layout*'.

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

Channel Number	Channel Name	Size 64 KByte	Size 8 KByte	Description
Channel 0	System Channel	512 Bytes	512 Bytes	System (card) related information, state and controls
Channel 1	Handshake Channel	256 Bytes	256 Bytes	Block of synchronization registers for all channels
Channel 2	Communication Channel 0	Variable n * 256 Bytes	Variable n * 256 Bytes	Fieldbus protocol specific information, states and controls
Channel 3	Communication Channel 1	Variable n * 256 Bytes	Not available	Fieldbus protocol specific information, states and controls
Channel 4	Communication Channel 2	Variable n * 256 Bytes	Not available	Fieldbus protocol specific information, states and controls
Channel 5	Communication Channel 3	Variable n * 256 Bytes	Not available	Fieldbus protocol specific information, states and controls
Channel 6	Application Channel 0	Variable n * 256 Bytes	Not available	Custom specific application (optional)
Channel 7	Application Channel 1	Variable n * 256 Bytes	Not available	Custom specific application (optional)
N/A	netX Register Block	512 Bytes	Not available	netX chip specific registers

Table 4: DPM Structure: DPM Layout 8 KByte / 64 Kbyte

2.2.1 Variable Layout

A netX firmware is also able to create variable DPM layouts where communication channel data blocks are variable in size (location is changed by the size).

Such a non-default layout is indicated by the *'Default Memory Map'* flag in the *'System Status Block'* of the *'System Channel'* (in the `ulSystemCOS` field). If the *'Default Memory Map'* flag is not set, the user application can determine the layout of the communication channels by using so called command packages.

A variable DPM layout has the following restrictions:

- System Channel
  Size, location and structure is fixed as defined in this manual
- Handshake Channel
  Size, location and structure is fixed as defined in this manual
- Communication Channel
  - *'Control Block'* is mandatory, always present, structure and size is fix
  - *'Common Status Block'* is mandatory, always present, structure and size is fix
  - *'Send Mailbox'* and *'Receive Mailbox'* are mandatory but variable in size and location
  - *'Input Areas'* and *'Output Areas'* are optional, may not be present or variable in size and location
  - *'Extended Status Block'* is optional, may not be present
- Application Channel are not defined yet

Note: The start address of the communication channels 1 to 4 is variable and depends on the size of preceding communication channels.
The start address of communication channel 0 is always fix because this one follows the system and the handshake channel which are fix in location and size.

Note: The **location** (offset) of a data block inside a channel is not directly defined. It is implicitly given by the channel start address and the size of the preceding blocks/channels.

2.3 Channel Definitions

Channels are structured memory areas in the DPM which contain data blocks. This chapter describes the available channels and the defined data blocks inside a channel.

2.3.1 System Channel

The system channel is always the first channel in the DPM structure. 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 (if no firmware is loaded).

The system channel is located at offset 0x0000 of the dual-port memory and has a fixed size of 512 byte. Inside the channel, the first 256 bytes and the containing structures are also fixed while the following 256 bytes are reserved for the mailbox system. The size of the mailbox structure is by default 128 bytes for the send mailbox and 128 bytes for the receive mailbox.

System Channel		
Data Block Name	**Size**	**Description**
System Information Block	48 Bytes	General system information (e.g. device number / serial number etc.) (see 5.2.1)
Channel Information Block	128 Bytes	Information about available channels and necessary to evaluate variable channel structure information (see 5.2.2)
Reserved	8 Bytes	Reserved, Not Used (see 5.2.3)
System Control Block	8 Bytes	Used for passing control information to the channel (see 5.2.4)
System Status Block	64 Bytes	Used to provide state information to the host system (see 5.2.5)
Mailbox System (Send / Receive Mailbox)	**default: 256 Bytes** (128Byte / 128Byte)	Send Mailbox / Receive Mailbox Used for non-cyclic data transfer of command data organized in packages (see 5.2.6)

Table 5: DPM Structure: System Channel - Overview

For more details about the *System Channel* refer to section 5.2.

2.3.2 Handshake Channel

The handshake channel contains the handshake registers for all channels. These registers with their defined handshake mechanism (see section 3) allow synchronization of data accesses between the host system and the netX.

The handshake channel always starts at DPM offset address 0x0200 and has a fixed size of 256 bytes and a fixed structure.

Handshake Channel		
Data Block Name	**Size**	**Description**
Handshake Register Block	256 Byte	Cumulated handshake register (see 5.3)

Table 6: DPM Structure: Handshake Channel - Overview

Note: Handshake register are special registers inside the DPM. They are able to generate physical interrupts if their content changes. These registers are also used for data access synchronization between a host and the netX system.

For more details about the *Handshake Channel* refer to section 5.3.

2.3.3 Communication Channel

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

Communication Channel		
Data Block Name	**Size**	**Description**
Reserved	8 Byte	Reserved, Not Used (see5.4.1)
Common Control Block	8 Byte	Control Register (see 5.4.2)
Common Status Block	64 Byte	Protocol Stack Status Information (see 5.4.3)
Extended Status Block	432 Byte	Network Specific Information (see 5.4.4)
Mailbox System (Send / Receive Mailbox)	**default: 3200 Byte** (1600Byte / 1600Byte) -> variable	Send Mailbox / Receive Mailbox Used for non-cyclic data transfer of command data organized in packages (see 5.4.5)
I/O Data Area (1) (Output / Input data)	**default: 128Byte** (64Byte / 64 Byte) -> variable	Reserved for the cyclic '*High Priority*' Input / Output Process Data Image (see 5.4.6)
Reserved	**default: 256 Byte** -> variable	Reserved, Not Used (see 5.4.7)
I/O Data Area (0) (Output / Input data)	**default: 11520 Byte** (5760Byte / 5760 Byte) -> variable	Cyclic Input / Output process data image (see 5.4.8)

Table 7: DPM Structure: Communication Channel - Overview

The first communication channel starts always at DPM offset 0x0300 while subsequent channels will follow without a gap in between. The start address of a following channel depends on the size of the preceding one while the size of each channel must be a multiple of 256 bytes.

Depending on the firmware implementation, multiple communication channels could be available.

For more details about the *Communication Channel* refer to section 5.4.

2.3.4 Application Channel

Depending on the implementation, an application channel may or may not be present in the dual-port memory.

The application channel is intended to be used by OEMs if they decide to create an own netX firmware including an additional application which needs the possibility to transfer data between the host and the application via the DPM.

An example for such an application could be a barcode scanner application doing some data preprocessing on netX system.

Application Channel		
Data Block Name	**Size**	**Description**
unknown	unknown	Application Specific, not defined here

Table 8: DPM Structure: Application Channel - Overview

Application channels must follow the rules for communication channels. They will follow preceding channels without a gap in between and the size must be a multiple of 256 bytes.

For more details about the *Application Channel* refer to section 5.5.

2.4 Data Block Definitions

'Data Blocks' are defined structures inside a channel and used to organize function specific data.

2.4.1 System Information Block

The *System Information Block* is only available in the system channel and holds general system depending information (e.g. device number / serial number etc.).

The system info block is written by the netX firmware and read by the host application.

For more information about the content of the *System Information Block* see section 5.2.1.

2.4.2 Channel Information Block

The *Channel Information Block* is only available in the system channel and contains information about available channels. It is needed to evaluate variable channel structure information.

The channel information block is written by the netX firmware and read by the host application.

For more information about the content of the *Channel Information Block* see section 5.2.2.

2.4.3 System Control Block

The *System Control Block* is only available in the system channel and used to pass control information to the general system (e.g. operating system).

The system control block is written by the host application and read by the netX firmware.

For more information about the content and the functionality of the System Control Block see section 5.2.4.

2.4.4 System Status Block

The *System Status Block* is only available in the system channel and provides general system status information like system errors, boot errors, CPU usage etc.

The block is written by the netX firmware and read by the host application.

For more information about the content of the *System Status Block* see section 5.2.5.

2.4.5 Common Control Block

The *Common Control Block* of a communication channel contains commands related to general channel functions which can be activated by writing to a memory address inside the control block. The commands vary between different types of channels, because a system channel offers other functions than a communication channel.

A control block is always present in both system and communication channel.

For some commands, additional information from the *Common Status Block* is necessary to handle the commands correctly.

The common control block is written by the host system while the netX firmware is only allowed to read it.

For more information about the content and the functionality of the *Common Control Block* see section 5.4.2.

2.4.6 Common Status Block

A *Common Status Block* is always present in a communication channel. It contains information about tasks, network states and network related issues.

The common status block is written by the protocol stack and is read by the host system.

For more information about the content of the *Common Status Block* see section 5.4.3.

2.4.7 Extended Status Block

The *Extended Status Block* is a fieldbus protocol specific information block. It is located in a communication channel and contains specific state information about the protocol stack.

The extended status block may be present or not (usually available on most protocol stacks). It is written by the netX firmware / protocol stack and read by the host application.

For more information about the content of the *Extended Status Block* see section 5.4.4.

2.4.8 Mailbox System

The mailbox system provides a non-cyclic data transfer mechanism for packet based commands and confirmations. This mechanism is used to access functions like firmware download, reading firmware information, executing resets, retrieving diagnostic information or to control functionality of a channel by using predefined command packets.

This is always necessary if the required information is not located in the memory area of the DPM.

For more information about the functionality of a mailbox system see section 4.1.

The position and mapping of the system mailbox is described in section 5.2.6 while the channel mailbox description can be found in section 5.4.5.

2.4.9 I/O Data Areas

I/O data areas are used to hold the cyclic process data of a fieldbus protocol stack, which consists of input and output data exchanged between members of a fieldbus network. The areas are dedicated to their direction and named *Input Data Area* and *Output Data Area*. To allow an independent handling of each area, separate data access synchronization is provided for each.

These areas are only provided by a communication channel.

For more information on the functionality of the I/O Data Area see section 4.2.

2.5 netX Chip Register Block

The *netX Register Block* is an area located at the end of the 64 KByte DPM containing internal registers of the netX chip. The availability depends on the netX firmware which must map the registers into the DPM and of the used physical connection to the hardware because some of the hardware does not support the necessary amount of address lines to address a 64 KByte area.

The content of the register block depends on the used netX chip type and is described in the corresponding '*netX Technical Data Reference Guide'*.

Note: It is not recommended to access the *netX Register Block* by a user application. It is only mentioned here because it can be seen on the end of the DPM.

3 Data access and synchronization

Data access to memory areas, shared between two independent CPUs systems, needs to be synchronized, especially if the information in the memory consists of multiple bytes. Synchronization is necessary, because two independent CPUs are not using the same clock source and not running with the same clock speed and therefore it is unpredictable when a CPU access sections of the memory. A single byte access is synchronized by the memory hardware itself (only one read/write access at a time).

This is also valid for the netX DPM and therefore a synchronization mechanism is introduced to ensure data consistency of memory areas which are allowed to be read and written from the netX and the host CPU.

The netX DPM synchronization mechanism is based on so called *Handshake Register*, controlling the access to certain areas inside the DPM (e.g. mailbox systems and I/O data areas) while the access to the handshake registers is strictly regulated to ensure register consistency.

Handshake Register Definition

Every channel has one 32-bit handshake register. This register is subdivided into *Host Register / Host Flags* and *netX Register / netX Flags*. This means, they are either dedicated to the host or to the netX.

- **Host Register / Host Flags**

  These registers and the containing flags are dedicated to the host side. Only the host is allowed to read and write the register while the netX is only allowed to read it.

- **netX Register / netX Flags**

  The netX registers and the containing flags are dedicated to the netX side and only the netX firmware is allowed to read and write these registers while the host is only allowed to read it.

Each channel (*System Channel* and *Communication Channels*) has its own handshake register and in the default DPM layout, these registers are accumulated in the *Handshake Channel* (see section *Handshake Channel* on page 95).

Three types of handshake register are defined:

- **System Channel - Handshake Register**

  Related to the *System Channel* are used by the host application to execute netX-wide (system wide) commands like reset, etc.

- **Communication Channel - Handshake Register**

  Are used to synchronize cyclic data transfer via I/O data areas or non-cyclic data over mailboxes in the communication channels as well as indicating status changes to the host system

- **Synchronization - Handshake Register**

  This register is used to synchronize the host system to fieldbus specific events.

Note: Handshake registers are able to generate physical interrupts when their content is changed.

3.1 Handshake Flag naming convention

Handshake registers contain bits, called handshake flags, where each flag has an assigned function or state depending if it is a member of the host or netX handshake register.

To better distinguish handshake flags, their names follow a simple pattern:

Member of	Location	Function	Signal Type
[H/N]	**[C/S]**	**_F_FUNCTION_**	**[ACK/CMD/**none**]**
			ACK = Acknowledge flag **CMD** = Command flag none = Simple Signal
		Name / Function of the handshake flag	
	S = System Channel flag **C** = Communication Channel flag		
H = Host flag **N** = netX flag			

Table 9: Handshake Flag Naming Convention

Example

- *HSF_SEND_MBX_CMD*
  Member of: HSF **H**ost **S**ystem **F**lags
  Function: SEND_MBX **Send M**ail**b**o**x**
  Signal Type: CMD **C**o**m**man**d** Flag
- *NCF_PD0_IN_CMD*
  Member of: NCF **N**etX **C**ommunication **F**lags
  Function: PD0_IN **P**rocess **D**ata Area **0** - **In**put
  Signal Type: CMD **C**o**m**man**d** Flag
- *NCF_PD0_IN_ACK*
  Member of: NCF **N**etX **C**ommunication **F**lags
  Function: PD0_IN **P**rocess **D**ata Area **0** - **In**put
  Signal Type: ACK **Ack**nowledge Flag

3.2 System Channel - Handshake Register and Flags

The system channel handshake registers and flags are used to synchronize the data transfer between the netX firmware and the host application. They hold information about the status of the rcX operating system and can be used to execute certain commands in the firmware (e.g. a system wide reset).

Note: Handshake registers are located in the handshake channel (see section *Handshake Channel* on page 95) of the DPM.

System Channel - Handshake Register Structure

The system channel handshake register needs less synchronization flags than a communication channel. The length of the flags is 8 bits for the netX firmware (netX Flags = *bNetxFlags*) and 8 bits for the host application (Host Flags = *bHostFlags*).

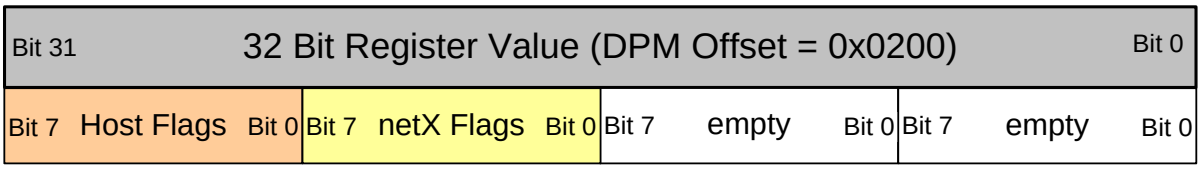


Table 10: System Channel - Handshake Register Structure

System Channel - Handshake Register / Flag Access Definition

- netX System Flags (**NSF**)
  Are read and written by netX firmware, host is only allowed to read the flags
- Host System Flags (**HSF**)
  Are read and written by the host, netX firmware is only allowed to reads these flags

System Channel - Handshake Register DPM Offset

- netX System Channel Register → located at DPM address 0x0202
- Host System Channel Register → located at DPM address 0x0203

System Channel - Handshake Flag Definition

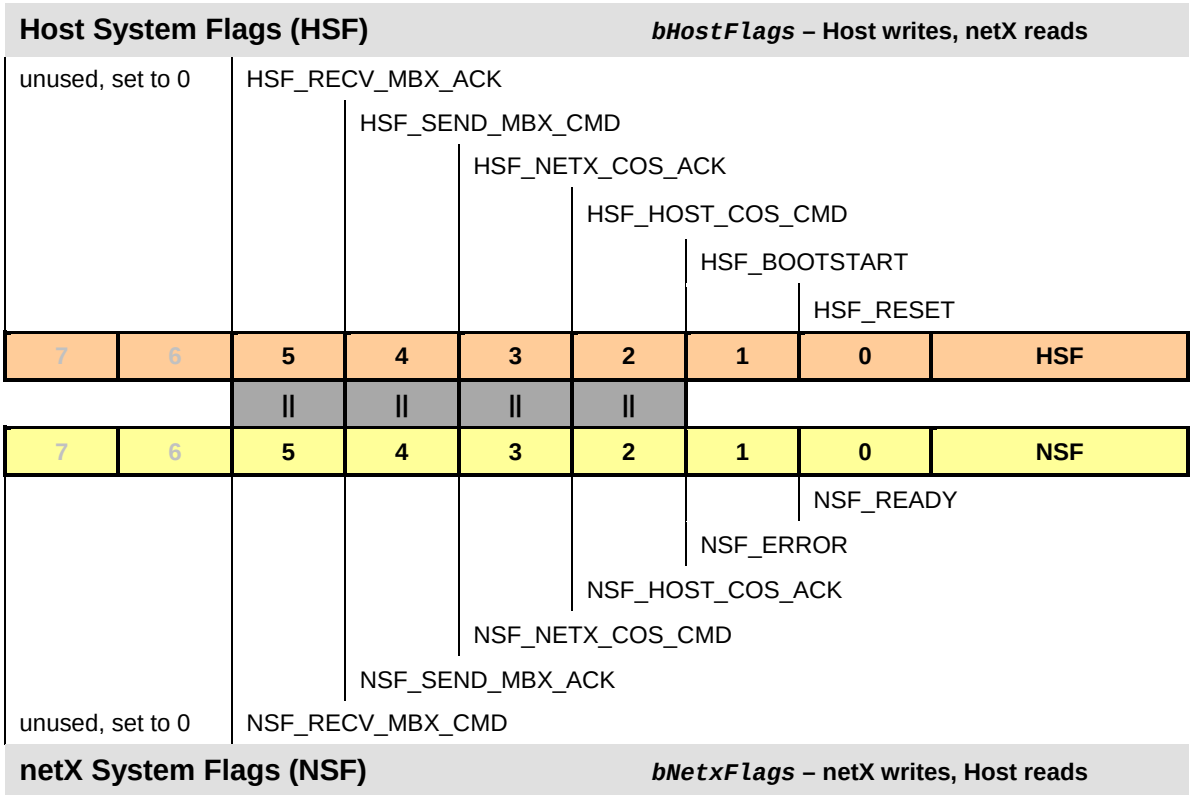


Table 11: System Channel - Handshake Register and Flag Definition

Host System Flags (HSF) (Host ⇨ netX System)

Variable: *bHostFlags*		
Bit	**Definition**	**Description**
0	HSF_RESET	**Reset** 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. For more details see section 6.4.
1	HSF_BOOTSTART	**Boot Start** 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. For more details see section 0.
2	HSF_HOST_COS_CMD	**Host Change Of State Command** 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* field of the *System Control Block* (see section 5.2.4).
3	HSF_NETX_COS_ACK	**netX Change Of State Acknowledge** 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 located in the *netX System Flags.*
4	HSF_SEND_MBX_CMD	**Send Mailbox Command** 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 firmware.

5	HSF_RECV_MBX_ACK	**Receive Mailbox Acknowledge** 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.
6-7		unused, set to zero

Table 12: System Channel - Host System Flags

netX System Flag (NSF) (netX ⇨ Host System)

Variable: *bNetxFlags*		
Bit	**Definition**	**Description**
0	NSF_READY	**Ready** 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 packets.
1	NSF_ERROR	**Error** The *Error* flag is set when the netX has detected an internal error condition. This is considered to be a fatal error and an error code, helping to identify the issue, is placed in the *ulSystemError* field of the *System Status Block* (see section 5.2.5).
2	NSF_HOST_COS_ACK	**Host Change Of State Acknowledge** 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*.
3	NSF_NETX_COS_CMD	**netX Change Of State Command** 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 Status Block* (see section 5.2.5).
4	NSF_SEND_MBX_ACK	**Send Mailbox Acknowledge** 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.
5	NSF_RECV_MBX_CMD	**Receive Mailbox Command** 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 system.
6-7		unused, set to zero

Table 13: System Channel - netX System Flags

3.3 Communication Channel - Handshake Register and Flags

The channel handshake registers and flags are used to control data synchronization of the mailbox system and of the process data image. They are also used to indicate the status of the protocol stack and to execute commands in the protocol stack (e.g. reset of a channel).

Note: Handshake registers are located in the *Handshake Channel* (see section 5.3) of the DPM.

Communication Channel - Handshake Register Structure

The communication channel handshake register defines 16 bits for the netX firmware (netX Flags = `usNetxFlags`) and 16 bits for the host application (Host Flags = `usHostFlags`).

A netX firmware supports up to 4 communication channels, numerated by an index from 0 to 3 with the same structure.

Communication Channel - Handshake Register Structure

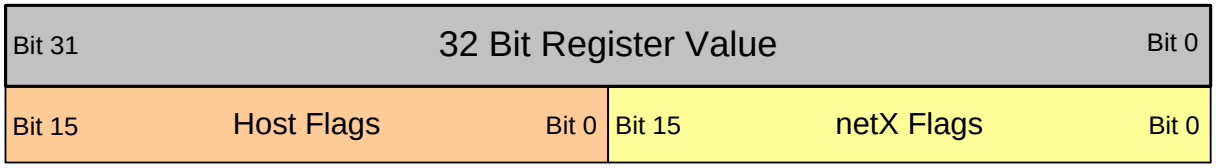


Table 14: Communication Channel - Handshake Register Structure

Communication Channel - Handshake Register / Flag Access Definition

- netX Communication Flags (**NCF**)
  Are read and written by the netX firmware, host is only allowed to read the flags
- Host Communication Flags (**HCF**)
  Are read and written by the host, netX firmware is only allowed to read the flags

Communication Channel - Handshake Register DPM Offset

- netX Communication Channel 0 register → located at DPM address 0x0208
- Host Communication Channel 0 register → located at DPM address 0x020A

Communication Channel - Handshake Flag Definition

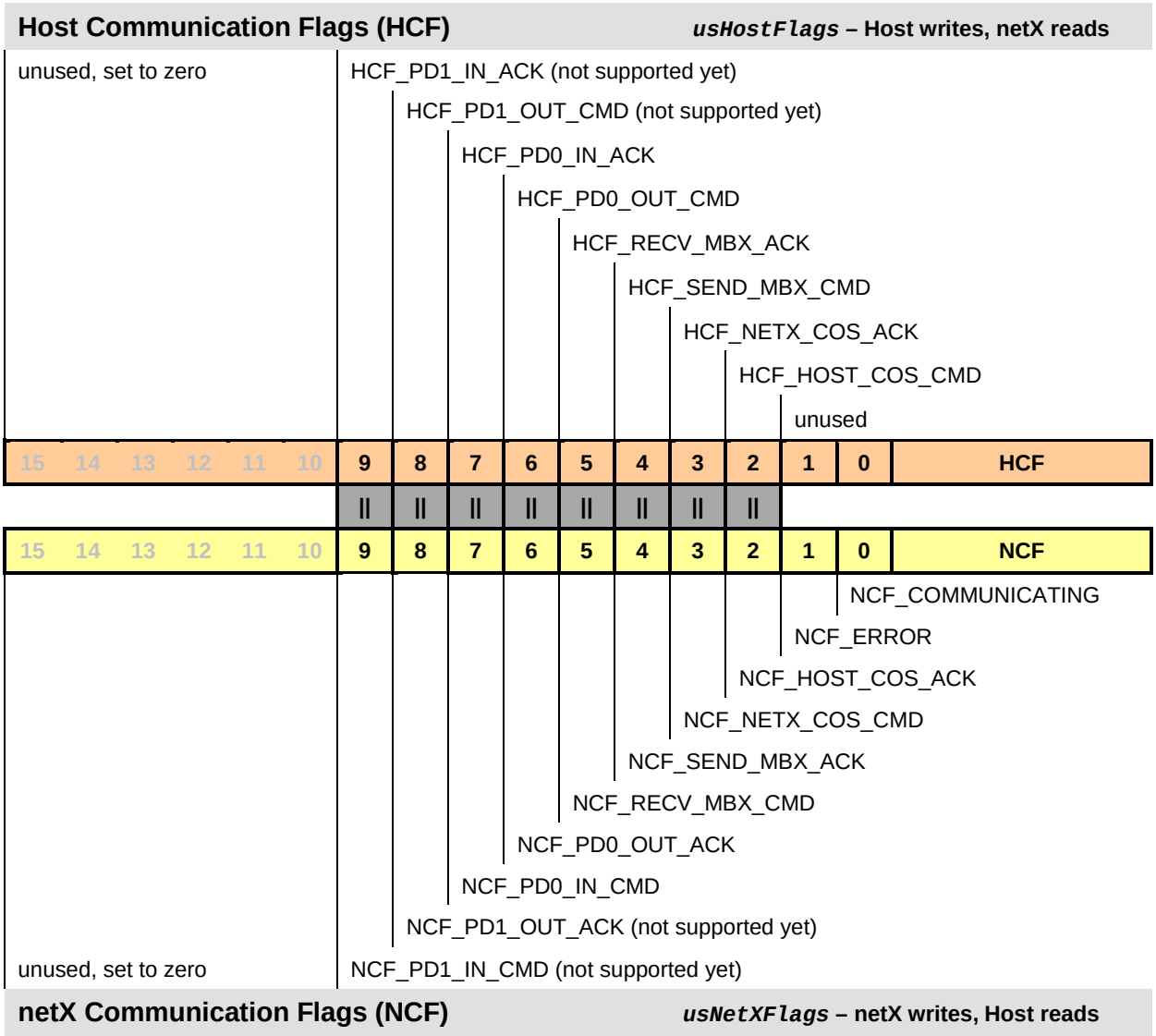


Table 15: Communication Channel - Handshake Register and Flag Definition

Host Communication Flags (HCF) (Application ⇨ netX System)

Variable: *usHostFlags*		
Bit	**Definition**	**Description**
0,1	undefined	unused, set to 0
2	HCF_HOST_COS_CMD	**Host Change Of State Command** The *HCF_HOST_COS_CMD* flag is used to signal a change in the state of the host application. The new state is set in the *ulApplicationCOS* variable in the *Common Control Block* (see section 5.4.2). The protocol stack has acknowledged the processing of the new state by toggling the *NCF_HOST_COS_ACK* flag. At initialization time, this flag is cleared.
3	HCF_NETX_COS_ACK	**Host Change Of State Acknowledge** The *HCF_NETX_COS_ACK* flag is used by host applications to indicate that the new state of the protocol stack has been read. At initialization time, this flag is cleared.
4	HCF_SEND_MBX_CMD	**Send Mailbox Command** Both flags *HCF_SEND_MBX_CMD* and *NCF_SEND_MBX_ACK* are used together to transfer non-cyclic packets between the application and the protocol stack. At initialization time, this flag is cleared.
5	HCF_RECV_MBX_ACK	**Receive Mailbox Acknowledge** Both flags *HCF_RECV_MBX_ACK* and *NCF_RECV_MBX_CMD* are used together to transfer non-cyclic packets between the protocol stack and the application. At initialization time, this flag is cleared.
6	HCF_PD0_OUT_CMD	**Process Data 0 Out Command** Both the *HCF_PD0_OUT_CMD* flag and the *NCF_PD0_OUT_ACK* flag are used together to transfer cyclic output data from the application to the protocol stack. At initialization time, this flag may be set, depending on the data exchange mode.
7	HCF_PD0_IN_ACK	**Process Data 0 In Acknowledge** Both flags *HCF_PD0_IN_ACK* and *NCF_PD0_IN_CMD* flag are used together to transfer cyclic input data from the protocol stack to the application. At initialization time, this flag may be set, depending on the data exchange mode.
8	HCF_PD1_OUT_CMD	**Process Data 1 Out Command** (**not supported yet**) Both flags *HCF_PD1_OUT_CMD* and *NCF_PD1_OUT_ACK* are used together to transfer cyclic output data from the application to the protocol stack.
9	HCF_PD1_IN_ACK	**Process Data 1 In Acknowledge (not supported yet**) Both the *HCF_PD1_IN_ACK* flag and the *NCF_PD1_IN_CMD* flag are used together to transfer cyclic input data from the protocol stack to the application.
10-15		Reserved, set to 0

Table 16: Communication Channel - Host Communication Flags

netX Communication Flags (NCF) (netX ⇨ Application)

Variable: *usNetXFlags*		
Bit	**Definition**	**Description**
0	NCF_COMMUNICATING	**Communicating** *NCF_COMMUNICATING* is set if the protocol stack has successfully opened a connection to at least ONE of the configured network slaves (for master protocol stacks), respectively has an open connection to the network master (for slave protocol stacks). If cleared, the input data should not be evaluated, because it may be invalid, old or both. At initialization time, this flag is cleared.
1	NCF_ERROR	**Error** If set, *NCF_ERROR* signals an error condition that is reported by the protocol stack. The corresponding error code is placed in the *ulCommunicationError* field of the *Common Status Block* (see section 5.4.3). At initialization time, this flag is cleared.
2	NCF_HOST_COS_ACK	**Host Change Of State Acknowledge** The *NCF_HOST_COS_ACK* flag is used by the protocol stack indicating that the new state of the host application has been read. At initialization time, this flag is cleared.
3	NCF_NETX_COS_CMD	**netX Change Of State Command** The *NCF_NETX_COS_CMD* flag is used to signal a change in the state of the protocol stack. The new state is set in the *ulCommunicationCOS* field in the *Common Status Block* (see section 5.4.3). If the host application has read the new protocol state, it has to acknowledge it by toggling the *HCF_NETX_COS_ACK* flag. At initialization time, this flag is cleared.
4	NCF_SEND_MBX_ACK	**Send Mailbox Acknowledge** Both flags NCF_SEND_MBX_ACK and *HCF_SEND_MBX_CMD* are used together to transfer non-cyclic packets between the protocol stack and the application. At initialization time, this flag is cleared.
5	NCF_RECV_MBX_CMD	**Receive Mailbox Command** Both flags *NCF_RECV_MBX_CMD* and *HCF_RECV_MBX_ACK* flag are used together to transfer non-cyclic packets between the application and the protocol stack. At initialization time, this flag is cleared.
6	NCF_PD0_OUT_ACK	**Process Data 0 Out Acknowledge** Both flags *NCF_PD0_OUT_ACK* and *HCF_PD0_OUT_CMD* are used together to transfer cyclic output data from the application to the protocol stack. At initialization time, this flag may be set, depending on the data exchange mode.
7	NCF_PD0_IN_CMD	**Process Data 0 In Command** Both flags *NCF_PD0_IN_CMD* and the *HCF_PD0_IN_ACK* are used together to transfer cyclic input data from the protocol stack to the application. At initialization time, this flag may be set, depending on the data exchange mode.
8	NCF_PD1_OUT_ACK	**Process Data 1 Out Acknowledge** (not supported yet) *NCF_PD1_OUT_ACK* and *HCF_PD1_OUT_CMD* are used together to transfer cyclic output data from the application to the protocol stack.
9	NCF_PD1_IN_CMD	**Process Data 1 In Command** (not supported yet) *NCF_PD1_IN_CMD* and *HCF_PD1_IN_ACK* are used together to transfer cyclic input data from the protocol stack to the application.
10-15		reserved, set to zero

Table 17: Communication Channel - netX Communication Flags

3.4 Synchronization - Handshake Register and Flags

The synchronization handshake register is a special register located in the handshake channel used for fieldbus specific synchronization events.

Synchronization is only available if the fieldbus system supports specific synchronization and it must be configured within the fieldbus configuration.

Note: The fieldbus specific synchronization register is located in the *Handshake Channel* (see section 5.3) at *Handshake Register 1*.

Synchronization - Handshake Register Structure

The synchronization handshake register defines 16 bits for the netX firmware (netX Flags = *usNSyncFlags*) and 16 bits for the host application (Host Flags = *usHsyncFlags*).

Each bit in the synchronization has a fix assignment to one of the possible 4 communication channels starting with bit 0 for communication channel 0.

Synchronization - Handshake Register Layout

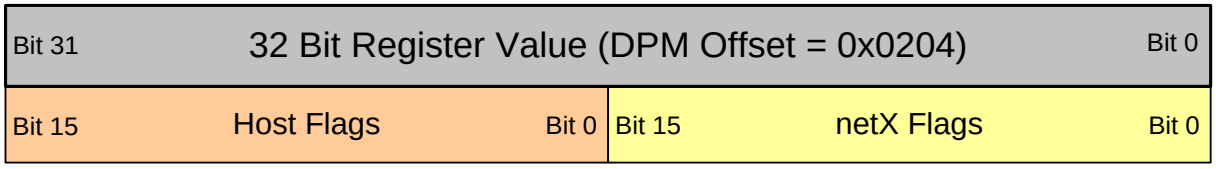


Table 18: Synchronization - Handshake Register Structure

Synchronization - Handshake Register / Flag Access Definition

- netX Synchronization Flags (**NCFSYNC**)
  Are read and written by the netX firmware, host is only allowed to read the flags
- Host Synchronization Flags (**HCFSYNC**)
  Are read and written by the host, netX firmware is only allowed to read the flags

Synchronization - Handshake Register DPM Offset

- netX Synchronization Register → located at DPM address 0x0204
- Host Synchronization Register → located at DPM address 0x0206

Synchronization - Handshake Flag Definition

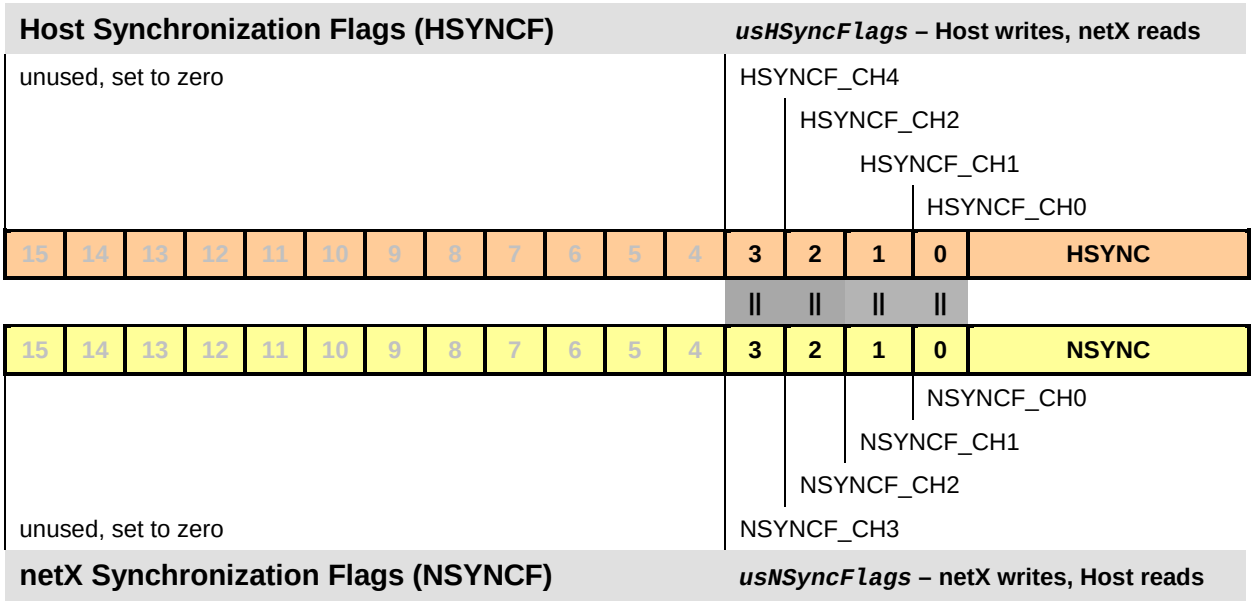


Table 19: Synchronization - Handshake Register and Flag Definition

Host Synchronization (HSYNCF) (Application ⇨ netX System)

Variable: *usHSyncFlags* (DPM offset 0x206)		
Bit	**Definition**	**Description**
0	HSYNCF_CH0	Fieldbus specific synchronization flags: Used to signal synchronization command events to the netX protocol stack or to acknowledge a netX synchronization command event. Each flag is fixed communication channel assignment.
1	HSYNCF_CH1	
2	HSYNCF_CH2	
3	HSYNCF_CH3	
4-15		Reserved, set to 0

Table 20: Synchronization - Host Synchronization Flags

netX Synchronization Flags (NSYNC) (netX ⇨ Application)

Variable: *usNSyncFlags* (DPM offset 0x204)		
Bit	**Definition**	**Description**
0	NSYNCF_CH0	Fieldbus specific synchronization flags: Used to signal synchronization command events to the host application or to acknowledge a host synchronization command event. Each flag is fixed communication channel assignment
1	NSYNCF_CH1	
2	NSYNCF_CH2	
3	NSYNCF_CH3	
4-15		Reserved, set to 0

Table 21: Synchronization - netX Synchronization Flags

Note: The complete description of synchronization handling is located in an own manual (*netX IO Synchronization Manual*).
Consult this manual on how to work with synchronization events.

Synchronization Information - *Common Status Block*

Additional information about the synchronization, like the sync source, sync handshake mode and the sync error counter, is located in the *Common Status Block* of the communication channel.

Type	Variable	Description
`uint8_t`	`bErrorSyncCnt`	Number of synchronization handshake errors Depending on the configuration the error counter is used if the sync information could not be updated because of a missing acknowledgement.
`uint8_t`	`bSyncHskMode`	Synchronization Handshake Mode Configured mode of the synchronization (host controlled / device controlled)
`uint8_t`	`bSyncSource`	Synchronization Source Definition of the sync source (bus cycle / hardware trigger etc.)

Table 22: Synchronization - Synchronization Information

4 Data Transfer Mechanism

The DPM, respectively the netX firmware, provides several data transfer mechanisms and synchronization methods depending on the data areas used for the transfer.

The netX firmware offers several general methods to exchange data with it.

- **Non-Cyclic data transfer via Mailbox and Packets**

  Non-cyclic data is binary data streams named *Packets*. A packet is a structure which consists of a header with general administration information (command / length / source / destination etc) and a data area. The mailbox system contains two memory areas used to exchange packets between the host and the netX device.

- **Cyclic data transfer via Input/Output Data Areas**

  Cyclic data is the fieldbus protocol stacks input and output data which is cyclically exchanged between members of a fieldbus network.

- **Change Of State**

  Used to signal state changes and commands between the host application and a communication channel.

4.1 Non-Cyclic Data Transfer via Mailbox and Packets

Mailboxes are data areas located in a channel and part of a *Mailbox System*. A mailbox system consists of two areas named *Receive Mailbox* and *Send Mailbox* (from the view of the host application) which are dedicated to a transfer direction and used to transfer *Packets* between the host program and the netX firmware, while packets itself describing the data which are transferred.

Each of the area owns a separate data access synchronization allowing an independent handling.

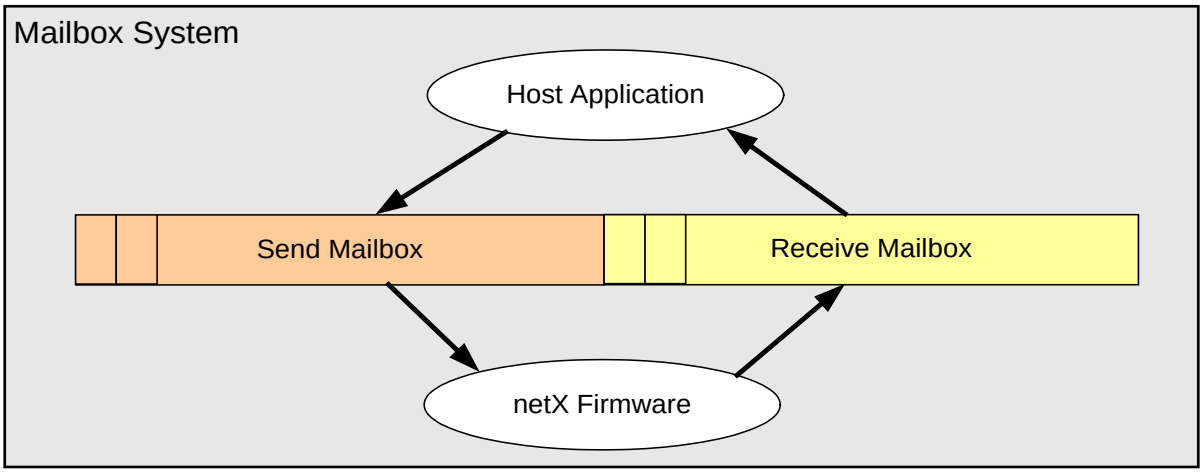


Figure 6: Packets: Mailbox System Overview

Mailbox System

- **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

Both *Send and Receive Mailboxes* are structured into a counter, for tracking the number of active packets and a data area which holds the packet data.

Mailbox Structure (System Channel):

```
/*****************************************************************************/
/*! System send packet mailbox (Size 128 Byte)                               */
/*****************************************************************************/
typedef __RCX_PACKED_PRE struct NETX_SYSTEM_SEND_MAILBOXtag
{
  uint16_t  usPackagesAccepted;                    /*!< Number of packages that can be accepted */
  uint16_t  usReserved;                            /*!< Reserved */
  uint8_t   abSendMbx[NETX_SYSTEM_MAILBOX_MIN_SIZE]; /*!< Send mailbox packet buffer */
} __RCX_PACKED_POST NETX_SYSTEM_SEND_MAILBOX;
/*****************************************************************************/
/*! System receive packet mailbox (Size 128 Byte)                            */
/*****************************************************************************/
typedef __RCX_PACKED_PRE struct NETX_SYSTEM_RECV_MAILBOXtag
{
  uint16_t  usWaitingPackages;                     /*!< Number of packages waiting to be processed */
  uint16_t  usReserved;                            /*!< Reserved */
  uint8_t   abRecvMbx[NETX_SYSTEM_MAILBOX_MIN_SIZE]; /*!< Receive mailbox packet buffer */
} __RCX_PACKED_POST NETX_SYSTEM_RECV_MAILBOX;
```

Note: The mailbox structure of a *Communication Channel* corresponds to the structure of a *System Channel*, except the user data area size is different.

Mailbox Data Buffer Size

Channel	Definition	Size [Bytes]
System Channel	NETX_SYSTEM_MAILBOX_MIN_SIZE	124
Communication Channel	NETX_CHANNEL_MAILBOX_SIZE	1596

Only one packet can be placed into a mailbox at a time (*abSendMbx[]* / *abRecvMbx[]*), while the counter in the send mailbox tracks the amount of packets acceptable by the netX firmware and the counter in the receive mailbox tracks the amount of packets waiting in the receive packet queue of the firmware.

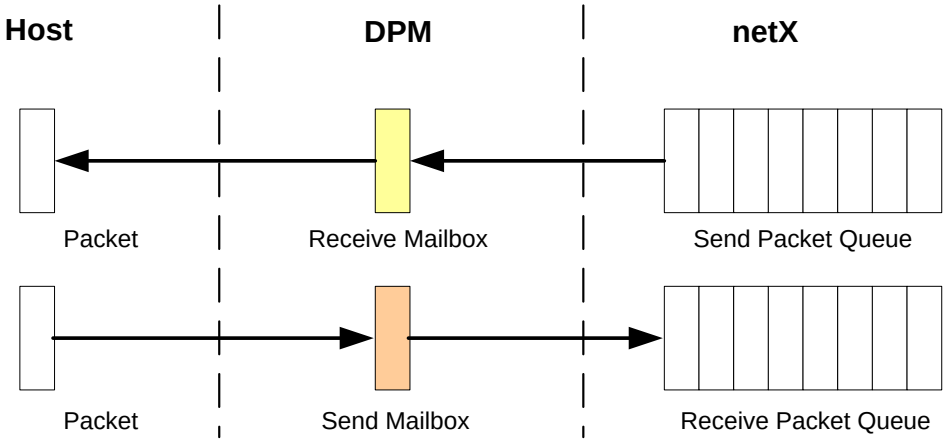


Figure 7: Data Transfer Mechanism: Mailbox Packet Exchange

Note: Each packet will occupy a piece of memory inside the netX firmware. Therefore the amount of concurrent packages is limited by the firmware (**default: 16**).

The netX firmware stores packets into internal memory segments organized in send and receive queues, where the size of the queues is limited.

If the internal packet queues are getting full, the firmware is not able to accept further packets from the send mailbox. This is because each send packet will usually be answered by the netX firmware, which creates a corresponding confirmation packet and stores the answers into the receive packet queue (to be send later on to the host application). In other words, each command packet sent to the firmware generates a confirmation packet which must be read by the host application.

Therefore, if non-cyclic data transfer is used, it is **strongly recommended** to empty the receive mailbox frequently, otherwise the sending of packets can be influenced.

Access to the mailboxes is synchronized by dedicated *Handshake-Register-Flags (*see *Handshake Registers*), signaling the state of a mailbox, which can be either empty or full.

4.1.1 Packet Structure

Packets are asynchronous commands/confirmations which can be exchanged with the firmware. A packet is a data structure containing a *Packet-Header* holding global command/confirmation, routing and handling information, followed by a *User-Data Area* containing the packet specific data.

Structure Information: *RCX_PACKET*				
Packet Header: *RCX_PACKET_HEADER*				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHeader	ulDest	uint32_t	0..0xFFFFFFFF	Destination Address / Handle
	ulSrc	uint32_t	0..0xFFFFFFFF	Source Address / Handle
	ulDestId	uint32_t	0..0xFFFFFFFF	Destination Identifier
	ulSrcId	uint32_t	0..0xFFFFFFFF	Source Identifier
	ulLen	uint32_t	0..max.packet data size	Packet Data Length (in Byte)
	ulId	uint32_t	0..0xFFFFFFFF	Packet Identifier
	ulState	uint32_t	0.. 0xFFFFFFFF	Packet State / Error
	ulCmd	uint32_t	0.. 0xFFFFFFFF	Packet Command / Answer
	ulExt	uint32_t	see below	Packet Extension
	ulRout	uint32_t	0.. 0xFFFFFFFF	Reserved (routing information)
Packet Data				
Area	**Variable**	**Type**	**Value / Range**	**Description**
abData	…	…		Packet Data (packet specific data)

Table 23: Packets: Packet Structure

- Minimum packet size is 40Byte (equal to the packet header structure without user data).
- Maximum packet size (header + data) is limited by the mailbox send / receive area size.

Default Packet Structure

```
/*****************************************************************************/
/*! Default RCX packet header structure                                      */
/*****************************************************************************/
typedef __RCX_PACKED_PRE struct RCX_PACKET_HEADERtag
{
  uint32_t   ulDest;                    /*!< 00:04, Destination of packet, process queue */
  uint32_t   ulSrc;                     /*!< 04:04, Source of packet, process queue */
  uint32_t   ulDestId;                  /*!< 08:04, Destination reference of packet*/
  uint32_t   ulSrcId;                   /*!< 12:04, Source reference of packet */
  uint32_t   ulLen;                     /*!< 16:04, Length of packet data without header */
  uint32_t   ulId;                      /*!< 20:04, Identification handle of sender */
  uint32_t   ulState;                   /*!< 24:04, Status code of operation */
  uint32_t   ulCmd;                     /*!< 28:04, Packet command */
  uint32_t   ulExt;                     /*!< 32:04, Extension */
  uint32_t   ulRout;                    /*!< 36:04, Router (internal use only) */
} __RCX_PACKED_POST RCX_PACKET_HEADER;

/*****************************************************************************/
/*! Default RCX packet structure, including user data                        */
/*****************************************************************************/
typedef __RCX_PACKED_PRE struct RCX_PACKETtag
{
  RCX_PACKET_HEADER tHeader;                    /*!< Packet header */
  uint8_t     abData[RCX_MAX_DATA_SIZE];        /*!< Packet data */
} __RCX_PACKED_POST RCX_PACKET;
```

Parameter Description

Variable	Description
ulDest	**Destination Address / Handle** *ulDest* defines (addresses) the receiver of a packet and must be filled out in any case. The destination address could be either a simple definition or a handle to a task packet input buffer. Definitions are used if packets passed via a mailbox, while handles are used inside a netX firmware. See 4.1.3 Packet Addressing via *ulDest*. A receiving process or task may evaluate this field and has to pass it back unchanged.
ulSrc	**Source Address / Handle** *ulSrc* field identifies the sender of the packet. If used by a host application (outside a netX firmware), any number, which identifies the application uniquely (e.g. process handle), can be used. If used inside a netX firmware (e.g. inter-task communication), this field holds the identifier of the sending task respectively the task input queue handle. See 4.1.4 Using ulSrc and ulSrcId The receiver may evaluate this field and has to pass it back unchanged.
ulDestId	**Destination Identifier** *ulDestId* is an additional field to identify the receiver of a packet. It can hold any number or handle. The packet command (ulCmd) defines if the value is necessary (used) or not. A receiving process or task may evaluate this field and has to pass it back unchanged.
ulSrcId	**Source Identifier** *ulSrcId* can be used by the originator of a command packet to additionally distinguish between sub components for which a packet is generated or sent. A receiving process or task does not evaluate this field and has to pass it back unchanged.
ulLen	**Packet Data Length** *ulLen* defines the length of the user data contained in the packet area (*abData[]*). The size of the packet header is **not** included. The header has a fixed size of 40 bytes. A process or task, creating a confirmation packet, must adjust the packet size according to the size of the confirmation packet data.
ulId	**Packet Identifier** *ulId* is a unique number used to identify multiple packets of the same type (given in *ulCmd*). It allows the originator of the packet to match a specific confirmation with a specific command packet, if multiple packets of the same type are activated. It is up to the originator of a packet if *ulId* is used or not, but it maybe necessary for some services (like fragmented packets). A receiving process or task may evaluate this value and has to pass it back unchanged.
ulState	**Packet State / Error** The *ulState* is used in confirmation packets to indicate failures in the command delivery, packet data or in command processing of the receiving process. Status and error codes that may be returned in *ulState* are outlined in section 7 on page 132. In a command packet, this field has to be zero.
ulCmd	**Packet Command / Answer** *ulCmd* holds the command or confirmation code. Each command *ulCmd* has a specified code, which is always an even number (bit 0 = 0) while the corresponding confirmation is always *ulCmd + 1* (an odd number, bit 0 = 1). It is used by the receiving process to identify the command (even number, excluding zero) and confirmation packets (odd number).

ulExt	**Packet Extension** The extension field *ulExt* is used for controlling packets that are sent in a sequenced or fragmented manner. The 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. 0x00000000 = RCX_PACKET_SEQ_NONE (**default**) 0x00000080 = RCX_PACKET_SEQ_FIRST 0x000000C0 = RCX_PACKET_SEQ_MIDDLE 0x00000040 = RCX_PACKET_SEQ_LAST
ulRout	**Routing Information** The *ulRout* field is used internally by the netX firmware only. It has no meaning to a host application and therefore it should be set to zero for commands and should be returned unchanged in confirmations.
abData	**Packet Data** The *abData* field contains the payload of the packet. Depending on the command (*ulCmd*) or confirmation (*ulCmd +1)* a packet may or may not have a data field. The length of the data field is given in the *ulLen* field.

Table 24: Packets: Packet Description

4.1.2 Default Packet Handling

For a complete transfer, a sent packet has to be answered (confirmed) by the receiving process. The creation of a confirmation packet must follow the rules described below.

Packet Creator

- The packet must be filled with the necessary information (header and/or data).
  Unused fields must be set to 0.
- Confirmations to a command must be verified (*ulCmd, ulState*, *ulLen* and/or content).

Packet Receiver

- Evaluation of the header and/or data content (at least *ulCmd* and *ulLen*).
- Each command must be answered.
  ulState must be set to the processing result (***ulState* = 0 = no Error)**.
  ulCmd must be set to ***ulCmd* + 1** marking the packet as an answer
  (e.g. Command value = 0x02 / Answer value = 0x03).
  ulLen must be set to the amount of data returned in the answer.
- **All other values** of the header must be returned unchanged.
  (*) except when using 4.1.6 Packet Fragmentation is used.

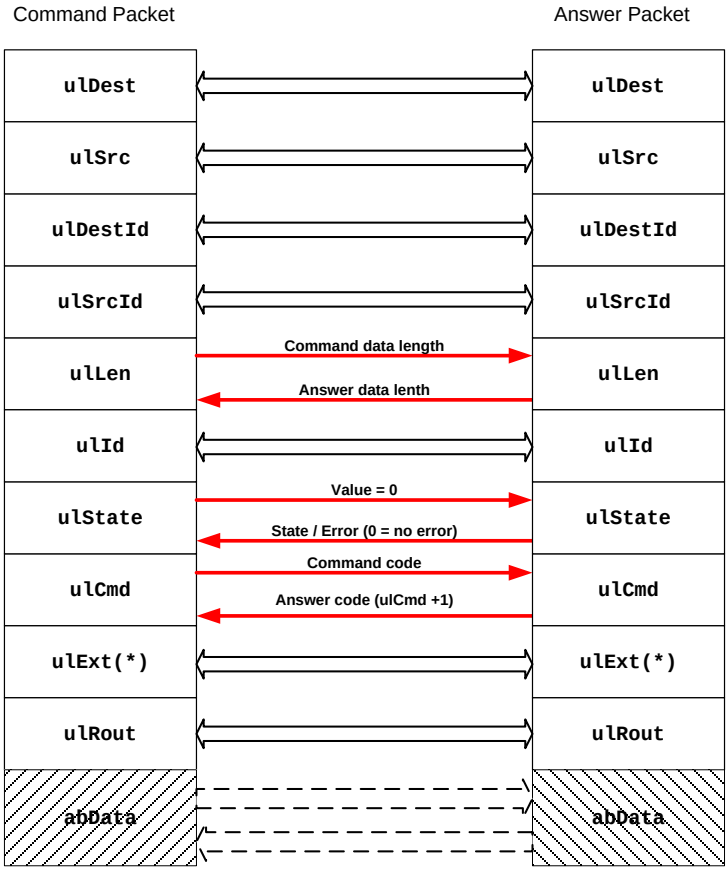


Figure 8: Packets: Default Packet Handling

Note: Default error codes are defined in rcX_User.h and named (RX_E_....)

4.1.3 Packet Addressing via *ulDest*

The receiver of a packet is addressed by the destination identifier *ulDest* in the packet header, which must be filled out according to the specified receiver. The netX firmware offers several addressable targets. A target itself is usually a task inside the firmware offering services.

Possible Packet Targets

- Operating system middleware task of the netX firmware (rcX operating system)
- A fieldbus protocol stack behind a communication channel
- Any task offering public services

Addressing a task inside a netX firmware

Tasks inside a netX firmware are addressed by their task handle created from the tasks dedicated packet queue. This handle can be retrieved by a system command but for this procedure the name of the task must be known.

Addressing a task via a DPM mailbox

To simplify the addressing for host applications working with the DPM, mailboxes are already connected with corresponding tasks inside the firmware.

In case of the *System Channel* mailbox, the connected task is the middleware task of the operating system. The *Communication Channel* mailboxes are connected to the application task (AP-task) of the fieldbus protocol stack. This makes it unnecessary for the host application to know the name of a task and to retrieve task handles from the system.

Default Target Addresses

Definition	Value	Description
RCX_PACKET_DEST_DEFAULT_CHANNEL	0x00000020	Default Channel (**recommended**) Packet is passed to default handler *System Channel* = Middleware *Communication Channel* = Fieldbus protocol
RCX_PACKET_DEST_SYSTEM	0x00000000	Packet is passed to the Middleware Independent of the channel

Table 25: Packets: Default Target Addresses for ulDest

Furthermore, the netX firmware also offers a routing mechanism which allows sending commands to other channels by using the channel index.

Additional Target Addresses

Definition	Value	Description
RCX_PACKET_DEST_PORT_0	0x00000001	Packet passed to communication channel 0
RCX_PACKET_DEST_PORT_1	0x00000002	Packet passed to communication channel 1
RCX_PACKET_DEST_PORT_2	0x00000003	Packet passed to communication channel 2
RCX_PACKET_DEST_PORT_3	0x00000004	Packet passed to communication channel 3

Table 26: Packets: Additional Target Addresses for ulDest

Note: The recommended way to address the middleware of the operating system (rcX) is through the system mailbox and a fieldbus protocol via the communication channel mailbox. This is because additional packets to the middleware via a channel mailbox could influence the performance of the protocol stack handling.

Note: Confirmations are always returned to the mailbox of the channel, which initiated the command.

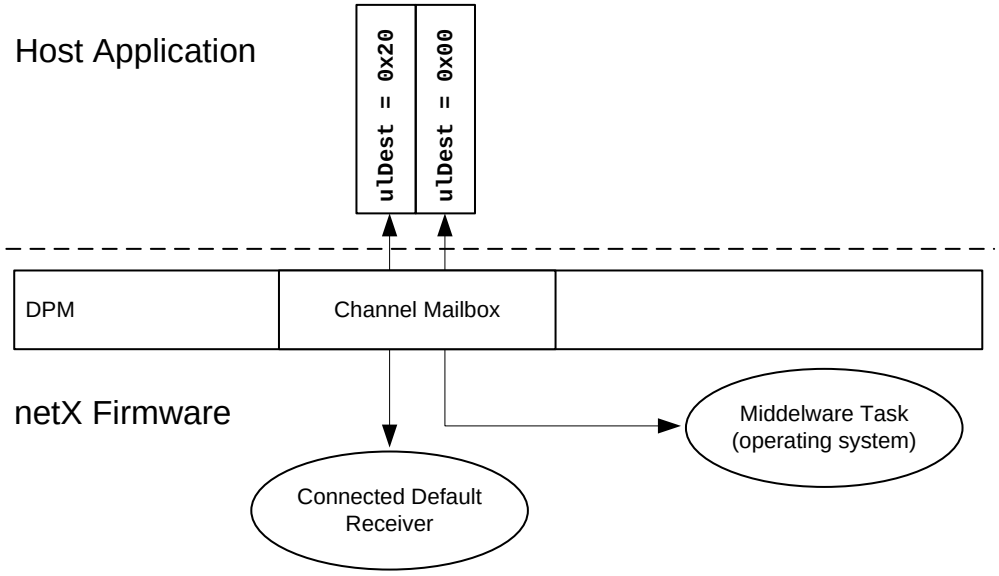


Figure 9: Packets: Target Addressing with ulDest

4.1.4 Using ulSrc and ulSrcId

The originator of a packet is defined by *ulSrc* which is the default identification of the sender (e.g. host application / firmware task). The identification of the packet sender depends on the location of the creating process.

Task inside the netX firmware

In case of a task inside a firmware, *ulSrc* is the task handle or better the address of the packet queue of the task.

Host application outside the netX firmware

Host applications, which are located outside of a netX firmware and using the DPM to communicate to the firmware, can set *ulSrc* to any value including 0.

But it is recommended to use a unique number (e.g. process handle) to prevent collisions with other send processes.

ulSrdId can be freely used by the sending application (e.g. to address internal components).

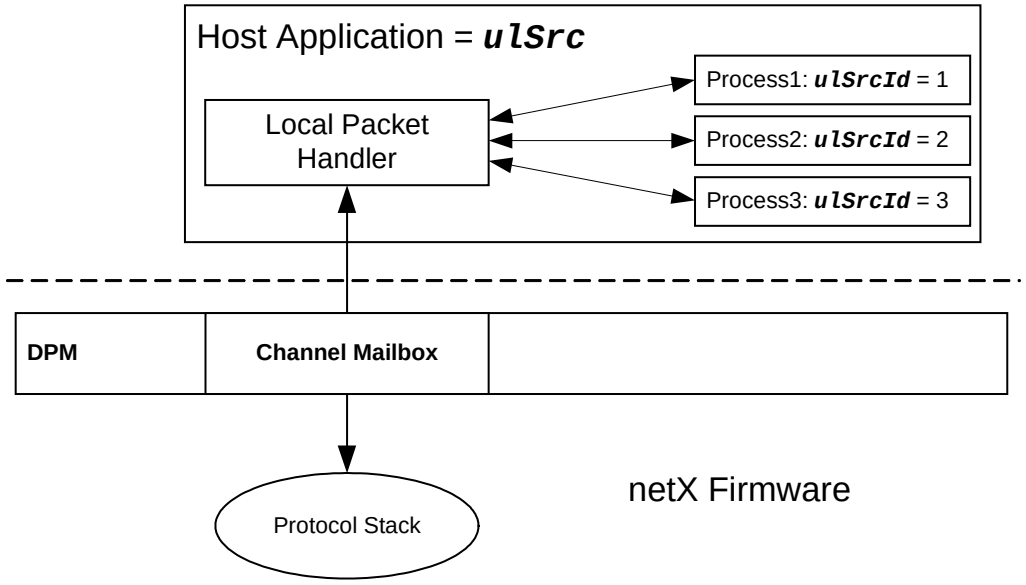


Figure 10: Packets: Using ulSrc *and* ulSrcId

Note: The netX firmware will not touch `ulSrc` and `ulSrcId` and both values are always delivered back in the confirmation packet.

4.1.5 Client/Server Mechanism

The *Client/Server Mechanism* is used to describe which part of the software (host application / netX firmware) has initiated a command and which one has to answer it.

Naming Convention

- Client is the initiator of a command
  Host commands are called requests (REQ) / netX commands are called indications (IND)
- Server is the responder to a command
  netX answers are called confirmations (CNF) / Host answers are called responses (RSP)

By default, the host application will be the *Client* and the netX firmware will be the *Server* in a transfer operation. The netX firmware usually doesn't create commands for the host application. This is because the host application must prepare itself to receive firmware commands and be able to answer them.

But there are several services, offered by a netX firmware / protocol stack, which allow a stack to actively inform the host application about state changes of the fieldbus system (e.g. alarm messages from a fieldbus device). Such firmware requests must be activated by the host application by sending an application register request command to the protocol stack. In this situation, the netX firmware becomes the *Client* and the host application the *Server*.

For a better distinction between *Client* and *Server* in conjunction with the transfer direction, commands are specified as *Requests* / *Indications* and answers are specified as *Confirmation* / *Response*.

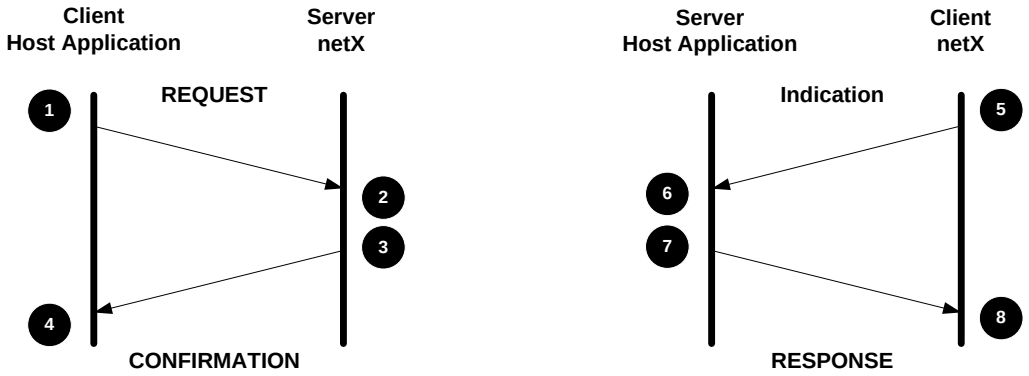


Figure 11: Client/Server mechanism

Host to netX		netX to Host	
Direction	**Description**	**Direction**	**Description**
❶ ➔ ❷	Host application sends a *Request* packet to the netX firmware	❺ ➔ ❻	Host application receives an *Indication* packet from the netX firmware
❸ ➔ ❹	netX firmware answers by a *Confirmation* packet	❼ ➔ ❽	Host application sends a *Response* packet back to the netX firmware (may not be required

Note: Indications must be explicitly activated by the host application.

4.1.6 Packet Fragmentation

The mechanism of transferring packets in a fragmented manner is used when the size of the packet (packet header and user data) exceeds the size of the mailbox or when the confirmation to a command packet could have a variable data size, which exceeds the size of the mailbox area.

Note: *Fragmenting Packets* is not a default mechanism for all packet commands. The general handling is described here and if support it is explicitly noted in the packet command definition!

Handling

Packet fragmentation is handled by the *ulExt* and *ulId* field in the packet header. The *ulExt* field defines if a packet belongs to a sequence and indicates the state of a sequenced transfer (first/middle/last). *ulId* is used as a packet index inside a sequence to ensure a strict packet order handling during a transfer.

Note: Fragmented packets must be sent in a strict order (given by *ulId*). Out of order transfers are not supported.

Header Variable	Description
ulExt	Indication of a sequenced packet transfer 0x00000000 = RCX_PACKET_SEQ_NONE — None sequenced (**default**) 0x00000080 = RCX_PACKET_SEQ_FIRST — First packet of a sequence 0x000000C0 = RCX_PACKET_SEQ_MIDDLE — Packet inside a sequence 0x00000040 = RCX_PACKET_SEQ_LAST — Last packet of a sequence
ulId	Packet number inside a sequence - Start value is 0 (not mandatory) - Incremented by one for each packet sequence

Table 27: Packet Fragmentation: Extension and Identifier Field

Example

Fragmented Transfer Host → netX Firmware - Initiated by host application

Host application knows that data does not fit into the send mailbox.

Step	Direction (App | Task)	Description	*ulCmd*	*ulId*	ulExt
1	→	Command	CMD	0	RCX_PACKET_SEQ_FIRST
	←	Answer	CMD + 1		RCX_PACKET_SEQ_FIRST
2	→	Command	CMD	1	RCX_PACKET_SEQ_MIDDLE
	←	Answer	CMD + 1		RCX_PACKET_SEQ_MIDDLE
3	→	Command	CMD	2	RCX_PACKET_SEQ_MIDDLE
	←	Answer	CMD + 1		RCX_PACKET_SEQ_MIDDLE
…	…	…	…	…	…
n	→	Command	CMD	n	RCX_PACKET_SEQ_LAST
	←	Answer	CMD + 1		RCX_PACKET_SEQ_LAST

Table 28: Packet Fragmentation: Example - Host to netX Firmware

Fragmented Transfer netX Firmware → Host - Initiated by host application

Host application does not know how many packets will be received.

Step	Direction (App | Task)	Description	*ulCmd*	*ulId*	ulExt
1	→	Command	CMD	0	RCX_PACKET_SEQ_NONE
	←	Answer	CMD + 1		**RCX_PACKET_SEQ_FIRST**
2	→	Command	CMD	1	RCX_PACKET_SEQ_MIDDLE
	←	Answer	CMD + 1		RCX_PACKET_SEQ_MIDDLE
3	→	Command	CMD	2	RCX_PACKET_SEQ_MIDDLE
	←	Answer	CMD + 1		RCX_PACKET_SEQ_MIDDLE
…	…	…	…	…	…
n	→	Command	CMD	n	RCX_PACKET_SEQ_MIDDLE
	←	Answer	CMD + 1		RCX_PACKET_SEQ_LAST

Table 29: Packet Fragmentation: Example - netX Firmware to Host

General Abort Handling

If an error is detected during a fragmented packet transfer, the transfer has to be aborted before the last packet is transferred. Examples for a fragmented packet transfers are file download and file upload functions.

Possible Errors:

- *ulId*, index skipped, used twice, out of order
- *ulExt*, state out of order
- *ulSta* in the answer not zero, returned by the answering process

Note: Packet services are described in the *netX DPM Packet Services* manual. If a service needs an abort handling will be mentioned in this manual.

Abort – Command

Structure Information: *RCX_PACKET_HEADER*				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	ulDest	UINT32	n	Destination Address / Handle
	ulSrc	UINT32	n	Source Address / Handle
	ulDestId	UINT32	n	Destination Identifier
	ulSrcId	UINT32	n	Source Identifier
	ulLen	UINT32	**0**	Packet Data Length (in Byte)
	ulId	UINT32	**ANY**	Packet Identifier
	ulState	UINT32	0	Packet State / Error
	ulCmd	UINT32	**Active CMD**	Packet Command / Confirmation
	ulExt	UINT32	**0x00000040**	Packet Extension Last Packet of Sequence
	ulRout	UINT32	0x00000000	Reserved (routing information)

Table 30: Packet Fragmentation: Abort Command

Abort - Confirmation

Structure Information: *RCX_PACKET_HEADER*				
Area	**Variable**	**Type**	**Value / Range**	**Description**
tHead	ulDest	UINT32	From Request	Destination Address / Handle
	ulSrc	UINT32	From Request	Source Address / Handle
	ulDestId	UINT32	From Request	Destination Identifier
	ulSrcId	UINT32	From Request	Source Identifier
	ulLen	UINT32	**0**	Packet Data Length (in Byte)
	ulId	UINT32	From Request	Packet Identifier
	ulState	UINT32	**0**	Packet State / Error RCX_S_OK (always)
	ulCmd	UINT32	**CMD+1**	Packet Command / Confirmation
	ulExt	UINT32	**0x00000040**	Packet Extension Last Packet of Sequence
	ulRout	UINT32		Reserved (routing information)

Table 31: Packet Fragmentation: Abort Confirmation

4.1.7 Packet transfer synchronization

The transfer of packets via the mailbox system is synchronized by the corresponding handshake flags in the handshake flag registers of each channel (see *usNetxFlags* / *usHostFlags*).

The following flag pairs are used to synchronize the packet transfer / mailbox handling.

- System Channel Flags

  HSF_SEND_MBX_CMD / NSF_SEND_MBX_ACK
  NSF_RECV_MBX_CMD / HSF_RECV_MBX_ACK

- Communication Channel Flags

  HCF_SEND_MBX_CMD / NCF_SEND_MBX_ACK
  NCF_RECV_MBX_CMD / HCF_RECV_MBX_ACK

Each mailbox has an owner allowed to initiate a transfer. The owner of the *Send Mailbox* is the host application while the *Receive Mailbox* is owned by the netX firmware. And a mailbox can have 2 states (EMPTY / FULL), signaled by the corresponding handshake flags.

General Mailbox Definition

***SYSTEM CHANNEL* - Send Mailbox** / Owner: Host Application				
Handshake Flag	**Status**		**Handshake Flag**	**Mailbox State**
HSF_SEND_MBX_CMD	0 1	0 1	NSF_SEND_MBX_ACK	EMPTY
	1 0	0 1		FULL
***SYSTEM CHANNEL* - Receive Mailbox** / Owner: netX Firmware				
NSF_RECV_MBX_CMD	0 1	0 1	HSF_RECV_MBX_ACK	EMPTY
	1 0	0 1		FULL

Table 32: Packet: System Channel Mailbox State Definition

***COMMUNICATION CHANNEL* - Send Mailbox** / Owner: Host Application				
Handshake Flag	**Status**		**Handshake Flag**	**Mailbox State**
HCF_SEND_MBX_CMD	0 1	0 1	NCF_SEND_MBX_ACK	EMPTY
	1 0	0 1		FULL
***COMMUNICATION CHANNEL* - Receive Mailbox** / Owner: netX Firmware				
NCF_RECV_MBX_CMD	0 1	0 1	HCF_RECV_MBX_ACK	EMPTY
	1 0	0 1		FULL

Table 33: Packet: Communication Channel Mailbox State Definition

Note: The owner of a mailbox is allowed to start a transfer but he has to ensure the mailbox is empty before starting it.

Evaluation of the actual mailbox state can be done by a simple XOR relation of the host and netX handshake registers and masking out the corresponding command flags:

```
if (0 == ((usHostFlags ^ usNetxFlags) & HCF_SEND_MAILBOX_CMD))
   /* Mailbox empty */
else
   /* Mailbox full */
```

Default Packet Handling:

Transmitter Handling	Receiver Handling
▪ Check if the mailbox is empty	▪ Check if the mailbox is full
▪ Copy packet into the mailbox	▪ Copy packet from the mailbox to a local buffer
▪ Toggle of the corresponding command flag (e.g. HCF_SEND_MBX_CMD / NCF_RECV_MBX_CMD).	▪ Toggle of the corresponding acknowledge flag (e.g. HCF_RECEIVE_MBX_ACK / NCF_SEND_MBX_ACK).

Example: Host sending a packet to the device via the *Send Mailbox*

<table>
<tr><th>Step</th><th colspan="2">Function / Flags</th><th>Mailbox State</th><th>Description</th></tr>
<tr><td rowspan="4">1</td><td colspan="4">Initial state - Host checks if Send Mailbox empty (Flags Equal: both 1 or both 0)</td></tr>
<tr><td>HCF_SEND_MBX_CMD</td><td>NCF_SEND_MBX_ACK</td><td rowspan="3">EMPTY</td><td rowspan="3">Access allowed by host.</td></tr>
<tr><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td></tr>
<tr><td rowspan="4">2</td><td colspan="4">Host - Send Mailbox Handling
▪ Host copies a packet to the Send Mailbox
▪ Host toggles the HCF_SEND_MBX_CMD (1→0 or 0→1)</td></tr>
<tr><td>HCF_SEND_MBX_CMD</td><td>NCF_SEND_MBX_ACK</td><td rowspan="3">FULL</td><td rowspan="3">Access switched to netX.</td></tr>
<tr><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td></tr>
<tr><td rowspan="4">3</td><td colspan="4">netX - Send Mailbox Handling
▪ netX firmware gets a handshake flag change interrupt
▪ Firmware checks for Send Mailbox full (Flags Not Equal)
▪ Firmware copies the packet from the Send Mailbox to internal RAM
▪ Firmware acknowledges the received packet by toggling NCF_SEND_MBX_ACK (0→1 or 1→0)</td></tr>
<tr><td>HCF_SEND_MBX_CMD</td><td>NCF_SEND_MBX_ACK</td><td rowspan="3">EMPTY</td><td rowspan="3">Mailbox is empty; access is switched back to host.</td></tr>
<tr><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td></tr>
<tr><td colspan="5">Back to Step 1</td></tr>
</table>

Table 34: Packet: Send Mailbox Example

Example: netX device sends a packet to the host via the *Receive Mailbox*

<table>
<tr><th>Step</th><th colspan="2">Function / Flags</th><th>Mailbox State</th><th>Description</th></tr>
<tr><td rowspan="4">1</td><td colspan="4">Initial state - netX checks if the Receive Mailbox is empty (Flags Equal: both 1 or both 0)</td></tr>
<tr><td>NCF_RECV_MBX_CMD</td><td>HCF_RECV_MBX_ACK</td><td rowspan="3">EMPTY</td><td rowspan="3">Access allowed by netX.</td></tr>
<tr><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td></tr>
<tr><td rowspan="4">2</td><td colspan="4">netX - Receive Mailbox Handling
▪ netX copies the packet into the Receive Mailbox
▪ netX toggles NCF_RECV_MBX_CMD (1→0 or 0→1)</td></tr>
<tr><td>NCF_RECV_MBX_CMD</td><td>HCF_RECV_MBX_ACK</td><td rowspan="3">FULL</td><td rowspan="3">Access switched to host.</td></tr>
<tr><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td></tr>
<tr><td rowspan="4">3</td><td colspan="4">Host - Receive Mailbox Handling
▪ Host cyclically checks for Receive Mailbox full (Flags Not Equal)
▪ Host copies the packet from the mailbox to internal RAM
▪ Host acknowledges the received packet by toggling HCF_RECEIVE_MBX_ACK (0→1 or 1→0)</td></tr>
<tr><td>NCF_RECV_MBX_CMD</td><td>HCF_RECV_MBX_ACK</td><td rowspan="3">EMPTY</td><td rowspan="3">Mailbox is empty; access is switched back to netX.</td></tr>
<tr><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td></tr>
<tr><td colspan="5">Back to step 1</td></tr>
</table>

Table 35: Packet: Receive Mailbox Example

Note: The system mailbox works in the same way, with the difference that the flags are called differently and located in a different handshake register.

Send Mailbox flags → **HSF**_SEND_MBX_CMD / **NSF**_SEND_MBX_ACK

Receive Mailbox Flags → **NSF**_RECV_MBX_CMD / **HSF**_RECV_MBX_ACK

4.2 Cyclic Data Transfer via Input/Output Data Areas

Cyclic process data of a fieldbus protocol stack (input and output data) is exchanged between members of a fieldbus network and stored in separate input and output areas. Each area has its own dedicated synchronization flags.

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

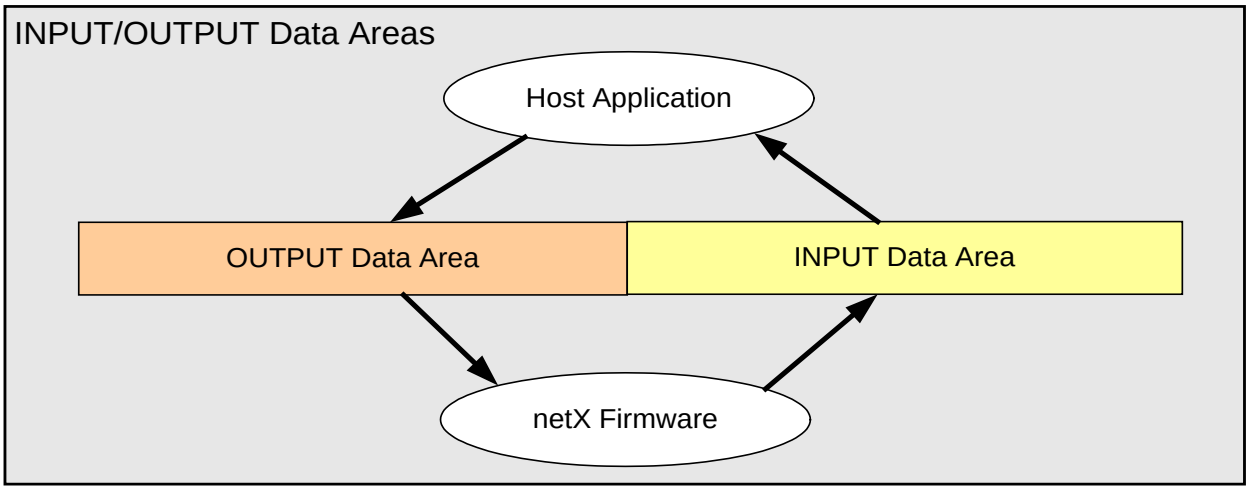


Figure 12: I/Os: Input / Output Data Areas

When a transfer takes place, either the host or the netX temporarily “owns” the input/output data area and has exclusive read/write access to it. This guarantees data consistency over the data areas while the synchronization can be performed independently for each of the areas.

The fieldbus protocol stacks also support different data exchange modes, which define the initiator of a data transfer and how I/O data from the fieldbus network are handled inside the netX firmware respectively a fieldbus protocol stack.

4.2.1 I/O Data Exchange Modes

I/O data exchange modes are defined to allow a data flow control of I/O data between a fieldbus network and the host application and the use of a simple synchronization flag mechanism to ensure data consistency during the transfer.

This might be necessary because the speed of a host system and the network can be different and a fieldbus protocol may offer additional options to exchange data with the fieldbus system. Both must be considered in the host application and may influence the data processing and program flow in the application.

General overview

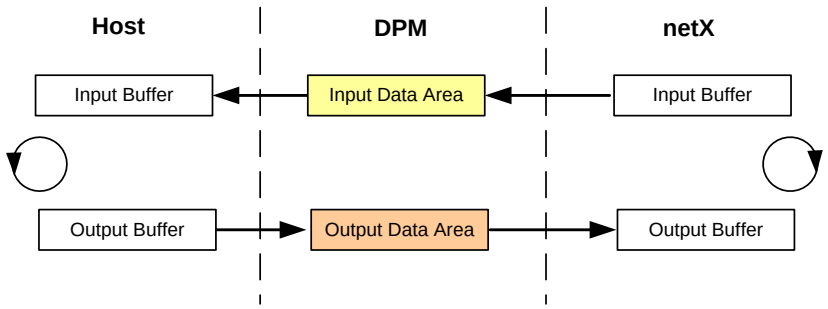


Figure 13: I/Os: I/O Data Exchange

A data exchange mode is described by two attributes. The first is the initiator of the transfer, described as the controlled mode: *Host Controlled* or *Device Controlled*. While the second attribute defines the transfer mode of the I/O data: *Buffered* or *Synchron*.

- Controlled Mode Host Controlled or Device Controlled
- Transfer Mode Buffered or Synchron

Both attributes are necessary to cover the possibilities of a data transfer and to handle it correctly.

- **Host Controlled Mode**

  In *Host Controlled Mode*, the host application initially has access to the I/O data areas in the DPM and can be the first to read and write data before starting a transfer between the netX firmware and the DPM.

- **Device Controlled Mode**

  Device Controlled Mode defines the device (netX / fieldbus protocol stack) as the initiator of a data transfer. In this case the device (netX) initially has access to the I/O data areas inside the DPM and will activate the data transfer between the host and the DPM.

- **Buffered Transfer**

  Defines that I/O data is automatically transferred between the fieldbus system and internal buffer inside the netX firmware. A transfer between the internal buffer and the DPM depends on the *Controlled Mode*.

- **Synchronous Transfer**

  Defines that I/O data is not buffered inside the device (netX / fieldbus protocol) instead they are exchanged (read/written) directly with the I/O areas in the DPM. The *Controlled Mode* defines the initiator of a transfer.

The following data exchange modes are defined

Data Exchange Modes	Controlled by	Consistency	Supported by
Buffered Host Controlled (**default**)	Host	Yes	Master & Slave Firmware
Buffered Device Controlled	netX	Yes	Slave Firmware
Uncontrolled Mode	none	No	currently not supported

Table 36: I/Os: Process Data Exchange Modes

Note: Data exchange modes are a part of the fieldbus protocol configuration which may allow configuring different exchange modes for the input / output areas but not all combinations are reasonable.

Meaningful configurations are:
Output Area → Buffer Host Controlled
Input Area → Buffered Host or Device Controlled / Buffered Device Controlled

4.2.1.1 Buffered Host Controlled Mode

The *Buffered Host Controlled Mode* is the default mode for the transfer of input/output data between the fieldbus system and the host application and available on master and slave devices.

In *Buffered* mode, the protocol stack uses internal buffers to store/send cyclic fieldbus data. The exchange between the local buffers and the DPM is only done if the host application requests an update. This allows a complete asynchronous handling of I/O data, on both sides, the fieldbus and the host application.

Note: Network cycle and task cycle of the host application are not synchronized but input and output data transfer is consistent.

General Definitions

- Host has access to the input and output data area
- Host has to request an update of the input/output area
- A request switches data access to the netX device
- The netX exchanges the content of the requested data with the internal buffer
- The netX confirms the data exchange by switching back data access to the host

Sequence of the process data exchange:

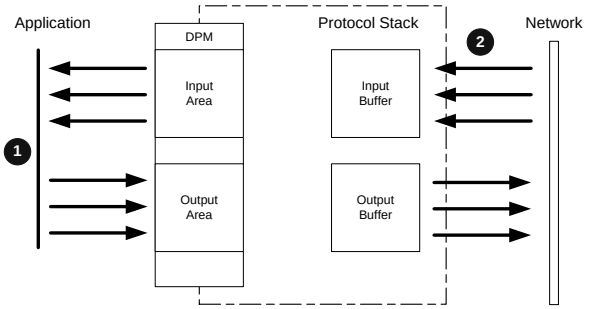
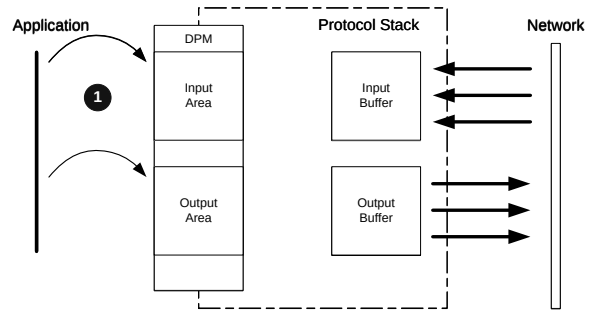
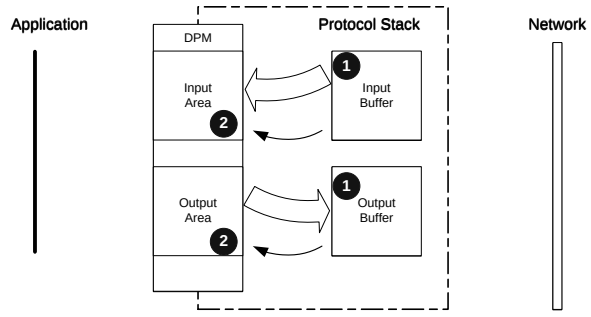
Step	Action	Figure
1	(1) The application (host) owns access rights to the input/output data areas in the DPM and is able to read and write the process data areas. (2) For each valid bus cycle, the protocol stack exchanges the process data with the internal buffers.	
2	(1) The application requests an update of the process data areas inside the DPM.	
3	(1) The protocol stack exchanges the data of the requested DPM area (input/output) with the internal buffer content. (2) The update process will be acknowledged and access to the DPM areas is switch back to the host application. Goto step 1	

Table 37: I/Os: Buffered Host Controlled Mode

Considerations

- If the host application is faster than the network cycle, it might be possible that data in the output buffers is overwritten without ever being sent to the network. As for the other direction, the host application may read the same input values over several read cycles or the protocol stack may block further input updates by delaying the acknowledgement.
- If the host application is slower than the network cycle, the protocol stack might overwrite the input buffer with new data received from the network, without it ever being received by the host application. The output data on the network will be the same over several network cycles.

Note: In case of a network fault (e.g. network cable disconnect) the protocol stack clears *NCF_COMMUNICATING* flag in the netX communication flags `usNetXFlags`, see section *Communication Channel - Handshake Register* on page 27.
Input data should not be evaluated anymore by the host application while output data can be still exchanged with the protocol stack.

4.2.1.2 Buffered Device Controlled Mode

The *Buffered Device Controlled Mode* defines the netX device (protocol stack) as the initiator of a data exchange between the protocol stack and the host application.

Like in the *Buffered Host Controlled Mode,* the *Buffered* attribute defines the data handling inside the fieldbus protocol stack, where cyclic fieldbus input/output data is stored to/send via internal buffers. The exchange of the local buffers with the corresponding DPM areas is activated by the protocol stack (netX device), which also signals the host application when new data is received from or send to the fieldbus network.

Note: The *Device Controlled Mode* is only offered for the input data of slave devices!

The general handling in a host application for data areas transferred in *Buffered Device Controlled Mode*, corresponds to the handling in *Buffered Host Controlled Mode*, except the initiator of a transfer is the netX device and therefore the evaluation of the synchronization flags is inverted.

General Definitions

- netX device has access to the input and output data area
- netX (protocol stack) signals new data received by the field bus network and written to the input data area of the DPM
- Access to the DPM input area is switched to the host
- The host copies the input data to a local buffer
- Host confirms the data processing
- Access to the DPM area is switched back to the netX

Sequence of the process data exchange:

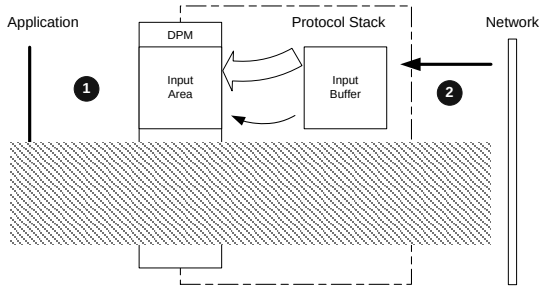
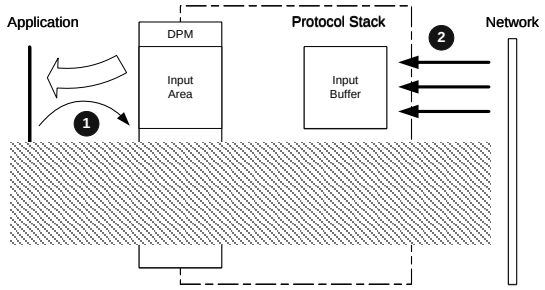
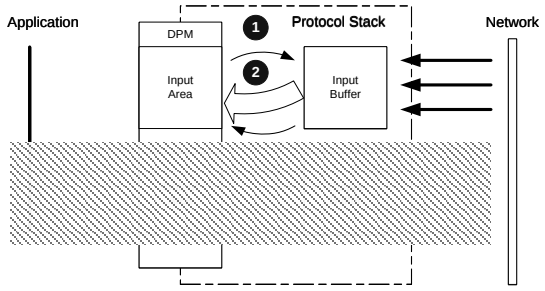
Step	Action	Picture
1	(1) The application (host) checks if access rights to the input area are available. (2) The protocol stack takes the input data from the first valid bus cycle, inserts the data into the internal buffer and the input DPM area and signals the new data available. This switches the access right to the host.	
2	(1) The application (host) recognizes the new data in the input area. Takes the data from the input area and confirms the processing. Access right to the DPM area is switch back to netX. (2) While the host has access to the input DPM area, the protocol stack will store any further received field bus data in the internal input buffer.	
3	(1) The protocol stack recognizes the acknowledgement from the host. (2) If new data from the field bus are available, Data is copied to the DPM area and host will be signaled. Access right is switch to the host. Goto step 2	

Table 38: I/Os: Buffered Device Controlled Mode

Considerations:

- If the host application is slower than the network cycle, the protocol stack buffers further network data and might overwrite the input buffer several times, resulting in data never send to the host application.
- A protocol stack counts the number of incomplete updates of the input data area in DPM (see *Common Status Block* - `bErrorPDInCnt`) where access was switched to the host application. This counter can be evaluated by the host application to find out if bus cycles have been missed.

Note: In case of a network fault (e.g. network cable disconnect) the protocol stack clears the *NCF_COMMUNICATING* flag in the netX communication flags `usNetXFlags` (see section *Communication Channel - Handshake Register* on page 27).
Input data should not be evaluated anymore by the host application while output data can be still exchanged with the protocol stack.

4.2.2 I/O Data Area Access Synchronization

Access to the I/O process data areas in the DPM is synchronized by dedicated handshake flags in the handshake register of the corresponding communication channel.

- I/O Process Data INPUT Flag Definitions
  NCF_PD0_IN_CMD / HCF_PD0_IN_ACK
- I/O Process Data OUTPUT Flag Definitions
  HCF_PD0_OUT_CMD / NCF_PD0_OUT_ACK

Attention: In the rcX_User.h file there is only **ONE** definition for the HCF_PD0_ and NCF_PD0_ handshake bits per direction _IN / _OUT available. Therefore the interpretation of _**CMD** / _**ACK** will change depending on the *Controlled Mode* by using the same definition. The correct functional interpretation will be given in brackets **(.....)**!

Handshake flag definition corresponding to the *Controlled Mode*:

- Input Data Area Flags

Controlled Mode	**Host**	**netX**
Host Controlled	HCF_PD0_IN_ACK**(_CMD)**	NCF_PDO_IN_CMD**(_ACK)**
Device Controlled	HCF_PD0_IN_ACK	NCF_PDO_IN_CMD

- Output Data Area Flags

Controlled Mode	**Host**	**netX**
Host Controlled	HCF_PDO_OUT_CMD	NCF_PDO_OUT_ACK
Device Controlled	HCF_PDO_OUT_CMD**(_ACK)**	NCF_PDO_OUT_ACK(_**CMD**)

Depending on the I/O data exchange mode, the handling of the handshake flags differs for the host and the netX. The exchange mode is part of the fieldbus protocol stack configuration and separated for the input and output data area. The current settings can be retrieved from the *Common Status Block* of the communication channel.

Handshake Mode Information - *Common Status Block*

Input Data Area			
Offset	**Type**	**Name**	**Description**
0x0020	uint8_t	bPDInHskMode	Handshake Mode Input Process Data Handshake Mode, see page 108.
0x0021	uint8_t	bPDInSource	Handshake Mode Reserved, set to zero
0x002D	uint8_t	bErrorPDInCnt	Number of input process data handshake errors
Output Data Area			
Offset	**Type**	**Name**	**Description**
0x0022	uint8_t	bPDOutHskMode	Handshake Mode Output Process Data Handshake Mode, see page 108.
0x0023	uint8_t	bPDOutSource	Handshake Mode Reserved, set to zero
0x002E	uint8_t	bErrorPDOutCnt	Number of output process data handshake errors

4.2.2.1 Synchronization in Buffered Host Controlled Mode

Example: Read **INPUT** data by the host in ***Buffered Host Controlled Mode***

<table>
<tr><th>Step</th><th colspan="2">Flags</th><th>INPUT State</th><th>Description</th></tr>
<tr><td rowspan="4">1</td><td colspan="4">Initial state - Host has access to the input data area</td></tr>
<tr><td>HCF_PD0_IN_ACK(_CMD)</td><td>NCF_PD0_IN_CMD(_ACK)</td><td rowspan="3">FREE</td><td rowspan="3">Access allowed by host.</td></tr>
<tr><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td></tr>
<tr><td rowspan="4">2</td><td colspan="4">Host - Input Data Handling
▪ Host checks for access right for the input area (HCF_PD0_IN_ACK equal to NCF_PD0_IN_CMD)
▪ Host copies the input data from the input data area of the DPM to a local buffer
▪ Host requests the device to update input data area in the DPM by toggling HCF_PD0_IN_ACK (1→0 or 0→1)</td></tr>
<tr><td>HCF_PD0_IN_ACK(_CMD)</td><td>NCF_PD0_IN_CMD(_ACK)</td><td rowspan="3">BUSY</td><td rowspan="3">Access switched to netX which should update the data.</td></tr>
<tr><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td></tr>
<tr><td rowspan="4">3</td><td colspan="4">netX - Input Data Handling
▪ netX gets an interrupt signaling a state change in the handshake flags
▪ netX checks the state of HCF_PD0_IN_ACK not equal NCF_PD0_IN_CMD (request from host)
▪ netX has access rights and copies the actual local input data into the input data area of the DPM and toggles NCF_PD0_IN_CMD (1→0 or 0→1)</td></tr>
<tr><td>NCF_PD0_IN_ACK(_CMD)</td><td>NCF_PD0_IN_CMD(_ACK)</td><td rowspan="3">DONE</td><td rowspan="3">Device has updated the INPUT data; access is switched back to host.</td></tr>
<tr><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td></tr>
<tr><td colspan="5">Back to Step 2</td></tr>
</table>

Table 39: I/Os: Synchronization in Buffered Host Controlled Mode - Input

Example: Write **OUTPUT** data by the host in ***Buffered Host Controlled Mode***

Step	Flags		OUTPUT State	Description
1	**Initial state - Host has access to the output data area**			
	HCF_PD0_OUT_CMD	NCF_PD0_OUT_ACK	FREE	Access allowed by host.
	1	1		
	0	0		
2	**Host - Output Data Handling** ▪ Host checks for access right to output area (HCF_PD0_OUT_CMD equal to NCF_PD0_OUT_ACK) ▪ Host copies its local data to the output data area of the DPM ▪ Host signals to the device that new output data is available in the DPM output data area by toggling HCF_PD0_OUT_CMD (1→0 or 0→1)			
	HCF_PD0_OUT_CMD	NCF_PD0_OUT_ACK	DATA available	Access switched to netX which should take the data.
	0	1		
	1	0		
3	**netX - Output Handling** ▪ netX gets an interrupt signaling a state change in the handshake flags ▪ netX checks the state of HCF_PD0_OUT_CMD not equal NCF_PD0_OUT_ACK (request from host) ▪ netX copies the data from the DPM output data area into the local output buffer and toggles NCF_PD0_OUT_ACK (1→0 or 0→1) (The Output data will be sent to the network with the next field bus cycle)			
	HCF_PD0_OUT_CMD	NCF_PD0_OUT_ACK	DONE	netX has taken the OUTPUT data; access is switched back to host.
	0	**0**		
	1	**1**		
Back to Step 2				

Table 40: I/Os: Synchronization in Buffered Host Controlled Mode - Output

4.2.2.2 Synchronization in Buffered Device Controlled Mode

In *Device Controlled Mode* the initiator of the transfer changes from HOST to netX. While in *Host Controlled Mode* the transfer for the INPUT data area was started by toggling *HCF_PD0_IN_ACK*, it is now used to acknowledge a transfer. In Device Controlled Mode the transfer starts by toggling *NCF_PD0_IN_CMD*.

Example: Read **INPUT** data by the host in ***Buffered Device Controlled Mode***

<table>
<tr><th>Step</th><th colspan="2">Flags</th><th>INPUT State</th><th>Description</th></tr>
<tr><td rowspan="4">1</td><td colspan="4">Initial state - netX has access to the input data area</td></tr>
<tr><td>NCF_PD0_IN_CMD</td><td>HCF_PD0_IN_ACK</td><td rowspan="3">FREE</td><td rowspan="3">Access allowed by netX.</td></tr>
<tr><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td></tr>
<tr><td rowspan="4">2</td><td colspan="4">netX - Input Handling
▪ netX gets new input data from the field bus network and stores the data in the local input buffer
▪ netX checks if handshake flags NCF_PD0_IN_CMD equal to HCF_PD0_IN_ACK. If not <code>bErrorPDInCnt</code> is incremented and new data will not be signaled to the host
▪ netX copies the content of the input buffer to the input data area of the DPM and toggles NCF_PD0_IN_CMD (1→0 or 0→1)</td></tr>
<tr><td>NCF_PD0_IN_CMD</td><td>HCF_PD0_IN_ACK</td><td rowspan="3">DATA available</td><td rowspan="3">Access switched to the host, which should take the data.</td></tr>
<tr><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td></tr>
<tr><td rowspan="4">3</td><td colspan="4">Host - Input Handling
▪ Host checks if handshake flag NCF_PD0_IN_CMD not equal to HCF_PD0_IN_ACK
▪ Host copies the input data area from the DPM to a local buffer and toggles HCF_PD0_IN_ACK (1→0 or 0→1)</td></tr>
<tr><td>NCF_PD0_IN_CMD</td><td>HCF_PD0_IN_ACK</td><td rowspan="3">DONE</td><td rowspan="3">Host has taken the INPUT data; access is switched back to device.</td></tr>
<tr><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td></tr>
<tr><td colspan="5">Back to Step 2</td></tr>
</table>

Table 41: I/Os: Synchronization in Buffered Device Controlled Mode - Input

Note: Buffered Device Controlled Mode for OUTPUT data is usually not supported by the protocol stacks.

4.3 Change of State Mechanism (COS)

A communication channel has more options and commands than bits (flags) available in the handshake flag register. Therefore a so called *Change of State (COS)* mechanism is defined which extends the direct usable handshake flags by an additional 32 Bit value, including the opportunity to generate interrupts when the value in the additional 32 Bits is changed.

In other words *Change of State* (COS) registers are basically extensions to the handshake registers.

Furthermore, the COS mechanism defines an acknowledgement of state changes before another state change will be signaled. The mechanism is direction oriented and distinguishes between state changes from the host application and from the device:

- **Communication COS Handling**

  Describes the handling of additional device states located in the *Communication COS Register* in the *Common Status Block* (see section 5.4.3).

- **Application COS Handling**

  Describes the handling of additional application states located in the *Application COS Registe*r in the *Common Control Block* (see section 5.4.2).

The *Change of State Command* and *Change of State Acknowledge* flags are located in the corresponding handshake registers of the communication channel (see section 3.3).

The COS handling also defines an ***Enable Flag Handling*** for signals inside the COS registers. This simplifies handling of COS signals, because the receiver of the COS information is able to clearly detect which signals are currently active and which not.

- **Enable Flag Handling**

  Is used to mark signals, inside a COS register, as active (*enabled*) or not active (*disabled*).

4.3.1 Communication COS Handling

The COS handling is used to signal additional states to the host application.

Sequence of the *Communication COS* handling

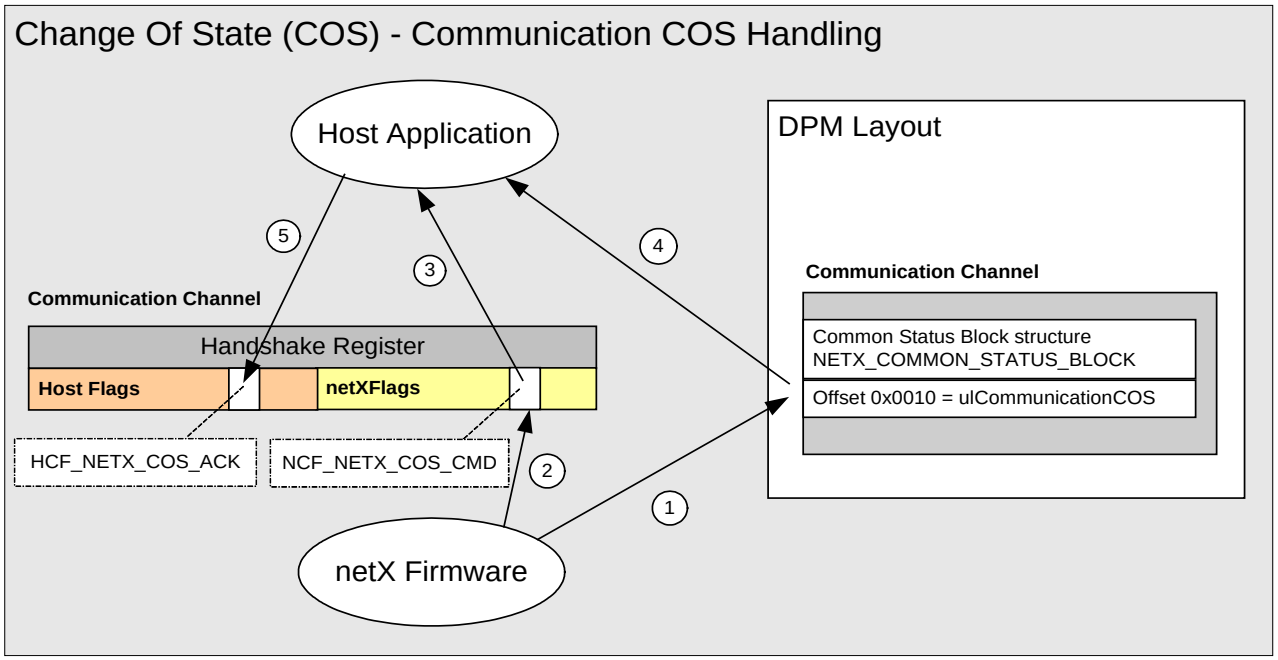


Figure 14: Change of State Mechanism (COS): Communication COS Handling

Step	Description
1	netX Firmware checks if access to the COS information is allowed (COS flags are equal, NCF_NETX_COS_CMD equal to NCF_NETX_COS_ACK) and updates the COS information in *ulCommunicationCOS.*
2	netX firmware toggles the NCF_NETX_COS_CMD which signals new information available to the host.
3	Host cyclically checks if COS flags are not equal (NCF_NETX_COS_CMD unequal to NCF_NETX_COS_ACK) and if so, new information is available and can be read by the host.
4	Host has access to *ulCommunicationCOS* and reads the value.
5	Host application acknowledges COS state change by toggling HCF_NETX_COS_ACK.

Table 42: Change of State Mechanism (COS): Communication COS Handling

4.3.2 Application COS Handling

Used to signal additional states / commands to a firmware.

Sequence of the *Application COS* handling

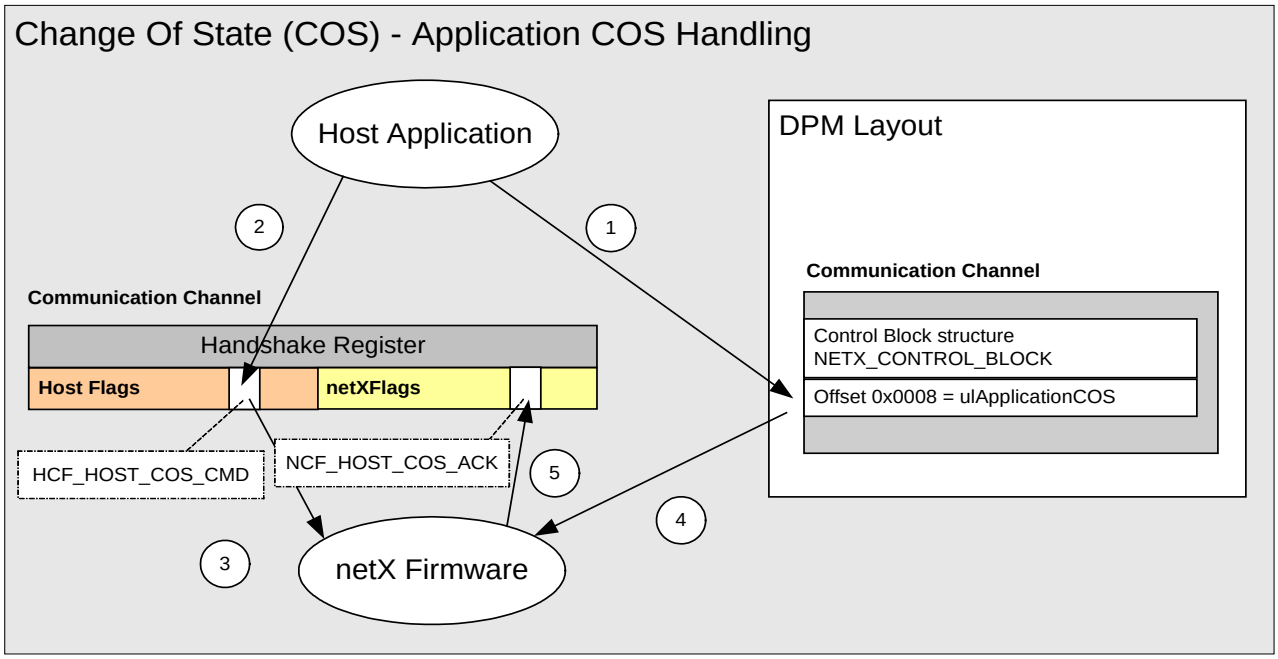


Figure 15: Change of State Mechanism (COS): Application COS Handling

Step	Description
1	Host checks if access to the COS information is allowed (COS flags are equal, HCF_HOST_COS_CMD equal to NCF_HOST_COS_ACK) and updates the COS information in *ulApplicationCOS.*
2	Host toggles HCF_HOST_COS_CMD which signals new information available.
3	Firmware gets an interrupt if the COS flags are changed and checks the COS flags for inequality (HCF_HOST_COS_CMD unequal to NCF_HOST_COS_ACK) and if so, new information is available and can be read by the firmware
4	netX firmware reads *ulApplicationCOS*.
5	Firmware acknowledges COS state change by toggling NCF_HOST_COS_ACK.

Table 43: Change of State Mechanism (COS): Application COS Handling

4.3.3 Enable Flag Handling

Enable flags are used in the *Communication COS* and *Application COS registers* to selectively activate and deactivate functions, without interfering with each other and to permit an evaluation of currently activated functions. It also simplifies, for both the initiator and receiver of a command, to identify actual active commands.

Note: If a command, equipped with an enable flag, should be signaled in the COS register, the enable flag must be set to 1 (enabled) before the state in the corresponding command flag will be evaluated by the receiver of the COS command. And the enable flag must be set to 0 (disabled) after the receiver has acknowledged the recognition of the command and before a new COS command is initiated.

Example: Signal *RCX_APP_COS_BUS_ON,* located in the *Application COS* flags

The command consists of two flags RCX_APP_COS_BUS_ON_ENABLE and RCX_APP_COS_BUS_ON. It can be used to switch the field bus communication ON and OFF and the command is executed by the fieldbus protocol stack.

Signal State	Description
RCX_APP_COS_BUS_ON_ENABLE	0 = command is not active 1 = command is active
RCX_APP_COS_BUS_ON	0 = switch OFF fieldbus communication 1 = switch ON fieldbus communication

Default COS Handling

<table>
<tr><th>Step</th><th colspan="2">Flags</th><th>COS State</th><th>Description</th></tr>
<tr><td rowspan="4">1</td><td colspan="4">Initial state - Host has no active COS state</td></tr>
<tr><td>HCF_HOST_COS_CMD</td><td>NCF_HOST_COS_ACK</td><td rowspan="3">Free</td><td rowspan="3">COS signal not active, host can signal a new state.</td></tr>
<tr><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td></tr>
<tr><td rowspan="4">2</td><td colspan="4">Host - COS Handling - Bus ON command
▪ Host checks if COS handling is possible (HCF_HOST_COS_CMD equal to NCF_HOST_COS_ACK)
▪ Host sets the command flag and the enable flag of the new state to signal
(e.g. RCX_APP_COS_BUS_ON = 1 / RCX_APP_COS_BUS_ON_ENABLE = 1)
▪ Host signals the new state to the device by toggling HCF_HOST_COS_CMD (1→0 or 0→1)</td></tr>
<tr><td>HCF_HOST_COS_CMD</td><td>NCF_HOST_COS_ACK</td><td rowspan="3">BUSY</td><td rowspan="3">Access to COS state is switched to device.</td></tr>
<tr><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td></tr>
<tr><td rowspan="4">3</td><td colspan="4">netX - COS Handling
▪ netX gets an interrupt signaling a state change in the handshake registers
▪ netX checks the state of HCF_HOST_COS_CMD not equal NCF_HOST_COS_ACK
(new COS state from the host/request from host)
▪ netX reads and evaluates the ulApplicationCOS register
(e.g. RCX_APP_COS_BUS_ON == 1 && RCX_APP_COS_BUS_ON_ENABLE == 1)
▪ netX firmware processes the Bus ON command
▪ netX acknowledges the COS handling by toggling NCF_HOST_COS_ACK (0→1 or 1→0)</td></tr>
<tr><td>HCF_HOST_COS_CMD</td><td>NCF_HOST_COS_ACK</td><td rowspan="3">DONE</td><td rowspan="3">Device has processed the COS flags and access is switched back to host.</td></tr>
<tr><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td></tr>
<tr><td>4</td><td colspan="4">Host - COS Handling - Deactivation
▪ Host recognizes a COS acknowledgment (HCF_HOST_COS_CMD equal to NCF_HOST_COS_ACK)
▪ Host disables the COS command by setting the enable flag RCX_APP_COS_BUS_ON_ENABLE = 0</td></tr>
<tr><td colspan="5">Goto Step 2</td></tr>
</table>

Table 44: Change of State Mechanism (COS): Enable Flag Handling

Note: The COS enable flag handling is identical for the host and the netX commands.

5 DPM Definitions / Mapping and Content

This section describes the DPM mapping in more detail. The presented DPM structures and definitions are provided by two C header files.

- **rcx_User.h**

  This C header file contains all parts necessary to work with the DPM by using symbolic names. It provides definitions of the DPM structure, data blocks and all global definitions offered by the DPM.

- **rcx_Public.h**

  Provides the "*rcX Public Packet*" function definitions

Note: All structures and definitions used in the following chapters can be found in two header files.

5.1 DPM Mapping

In the default memory layout, each channel has a fixed structure and a fixed length. The following chapter describes the structures and elements offered by the DPM layout.

Dual-Port Memory Mapping: *Default Mapping (64KByte)*

System Channel (0x000 … 0x01FF)			
Name	**Offset**	**Size**	**Description**
System Information	0x0000	48 Bytes	System Information Area
Channel Information	0x0030	128 Bytes	Channel Information Area
Reserved	0x00B0	8 Bytes	Reserved
System Control	0x00B8	8 Bytes	System Control and Commands
System Status	0x00C0	64 Bytes	System Status Information
System Mailboxes	0x0100	256 Bytes	System Send / Receive Mailbox
Handshake Channel (0x0200 … 0x02FF)			
Name	**Offset**	**Size**	**Description**
Handshake Register	0x0200	64 Bytes	Handshake Registers Area
Reserved	0x0240	192 Bytes	Reserved

Communication Channel 0 (0x0300 … 0x3FFF)			
Name	**Offset**	**Size**	**Description**
Reserved	0x0300	8 Bytes	Reserved
Common Control Block	0x0308	8 Bytes	Common Control Register
Common Status Block	0x0310	64 Bytes	Common Protocol Stack Status Information
Extended Status Block	0x0350	432 Bytes	Network Specific Information
Send Mailbox	0x0500	1600 Bytes	Send Mailbox for Non-Cyclic Data Transfer
Receive Mailbox	0x0B40	1600 Bytes	Receive Mailbox for Non-Cyclic Data Transfer
Output Data Image 1	0x1180	64 Bytes	High Priority Output Process Data Image
Input Data Image 1	0x11C0	64 Bytes	High Priority Input Process Data Image
Reserved	0x1200	256 Bytes	Reserved for Future Use, Set to Zero
Output Data Image 0	0x1300	5760 Bytes	Cyclic Output Process Data Image
Input Data Image 0	0x2980	5760 Bytes	Cyclic Input Process Data Image
Communication Channel 1..3 (0x4000 …)			
Name	**Offset**	**Size**	**Description**
Comm. Channel 1	0x4000	15616 Bytes	Structure see Communication Channel 0
Comm. Channel 2	0x7D00	15616 Bytes	Structure see Communication Channel 0
Comm. Channel 3	0xBA00	15616 Bytes	Structure see Communication Channel 0
Application Channel 0..1			
Name	**Offset**	**Size**	**Description**
App. Channel 0/1	unknown	unknown	Application channel not defined yet

Table 45: DPM Mapping: Default Mapping

- Default DPM size of 64 KByte (65536 Byte) results from the sizes of the *System Channel*, *Handshake Channel* and 4 *Communication Channels*.
- The default communication channel size is 15616 Byte.

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

Note: The default mapping is also available with 16 KByte DPM size, but than with only one communication channel.

Dual-Port Memory Mapping: *8Kbyte Mapping*

System Channel (0x000 … 0x01FF)			
Name	**Offset**	**Size**	**Description**
System Information	0x0000	48 Bytes	System Information Area
Channel Information	0x0030	128 Bytes	Mapping Information
Reserved	0x00B0	8 Bytes	Reserved
System Control Block	0x00B8	8 Bytes	System Control and Commands
System Status Block	0x00C0	64 Bytes	System Status Information
System Mailboxes	0x0100	256 Bytes	System Send / Receive Mailbox
Handshake Channel (0x0200 … 0x02FF)			
Name	**Offset**	**Size**	**Description**
Handshake Register	0x0200	64 Bytes	Cumulated Handshake Registers
Reserved	0x0240	192 Bytes	Reserved
Communication Channel 0 (0x0300 … 0x1FFF)			
Name	**Offset**	**Size**	**Description**
Reserved	0x0300	8 Bytes	Reserved
Common Control Block	0x0308	8 Bytes	Common Control Register
Common Status Block	0x0310	64 Bytes	Common Protocol Stack Status Information
Extended Status Block	0x0350	432 Bytes	Network Specific Information
Send Mailbox	0x0500	1600 Bytes	Send Mailbox for Non-Cyclic Data Transfer
Receive Mailbox	0x0B40	1600 Bytes	Receive Mailbox for Non-Cyclic Data Transfer
Output Data Image 1	0x1180	64 Bytes	High Priority Cyclic Output Process Data Image
Input Data Image 1	0x11C0	64 Bytes	High Priority Cyclic Input Process Data Image
Reserved	0x1200	256 Bytes	Reserved for Future Use, Set to Zero
Output Data Image 0	0x1300	1536 Bytes	Cyclic Output Process Data Image
Input Data Image 0	0x1900	1536 Bytes	Cyclic Input Process Data Image
Reserved Area (0x1F00 … 0x1FFF)			
Name	**Offset**	**Size**	**Description**
Reserved	0x1F00	256 Bytes	Reserved for Future Use, set to Zero

Table 46: DPM Mapping: 8 Kbyte Mapping

5.2 System Channel

The System Channel is the first channel in the dual-port. It holds information about the system itself (netX firmware/netX operating system) and provides a mailbox transfer mechanism for system related messages or packets.

Note: Offsets are given relative to the start offset of the *System Channel* start address.

Structure of the *System Channel*

System Channel: *NETX_SYSTEM_CHANNEL*			**DPM Start Offset = 0x0000**
Offset	**Type**	**Name**	**Description**
0x0000	Structure	tSystemInfo	System Information Block Identifies netX Dual-Port Memory (see section 5.2.1)
0x0030	Structure	atChannelInfo[8]	Channel Information Block Contains Configuration Information about available Communication and Application Channel Blocks (see section 5.2.2)
0x00B0	Structure	tSysHandshake	System Handshake Block (not used, set to zero)
0x00B4	uint8_t	bReserved[4]	Reserved
0x00B8	Structure	tSystemControl	System Control Block System Control and Commands (see section 5.2.4)
0x00C0	Structure	tSystemState	System Status Block System Status Information (see section 5.2.5)
0x0100 **0x0180**	Structure	tSystemSendMailbox tSystemRecvMailbox	System Mailboxes System Send and Receive Packet Mailbox Area, always located at the end of the System Block (see section 5.2.6)

Table 47: System Channel: System Channel Structure

System Channel Structure Reference

```
typedef struct NETX_SYSTEM_CHANNELtag
{
  NETX_SYSTEM_INFO_BLOCK            tSystemInfo;
  NETX_CHANNEL_INFO_BLOCK           atChannelInfo[8];
  NETX_HANDSHAKE_CELL               tSysHandshake;
  uint8_t                           abReserved[4];
  NETX_SYSTEM_CONTROL_BLOCK         tSystemControl;
  NETX_SYSTEM_STATUS_BLOCK          tSystemState;
  NETX_SYSTEM_SEND_MAILBOX          tSystemSendMailbox;
  NETX_SYSTEM_RECV_MAILBOX          tSystemRecvMailbox;
} NETX_SYSTEM_CHANNEL;
```

5.2.1 System Information Block

The *System Information Block* helps to identify the netX dual-port memory and holds general information about the netX device including a cookie, DPM length as well as information regarding the firmware running on the netX.

Structure of the *System Information Block*

System Information Block: *NETX_SYSTEM_INFO_BLOCK*			
Offset	**Type**	**Name**	**Description**
0x0000	uint8_t	abCookie[4]	netX DPM Identification (start of DPM) ASCII characters: '**netX**' = firmware is running
0x0004	uint32_t	ulDpmTotalSize	DPM Size in bytes (see page 72)
0x0008	uint32_t	ulDeviceNumber	Device Number (see page 73)
0x000C	uint32_t	ulSerialNumber	Serial Number (see page 73)
0x0010	uint16_t	ausHwOptions[4]	Hardware Assembly Options (see page 73)
0x0018	uint16_t	usManufacturer	Manufacturer Code (see page 74)
0x001A	uint16_t	usProductionDate	Production Date (see page 75)
0x001C	uint32_t	ulLicenseFlags1	License Code - Flags 1 (see page 76)
0x0020	uint32_t	ulLicenseFlags2	License Code - Flags 2 (see page 76)
0x0024	uint16_t	usNetxLicenseID	netX License Identification (see page 76)
0x0026	uint16_t	usNetxLicenseFlags	netX License Flags (see page 76)
0x0028	uint16_t	usDeviceClass	Device Class (see page 78)
0x002A	uint8_t	bHwRevision	Hardware Revision (see page 80)
0x002B	uint8_t	bHwCompatibility	Hardware Compatibility (see page 80)
0x002C	uint8_t	bDevIdNumber	Hardware Device Identification Number (DIP-switch / Rotary Switch, see page 80)
0x002D	uint8_t	bReserved	Reserved Set to Zero
0x002E … 0x002F	uint16_t	usReserved	Reserved Set to Zero

Table 48: System Channel: System Information Block

System Information Block Structure Reference

```
typedef struct NETX_SYSTEM_INFO_BLOCKtag
{
  uint8_t  abCookie[4];           /* 'netX' cookie                */
  uint32_t ulDpmTotalSize;        /* DPM size (in bytes)          */
  uint32_t ulDeviceNumber;        /* device number                */
  uint32_t ulSerialNumber;        /* serial number                */
  uint16_t ausHwOptions[4];       /* hardware options             */
  uint16_t usManufacturer;        /* manufacturer                 */
  uint16_t usProductionDate;      /* production date              */
  uint32_t ulLicenseFlags1;       /* license flags 1              */
  uint32_t ulLicenseFlags2;       /* license flags 2              */
  uint16_t usNetxLicenseID;       /* license ID                   */
  uint16_t usNetxLicenseFlags;    /* license flags                */
  uint16_t usDeviceClass;         /* device class                 */
  uint8_t  bHwRevision;           /* hardware revision            */
  uint8_t  bHwCompatibility;      /* hardware compatibility       */
  uint8_t  bDevIdNumber;          /* device identification        */
  uint8_t  bReserved;
  uint16_t usReserved;
} NETX_SYSTEM_INFO_BLOCK;
```

netX DPM Identification

The netX DPM Identification *abCookie[4]*, identifies the start of the dual-port memory. It has a length of 4 bytes and is always present if a netX firmware is running.

Note: The cookie is the last element, initialized by the firmware. The firmware ensures, if a valid value is inserted in the cookie, all other elements of the DPM are initialized.

Variable	Value	Definition / Description
abCookie[4]	'**netX**'	netX Firmware cookie
	'**BOOT**'	netX Bootloader cookie
	0x0BAD	Bad memory content (RCX_SYS_BAD_MEMORY_COOKIE), memory is not correctly mapped by the firmware or no firmware running.
	0xFFFF	DPM not available

Table 49: System Channel: netX Identification, netX Cookie

Dual-Port Memory Size

The size field `ulDpmTotalSize` holds the total size of the dual-port memory in bytes (8 Kbyte / 16 KByte / 64 KByte). The usable size may differ because each channel defines an own layout.

If the default memory layout is used, the usable size is 16 Kbyte (see section 5.1).

Device Number

The device number given in *ulDeviceNumber* holds the hardware order or item number of a device. The number is given as a hexadecimal number and follows the following format: xxx.yyyy.zzz.

Example:
A value of 0x499602D2 is translated into the decimal value of 1234567890 which results in an order number of 123.4567.890

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

Serial Number

The serial number given in ulSerialNumber holds the hardware serial number used in conjunction with the device number to uniquely identify a device. It is a 32-bit value given as hexadecimal number.

Example:
A value of 0x00004E21 compares to the decimal number 20021.

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

Hardware Assembly Options (xC Port 0..3)

The assembly option defines the physical hardware interface connected to a netX xC (Communication Controller) port. A netX chip offers up to 4 xC ports and the assembly option is an array of 4 elements (*ausHWOptions[4]*) where each element holds the definition of one xC port (Port 0..3).

The definition is necessary because each fieldbus protocol has specific requirements to the physical interface and only with a correct assembled physical interface the communication on the fieldbus system is possible.

Hardware Assembly Options

Variable: *ausHWOptions*	
Value	**Definition / Description**
0x0000	*RCX_HW_ASSEMBLY_UNDEFINED* The xC port is marked UNDEFINED, if the hardware cannot be determined. This might be the case, if no security memory / device label is found or read access to it failed.
0x0001	*RCX_HW_ASSEMBLY_NOT_AVAILABLE* The xC port is not available (netX50 will show this for xC2 / xC3)
0x0003	*RCX_HW_ASSEMBLY_USED* The xC port is marked used if a port is already used by another component of the netX firmware (e.g. another protocol stack or other functionalities)
0x0010	*RCX_HW_ASSEMBLY_SERIAL* A serial RS232 interface is connected to the xC port
0x0020	*RCX_HW_ASSEMBLY_ASI* The connected interface allows communication according to the AS-INTERFACE standard
0x0030	*RCX_HW_ASSEMBLY_CAN* The connected interface allows communication according to the CAN (Controller Area Network) specification
0x0040	*RCX_HW_ASSEMBLY_DEVICENET* The connected interface allows communication according to the DeviceNet specification

Variable: *ausHWOptions*	
Value	**Definition / Description**
0x0050	*RCX_HW_ASSEMBLY_PROFIBUS* The connected interface allows communication according to the PROFIBUS specification
0x0070	*RCX_HW_ASSEMBLY_CCLINK* The connected interface allows communication according to the CC-Link specification
0x0080	*RCX_HW_ASSEMBLY_ETHERNET* (internal Phy) The connected interface allows communication via Ethernet (internal Phy)
0x0081	*RCX_HW_ASSEMBLY_ETHERNET_X_PHY* (external Phy) The connected interface allows communication via Ethernet (external Phy)
0x0082	*RCX_HW_ASSEMBLY_ETHERNET_FIBRE_OPTIC* (internal Phy) The connected interface allows communication via Ethernet (internal Phy) with connected fiber optics
0x0090	*RCX_HW_ASSEMBLY_SPI* (Serial Peripheral Interface) The connected interface allows communication via a SPI (Serial Peripheral Interface) interface
0x00A0	*RCX_HW_ASSEMBLY_IO_LINK* The connected interface allows communication according to the IO-Link specification
0x00B0	*RCX_HW_ASSEMBLY_COMPONET* The connected interface allows communication according to the CompoNet specification
0xFFF4	*RCX_HW_ASSEMBLY_I2C_UNKNOWN* I2C is used to determine physical interface but Interface can't be determined (e.g. option module is not connected)
0xFFF5	*RCX_HW_ASSEMBLY_SSI* The physical interface is SSI conform
0xFFF6	*RCX_HW_ASSEMBLY_SYNC* xC port is used for special synchronization signals
0xFFFA	*RCX_HW_ASSEMBLY_TOUCH_SCREEN* xC port is connected to a touch screen interface
0xFFFB	*RCX_HW_ASSEMBLY_I2C_PIO* I2C is used to determine physical interface. This is used for option modules (AIFX modules) offering the identification via I2C. If the detection of the option module succeeds, the value is replaced by the real information from the option module. If the module detection fails, *I2C INTERFACE UNKNOWN* is shown.
0xFFFC	*RCX_HW_ASSEMBLY_I2C_PIO_NT* (netTAP device) I2C is used to determine a physical interface on a netTAP hardware platform. If the detection succeeds, the value is replaced by the real information from the option module
0xFFFD	*RCX_HW_ASSEMBLY_PROPRIETARY* Unspecified physical interface
0xFFFE	*RCX_HW_ASSEMBLY_NOT_CONNECTED* No physical interface connectable to the xC port (the xC port can only be used for chip-internal purposes)
0xFFFF	RESERVED, DO NOT USE

Table 50: System Channel: Hardware Assembly Options (xC Port 0..3)

Manufacturer

The manufacturer code defines the manufacturer of the device (hardware)

Variable: *usManufacturer*	
Value	**Definition / Description**
0x0000	*RCX_MANUFACTURER_UNDEFINED* Undefined manufacturer
0x0001... ... 0x00FF	*RCX_MANUFACTURER_HILSCHER_GMBH* Manufacturer is Hilscher Gesellschaft für Systemautomation mbH **Note: 0x0001...0x000FF reserved for Hilscher** RCX_MANUFACTURER_HILSCHER_GMBH_MAX
Other values are usable for third party manufacturers and given by Hilscher	

Table 51: System Channel: Manufacturer

Production Date

The production date entry is comprised of the calendar week and year (starting with year 2000) when the hardware was produced. Both, year and week are shown in hexadecimal notation.

If the value is equal to zero, the manufacturer date is not set.

Variable: *usProductionDate*	
Bit No. 15..8 = Production Year	**Bit No 7..0 = Production (Calendar) Week**
0 - 255 (+2000)	1 - 52

Table 52: System Channel: Production Date

Example:
Production Date = 0x062B indicates year = 2006 / production week = 43

License Flags 1 / License Flags 2

License flags contain license information for the netX firmware and tools and are used to enable functionalities which are protected by them. The flags are stored in a security memory on the hardware and read during system startup.

If the license information fields / flags are zero, a license or license code is not set.

License Flags 1 are used to enable fieldbus master protocol stacks.

- Bits 0 to bit 29 License flags dedicated to a specific protocol stack
- Bits 30 and 31 Number of concurrent master protocol stack licenses

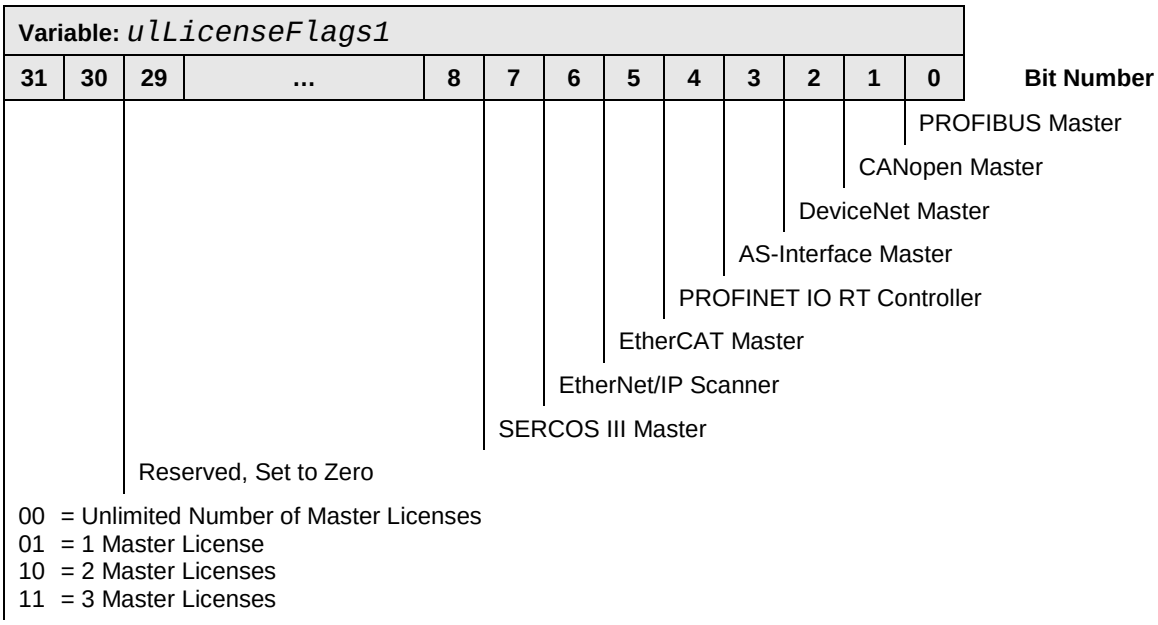


Table 53: System Channel: License Flags 1

License Flags 2 are used for tool licensing, e. g. SYCON.net, OPC servers and drivers etc.

If a flag is set, a tool license is present.

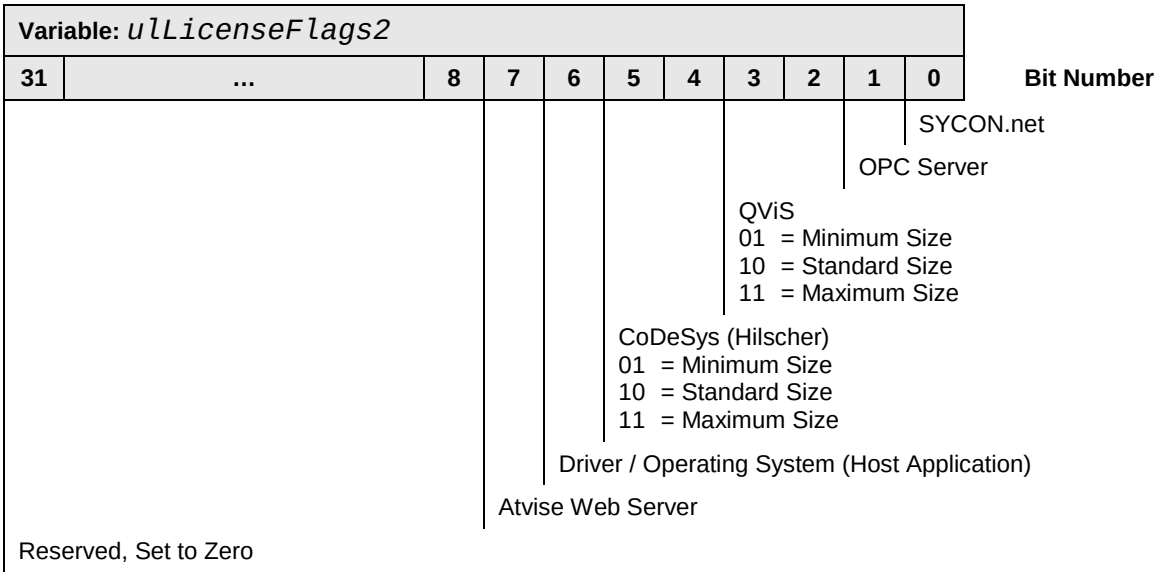


Table 54: System Channel: License Flags 2

netXLicenseID / netXLicenseFlags

netX license ID and flags (*usNetxLicenseID* / *usNetxLicenseFlags*) are reserved for OEM customers allowing them to integrate their own license definition.

Device Class

The device class is used to create device groups specified by their functionality and hardware options (e.g. used netX chip, special functionalities). This is necessary because a netX firmware may not support all hardware devices or just one specific device.

The netX firmware file header also contains a device class definition, specifying which devices are supported. This information can be used by host applications to verify if a given firmware will work on a device before downloading it to the hardware.

The following hardware device classes are defined.

Variable: *usDeviceClass*		
Value	**Definition**	**Description**
0x0000	RCX_HW_DEV_CLASS_UNDEFINED	Hardware is not defined
0x0001	RCX_HW_DEV_CLASS_UNCLASSIFIABLE	Hardware not classifiable
0x0002	RCX_HW_DEV_CLASS_CHIP_NETX_500	netX500 chip based hardware
0x0003	RCX_HW_DEV_CLASS_CIFX	All PC cards (ISA/PCI/PCIe)
0x0004	RCX_HW_DEV_CLASS_COMX_100	COMX module netX00 based
0x0005	RCX_HW_DEV_CLASS_EVA_BOARD	netX Evaluation board (e.g. NXHX51)
0x0006	RCX_HW_DEV_CLASS_NETDIMM	netDIMM netX500 chip based DIMM module
0x0007	RCX_HW_DEV_CLASS_CHIP_NETX_100	netX100 chip based
0x0008	RCX_HW_DEV_CLASS_NETX_HMI	netHMI hardware
0x0009		reserved
0x000A	RCX_HW_DEV_CLASS_NETIO_50	netIO netX50 based I/O module
0x000B	RCX_HW_DEV_CLASS_NETIO_100	netIO netX100 based I/O module
0x000C	RCX_HW_DEV_CLASS_CHIP_NETX_50	netX50 based hardware
0x000D	RCX_HW_DEV_CLASS_GW_NETPAC	netPAC Gateway
0x000E	RCX_HW_DEV_CLASS_GW_NETTAP	netTAP netX100 based Gateway
0x000F	RCX_HW_DEV_CLASS_NETSTICK	netSTICK netX50 based USB stick
0x0010	RCX_HW_DEV_CLASS_NETANALYZER	netANALYZER PC card
0x0011	RCX_HW_DEV_CLASS_NETSWITCH	netSWITCH Ethernet switch
0x0012	RCX_HW_DEV_CLASS_NETLINK	netLINK network linking
0x0013	RCX_HW_DEV_CLASS_NETIC_50	netIC netX50 based DIL-32 communication IC
0x0014	RCX_HW_DEV_CLASS_NPLC_C100	netPLC netX100 based PLC PC card
0x0015	RCX_HW_DEV_CLASS_NPLC_M100	netPLC netX100 based PLC module
0x0016	RCX_HW_DEV_CLASS_GW_NETTAP_50	netTAP netX50 based Gateway
0x0017	RCX_HW_DEV_CLASS_NETBRICK	netBRICK netX100 based Gateway (IP67)
0x0018	RCX_HW_DEV_CLASS_NPLC_T100	netPLC netX100 based PLC module (DIN rail)
0x0019	RCX_HW_DEV_CLASS_NETLINK_PROXY	netLINK proxy
0x001A	RCX_HW_DEV_CLASS_CHIP_NETX_10	netX10 based hardware
0x001B	RCX_HW_DEV_CLASS_NETJACK_10	netJACK netX10 based exchangeable module
0x001C	RCX_HW_DEV_CLASS_NETJACK_50	netJACK netX50 based exchangeable module
0x001D	RCX_HW_DEV_CLASS_NETJACK_100	netJACK netX100 based exchangeable module
0x001E	RCX_HW_DEV_CLASS_NETJACK_500	netJACK netX500 based exchangeable module
0x001F	RCX_HW_DEV_CLASS_NETLINK_10_USB	netLINK netX10 based with USB connection
0x0020	RCX_HW_DEV_CLASS_COMX_10	COMX module netX10 based
0x0021	RCX_HW_DEV_CLASS_NETIC_10	netIC netX10 based DIL-32 communication IC
0x0022	RCX_HW_DEV_CLASS_COMX_50	COMX module netX50 based

Variable: *usDeviceClass*		
Value	**Definition**	**Description**
0x0023	RCX_HW_DEV_CLASS_NETRAPID_10	netRAPID netX10 based ready to solder chip-carrier
0x0024	RCX_HW_DEV_CLASS_NETRAPID_50	netRAPID netX50 based ready to solder chip-carrier
0x0025	RCX_HW_DEV_CLASS_NETSCADA_T51	netSCADA netX51 based visualizing modem
0x0026	RCX_HW_DEV_CLASS_CHIP_NETX_51	netX51 based hardware
0x0027	RCX_HW_DEV_CLASS_NETRAPID_51	netRAPID netX51 based ready to solder chip-carrier
0x0028	RCX_HW_DEV_CLASS_GW_EU5C	EU5C Gateway
0x0029	RCX_HW_DEV_CLASS_NETSCADA_T50	netSCADA netX50 based visualizing modem
0x002A	RCX_HW_DEV_CLASS_NETSMART_50	netSMART netX50 based Smartwire module
0x002B	RCX_HW_DEV_CLASS_IOLINK_GW_51	netX51 bases IO-LINK gateway
0x002C	RCX_HW_DEV_CLASS_NETHMI_B500	netHMI netX500 based operator panel
0x002D	RCX_HW_DEV_CLASS_CHIP_NETX_52	netX52 based hardware
0x002E	RCX_HW_DEV_CLASS_COMX_51	COMX module netX51 based
0x002F	RCX_HW_DEV_CLASS_NETJACK_51	netJACK netX51 based exchangeable module
0x0030	RCX_HW_DEV_CLASS_NETHOST_T100	netHOST netX100 based Lan controlled master
0x0031	RCX_HW_DEV_CLASS_NETSCOPE_C100	netSCOPE netX100 based PC card
0x0032	RCX_HW_DEV_CLASS_NETRAPID_52	netRAPID netX52 based ready to solder chip-carrier
0x0033	RCX_HW_DEV_CLASS_NETSMART_T51	netSMART netX51 based Smartwire module
0x0034	RCX_HW_DEV_CLASS_NETSCADA_T52	netSCADA netX52 based visualizing modem
0x0035	RCX_HW_DEV_CLASS_NETSAFETY_51	netSAFETY nex51 based safety module
0x0036	RCX_HW_DEV_CLASS_NETSAFETY_52	netSAFETY nex52 based safety module
0x0037	RCX_HW_DEV_CLASS_NETPLC_J500	netPLC netX500 based PLC module
0x0038	RCX_HW_DEV_CLASS_NETIC_52	netIC netX52 based (DIL-32 communication IC)
0x0039	RCX_HW_DEV_CLASS_GW_NETTAP_151	netTAP151 dual netX51 based Gateway
0x003A	RCX_HW_DEV_CLASS_CHIP_NETX_4000_COM	netX 4000 chip based hardware (communication)
0x003B	RCX_HW_DEV_CLASS_CIFX_4000	CIFX 4000 PC cards (PCIe)
0x003C	RCX_HW_DEV_CLASS_CHIP_NETX_90_COM	netX 90 chip based hardware (communication)
0x003D	RCX_HW_DEV_CLASS_NETRAPID_51_IO	netRAPID netX51 based for I/O ready to solder chip-carrier
0x003E	RCX_HW_DEV_CLASS_GW_NETTAP_151_CCIES	netTAP151 netX100 and netX51 based Gateway for CC-Link IE Slave
0x003F	RCX_HW_DEV_CLASS_CIFX_CCIES	PC cards for CC-Link IE Slave with 1 GBit Ethernet
0x0040 … 0x7FFF		Reserved
0x8000 … 0xEFFF		Reserved
0xFFFE	RCX_HW_DEV_CLASS_OEM_DEVICE	OEM Device
Other values are reserved		

Table 55: System Channel: Device Class

Hardware Revision

The hardware revision field indicates the current revision of a module. In difference to the printed device label, the revision field counts from 1 to 35 while the printed label shows the letters A to Z for revision numbers greater 9.

Variable: *bHwRevision*		
Revision number	**Revision Field**	**Printed Device Label**
1..9	1..9	1..9
10..35	10..35	A..Z

Table 56: System Channel: Hardware Revision

Hardware Compatibility Index

The hardware compatibility index *bHwCompatibility* is used to indicate incompatible hardware changes during the life time of a hardware product. The index starts with 0 and is incremented for every incompatible hardware change made to the hardware. The same index also exists in the header of a netX firmware file which allows a host application to match a firmware index against a given hardware index. This allows a host application to prevent a firmware download if necessary.

Device Identification Number

The device identification number *bDevIdNumber* is intended to be used as a uniquely distinction of hardware modules/cards from each other. This might be necessary if multiple modules/cards of the same type are used at the same time on one host system (e.g. multiple PCI cards in a PC system).

The creation of the identification number is hardware based and needs either a DIP switch or Rotary switch equipped on the modules/cards. In this case the setting from the switch will be read by the firmware and shown in the identification number.

A zero (0) indicates that no identification number was assigned to the device. The value range of the identification number depends on the used DIP switch/Rotary switch (usually between 0..9).

Note: The *Device Identification Number* is a hardware created number and the hardware must support this option. It should not be mixed up with any addresses from a fieldbus system (e.g. slave address on a fieldbus network).

5.2.2 Channel Information Block

The channel information block structure holds information about the channels available in the dual-port memory. A channel description is a 16 bytes structure and the first element (*bChannelType*) of the structure defines the channel type and therefore the corresponding channel structure.

Structure of the *Channel Information Block*:

Channel Information Block: *NETX_CHANNEL_INFO_BLOCK*			
Address	**Channel**	**Structure Name:** *NETX_SYSTEM_CHANNEL_INFO*	
0x0030	**System Channel**	**Data Type**	**Description**
		uint8_t	Channel Type = SYSTEM (see page 83)
		uint8_t	Reserved (set to zero)
		uint8_t	Size / Position of Handshake Registers
		uint8_t	Total Number of Blocks
		uint32_t	Size of Channel in Bytes
		uint16_t	Size of Send and Receive Mailbox in Bytes
		uint16_t	Mailbox Start Offset
… 0x003F		uint8_t[4]	4 Byte Reserved (set to zero)
Address	**Channel**	**Structure Name:** *NETX_HANDSHAKE_CHANNEL_INFO*	
0x0040	**Handshake Channel**	**Data Type**	**Description**
		uint8_t	Channel Type = HANDSHAKE (see page 83)
		uint8_t[3]	3 Byte Reserved (set to zero)
		uint32_t	Channel Size in Bytes
… 0x004F		uint8_t[8]	8 Byte Reserved
Address	**Channel**	**Structure Name:** *NETX_COMMUNICATION_CHANNEL_INFO*	
0x0050	**Communication Channel 0**	**Data Type**	**Description**
		uint8_t	Channel Type = COMMUNICATION (see page 83)
		uint8_t	Channel ID, Channel Number
		uint8_t	Size / Position of Handshake Registers
		uint8_t	Total Number of Blocks in this Channel
		uint32_t	Size of Channel in Bytes
		uint16_t	Communication Class (Master, Slave...)
		uint16_t	Protocol Class (PROFIBUS, PROFINET....)
		uint16_t	Protocol Conformance Class (DPV1, DPV2...)
… 0x005F		uint8_t[2]	2 Byte Reserved (set to zero)
0x0060.. … 0x008F	**Communication Channel 1, 2 & 3**	Structure	Same as Communication Channel 0

Address	Channel	Structure Name: *NETX_APPLICATION_CHANNEL_INFO*	
0x0090	**Application Channel 0**	**Data Type**	**Description**
		uint8_t	Channel Type = APPLICATION (see page 83)
		uint8_t	Channel ID, Channel Number
		uint8_t	Size / Position of Handshake Registers
		uint8_t	Total Number of Blocks in this Channel
		uint32_t	Size of Channel in Bytes
… 0x009F		uint8_t [8]	8 Byte Reserved (set to zero)
0x00A0 … 0x00AF	**Application Channel 1**	Structure	Same as Application Channel 0

Table 57: System Channel: Channel Information Block

Structure Reference: NETX_CHANNEL_INFO_BLOCK

```
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;
```

Structure Reference: NETX_SYSTEM_CHANNEL_INFO

```
typedef struct NETX_SYSTEM_CHANNEL_INFOtag
{
  uint8_t    bChannelType;
  uint8_t    bReserved;
  uint8_t    bSizePositionOfHandshake;
  uint8_t    bNumberOfBlocks;
  uint32_t   ulSizeOfChannel;
  uint16_t   usSizeOfMailbox;
  uint16_t   usMailboxStartOffset;
  uint8_t    abReserved[4];
} NETX_SYSTEM_CHANNEL_INFO;
```

Structure Reference: NETX_HANDSHAKE_CHANNEL_INFO

```
typedef struct NETX_HANDSHAKE_CHANNEL_INFOtag
{
  uint8_t    bChannelType;
  uint8_t    bReserved[3];
  uint32_t   ulSizeOfChannel;
  uint8_t    abReserved[8];
} NETX_HANDSHAKE_CHANNEL_INFO;
```

Structure Reference: NETX_COMMUNICATION_CHANNEL_INFO

```
typedef struct NETX_COMMUNICATION_CHANNEL_INFOtag
{
  uint8_t    bChannelType;
  uint8_t    bChannelId;
  uint8_t    bSizePositionOfHandshake;
  uint8_t    bNumberOfBlocks;
  uint32_t   ulSizeOfChannel;
  uint16_t   usCommunicationClass;
  uint16_t   usProtocolClass;
  uint16_t   usConformanceClass;
  uint8_t    abReserved[2];
} NETX_COMMUNICATION_CHANNEL_INFO;
```

Structure Reference: NETX_APPLICATION_CHANNEL_INFO

```
typedef struct NETX_APPLICATION_CHANNEL_INFOtag
{
  uint8_t    bChannelType;
  uint8_t    bChannelId;
  uint8_t    bSizePositionOfHandshake;
  uint8_t    bNumberOfBlocks;
  uint32_t   ulSizeOfChannel;
  uint8_t    abReserved[8];
} NETX_APPLICATION_CHANNEL_INFO;
```

Channel Type

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

Variable: *bChannelType*		
Value	**Definition**	**Description**
0x00	RCX_CHANNEL_TYPE_UNDEFINED	Undefined
0x01	RCX_CHANNEL_TYPE_NOT_AVAILABLE	Not available
0x02	RCX_CHANNEL_TYPE_RESERVED	Reserved
0x03	RCX_CHANNEL_TYPE_SYSTEM	System Channel
0x04	RCX_CHANNEL_TYPE_HANDSHAKE	Handshake Channel
0x05	RCX_CHANNEL_TYPE_COMMUNICATION	Communication Channel
0x06	RCX_CHANNEL_TYPE_APPLICATION	Application Channel
0x07 … 0x7F	Reserved	Reserved for future use
0x80 … 0xFF	RCX_CHANNEL_TYPE_USER_DEFINED_START	Start of user defined channels

Table 58: System Channel: Channel Type

Channel ID, Channel Number (Communication and Application Channel Only)

The value given in *bChannelId* is used to enumerate the communication and application channels. The value is unique in the system and ranges from 0 to 7 and corresponds to the order of a channel in the DPM (e.g. Communication Channel 0 = 0, Communication Channel 1 = 1 etc.).

Size / Position of Handshake Registers

This field identifies the position of the handshake registers and their size. The handshake registers may be located at the beginning of the channel itself or in a separate handshake area.

The size of the handshake registers can be either 8 or 16 bit.

Variable: *bSizePositionOfHandshake*	
Bit No.	**Definition / Description**
0..3	Handshake Register Size 0 = RCX_HANDSHAKE_SIZE_NOT_AVAILABLE 1 = RCX_HANDSHAKE_SIZE_8BIT (default for the System Channel) 2 = RCX_HANDSHAKE_SIZE_16BIT (default for the Communication Channel) Other values are reserved
4..7	Handshake Register Position 0 = RCX_HANDSHAKE_POSITION_BEGINNING (located at the beginning of a channel) 1 = RCX_HANDSHAKE_POSITION_CHANNEL (default: located in the handshake channel) Other values are reserved

Table 59: System Channel: Size / Position of Handshake Registers

Total Number of Blocks

A channel consists of data blocks (e.g. mailbox and status blocks) and the number of blocks *bNumberOfBlocks* indicates how many blocks are contained in the channel structure.

Size of Channel

This field *ulSizeOfChannel* contains the size of the entire channel in bytes.

Size of Send and Receive Mailbox (System Channel Only)

The mailbox size field *usSizeOfMailbox* holds the size of the system mailbox structure (sum of the send / receive mailbox). While the size of the send and receive mailbox is always symmetric, the given size must be divided by 2 to get the size of the send/receive mailbox structure.

The default size of the send/receive mailbox structure is 128 bytes (sum = 256 bytes). Each mailbox (send & receive) consists of one field for the package counter (packages accepted / packages waiting) and 124 bytes for the user data (see section 5.2.6).

Mailbox Start Offset (System Channel Only)

The mailbox start offset element *usMailboxStartOffset* defines location (byte offset) of the mailbox inside the system channel, starting with the send mailbox structure.

Communication Class (Communication Channel Only)

The communication class defines functionality (e.g. Master / Slave etc.) of the fieldbus protocol stack and can be used by an application to adapt the handling for the protocol stack.

Variable: *usCommunicationClass*		
Value	**Definition**	**Description**
0x0000	RCX_COMM_CLASS_UNDEFINED	Undefined
0x0001	RCX_COMM_CLASS_UNCLASSIFIABLE	Unclassifiable
0x0002	RCX_COMM_CLASS_MASTER	Master stack
0x0003	RCX_COMM_CLASS_SLAVE	Slave stack
0x0004	RCX_COMM_CLASS_SCANNER	Scanner
0x0005	RCX_COMM_CLASS_ADAPTER	Adapter
0x0006	RCX_COMM_CLASS_MESSAGING	Messaging (no cyclic I/O data)
0x0007	RCX_COMM_CLASS_CLIENT	Client
0x0008	RCX_COMM_CLASS_SERVER	Server
0x0009	RCX_COMM_CLASS_IO_CONTROLLER	IO-Controller
0x000A	RCX_COMM_CLASS_IO_DEVICE	IO-Device
0x000B	RCX_COMM_CLASS_IO_SUPERVISOR	IO-Supervisor
0x000C	RCX_COMM_CLASS_GATEWAY	Gateway
0x000D	RCX_COMM_CLASS_MONITOR	Monitor / Analyzer
0x000E	RCX_COMM_CLASS_PRODUCER	Producer
0x000F	RCX_COMM_CLASS_CONSUMER	Producer
0x0010	RCX_COMM_CLASS_SWITCH	Switch
0x0011	RCX_COMM_CLASS_HUB	Hub
0x0012	RCX_COMM_CLASS_COMBI	Not used in the DPM This value is used inside a netX firmware file header, if the firmware file combines different protocol stacks in one file.
0x0013	RCX_COMM_CLASS_MANAGING_NODE	Managing node
0x0014	RCX_COMM_CLASS_CONTROLLED_NODE	Controlled node
0x0015	RCX_COMM_CLASS_PLC	Programmable Logic Controller (PLC)
0x0016	RCX_COMM_CLASS_HMI	Human Machine Interface (HMI)
0x0017	RCX_COMM_CLASS_ITEM_SERVER	Item server
0x0018	RCX_COMM_CLASS_SCADA	SCADA system
Other values are reserved		

Table 60: System Channel: Communication Class

Protocol Class (Communication Channel Only)

This field defines the fieldbus protocol running on the communication channel or a specific process like a PLC program, running on the communication channel.

Variable: *usProtocolClass*		
Value	**Definition**	**Description**
0x0000	RCX_PROT_CLASS_UNDEFINED	Undefined
0x0001	RCX_PROT_CLASS_3964R	3964R
0x0002	RCX_PROT_CLASS_ASINTERFACE	AS Interface
0x0003	RCX_PROT_CLASS_ASCII	ASCII
0x0004	RCX_PROT_CLASS_CANOPEN	CANopen
0x0005	RCX_PROT_CLASS_CCLINK	CC-Link
0x0006	RCX_PROT_CLASS_COMPONET	CompoNet
0x0007	RCX_PROT_CLASS_CONTROLNET	ControlNet
0x0008	RCX_PROT_CLASS_DEVICENET	DeviceNet
0x0009	RCX_PROT_CLASS_ETHERCAT	EtherCAT
0x000A	RCX_PROT_CLASS_ETHERNET_IP	EtherNet/IP
0x000B	RCX_PROT_CLASS_FOUNDATION_FB	Foundation Fieldbus
0x000C	RCX_PROT_CLASS_FL_NET	FL Net
0x000D	RCX_PROT_CLASS_INTERBUS	InterBus
0x000E	RCX_PROT_CLASS_IO_LINK	IO-Link
0x000F	RCX_PROT_CLASS_LON	LON
0x0010	RCX_PROT_CLASS_MODBUS_PLUS	Modbus Plus
0x0011	RCX_PROT_CLASS_MODBUS_RTU	Modbus RTU
0x0012	RCX_PROT_CLASS_OPEN_MODBUS_TCP	Open Modbus TCP
0x0013	RCX_PROT_CLASS_PROFIBUS_DP	PROFIBUS DP
0x0014	RCX_PROT_CLASS_PROFIBUS_MPI	PROFIBUS MPI
0x0015	RCX_PROT_CLASS_PROFINET_IO	PROFINET IO
0x0016	RCX_PROT_CLASS_RK512	RK512
0x0017	RCX_PROT_CLASS_SERCOS_II	SERCOS II
0x0018	RCX_PROT_CLASS_SERCOS_III	SERCOS III
0x0019	RCX_PROT_CLASS_TCP_IP_UDP_IP	TCP/IP, UDP/IP
0x001A	RCX_PROT_CLASS_POWERLINK	Powerlink
0x001B	RCX_PROT_CLASS_HART	HART
0x001C	RCX_PROT_CLASS_COMBI	Not used in the DPM This value is used inside a netX firmware file header, if the firmware file combines different protocol stacks in one file.
0x001D	RCX_PROT_CLASS_PROG_GATEWAY	Programmable Gateway The programmable gateway function uses netSCRIPT as programming language.
0x001E	RCX_PROT_CLASS_PROG_SERIAL	Programmable Serial The programmable serial protocol function uses netSCRIPT as programming language.
0x001F	RCX_PROT_CLASS_PLC_CODESYS	PLC: CoDeSys
0x0020	RCX_PROT_CLASS_PLC_PROCONOS	PLC: ProConOS
0x0021	RCX_PROT_CLASS_PLC_IBH_S7	PLC: IBH S7
0x0022	RCX_PROT_CLASS_PLC_ISAGRAF	PLC: ISaGRAF
0x0023	RCX_PROT_CLASS_VISU_QVIS	Visualization: QviS

Variable: *usProtocolClass*		
Value	**Definition**	**Description**
0x0024	RCX_PROT_CLASS_ETHERNET	Ethernet
0x0025	RCX_PROT_CLASS_RFC1006	RFC1006
0x0026	RCX_PROT_CLASS_DF1	DF1
0x0027	RCX_PROT_CLASS_VARAN	VARAN
0x0028	PROT_CLASS_3S_PLC_HANDLER	3S PLC Handler
0x0029	RCX_PROT_CLASS_ATVISE	Atvise
0x002A	RCX_PROT_CLASS_MQTT	MQTT
0x002B	RCX_PROT_CLASS_OPCUA	OPCUA
0xFFF0	RCX_PROT_CLASS_OEM	OEM, Proprietary
Other values are reserved		

Table 61: System Channel: Protocol Class

Conformance Class (Communication Channel Only)

The conformance class describes an additional characteristic of a fieldbus protocol or process and depends on the **Protocol Class**.

It is used to describe sub functionalities of the protocol stack like PROFIBUS DPV1/DPV2 support or the conformance class (A/B/C) of a PROFINET protocol.

Because of protocol class dependency, values in here are specified in the corresponding protocol/process specific manuals.

5.2.3 System Handshake Block

The *System Handshake Block* is a reserved area which is intended to hold the system handshake register and maybe used in the future.

Note: Currently all handshake registers of all channels are located in the *Handshake Channel* of the DPM.

The *System Handshake Block* inside a channel is not yet supported and therefore set to zero.

System Handshake Block: *NETX_HANDSHAKE_CELL*			
Offset	**Type**	**Name**	**Description**
0x00B0	uint8_t	t8Bit.abData[2]	Reserved, set to 0
0x00B2	uint8_t	t8Bit.bNetxFlags	netX system channel handshake register
0x00B3	uint8_t	t8Bit.bHostFlags	Host system channel handshake register

Table 62: System Channel: System Handshake Block

Structure Reference - System Handshake Block

```
/*****************************************************************************/
/*! Handshake cell definition                                                */
/*****************************************************************************/
typedef __RCX_PACKED_PRE union NETX_HANDSHAKE_CELLtag
{
  __RCX_PACKED_PRE struct
  {
    volatile uint8_t abData[2];        /*!< Data value, not belonging to handshake */
    volatile uint8_t bNetxFlags;       /*!< Device status flags (8Bit Mode) */
    volatile uint8_t bHostFlags;       /*!< Device command flags (8Bit Mode) */
  } __RCX_PACKED_POST t8Bit;

  __RCX_PACKED_PRE struct
  {
    volatile uint16_t usNetxFlags;     /*!< Device status flags (16Bit Mode) */
    volatile uint16_t usHostFlags;     /*!< Device command flags (16Bit Mode)*/
  } __RCX_PACKED_POST t16Bit;
  volatile uint32_t ulValue;           /*!< Handshake cell value */
} NETX_HANDSHAKE_CELL;
```

5.2.4 System Control Block

The *System Control Block* offers the possibility to define commands for the netX system in the future. Those commands will be initiated by the host application and passed to the netX via the COS (Change of State) mechanism.

Note: Currently no commands are defined.

System Control Block: *NETX_SYSTEM_CONTROL_BLOCK*			
Offset	**Type**	**Name**	**Description**
0x00B8	uint32_t	`ulSystemCommandCOS`	System Change Of State System Reset = RCX_SYS_RESET_COOKIE
0x00BC	uint32_t	`ulReserved`	Reserved Not used, set to 0

Table 63: System Channel: System Control Block

Structure Reference - System Control Block

```
typedef struct NETX_SYSTEM_CONTROL_BLOCKtag
{
  uint32_t   ulSystemCommandCOS;
  uint32_t   ulReserved;
} NETX_SYSTEM_CONTROL_BLOCK;
```

The *Change of State* (COS) mechanism is described in section 4.3 of this manual.

5.2.5 System Status Block

The system status block provides general status information of the device and the netX firmware.

System Status Block: *NETX_SYSTEM_STATUS_BLOCK*			
Offset	**Type**	**Name**	**Description**
0x00C0	uint32_t	ulSystemCOS	System Change Of State General system states (see page 91)
0x00C4	uint32_t	ulSystemStatus	System Status Information about file system, boot medium etc. (see page 91)
0x00C8	uint32_t	ulSystemError	System Error Indicates runtime errors of the firmware (see page 92)
0x00CC	uint32_t	ulBootError	Boot Error Indicates faults during the hardware boot process (see page 92)
0x00D0	uint32_t	ulTimeSinceStart	Time Since Startup Time elapsed since system start in seconds (see page 92)
0x00D4	uint16_t	usCpuLoad	CPU Load CPU Load in 0.01% units (see page 92)
0x00D6	uint16_t	usReserved	Reserved, set to 0
0x00D8	uint32_t	ulHWFeatures	Hardware Features Available hardware features (see page 93)
0x00DC … 0x00FF	uint8_t	abReserved[36]	Reserved Set to 0

Table 64: System Channel: System Status Block

System Status Block Structure Reference

```
typedef struct NETX_SYSTEM_STATUS_BLOCKtag
{
  uint32_t   ulSystemCOS;
  uint32_t   ulSystemStatus;
  uint32_t   ulSystemError;
  uint32_t   ulBootError;
  uint32_t   ulTimeSinceStart;
  uint16_t   usCpuLoad
  uint16_t   usReserved;
  uint32_t   ulHWFeatures;
  uint8_t    abReserved[36];
} NETX_SYSTEM_STATUS_BLOCK;
```

System Change of State

The change of state field contains information about the current operating status of the system channel.

Variable: *ulSystemCOS*	
Bit No.	**Definition / Description**
0..30	empty / undefined
31	DPM Memory Layout Indicates if the DPM follows the default memory layout it set once after power up or reset. 0 = none default layout 1 = RCX_SYS_COS_DEFAULT_MEMORY
Other values are reserved	

Table 65: System Channel: System Change of State

System Status

The system status holds general state information about the device, e.g. the used boot media, the file system and its location and the supported firmware type.

Variable: *ulSystemStatus*												
31	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**	**…**	**1**	**0**	**Bit Number**
											RCX_SYS_STATUS_OK	
								unused, set to zero				
				Boot Medium 0000 = RCX_SYS_STATUS_BOOTMEDIUM_RAM 0001 = RCX_SYS_STATUS_BOOTMEDIUM_SERFLASH 0010 = RCX_SYS_STATUS_BOOTMEDIUM_PARFLASH								
			unused, set to zero									
		RCX_SYS_STATUS_NO_SYSVOLUME										
	RCX_SYS_STATUS_SYSVOLUME_FFS											
RCX_SYS_STATUS_NXO_SUPPORTED												

Table 66: System Channel: System Status Field

Bit No.	Definition / Description
0	Actual System State (RCX_SYS_STATUS_OK) If set, the data in system status register is valid (for backwards compatibility) 0 = System status not valid 1 = RCX_SYS_STATUS_OK
1..23	Reserved, set to 0
24..27	Boot Medium 0000 = RAM (RCX_SYS_STATUS_BOOTMEDIUM_RAM) 0001 = Serial Flash (RCX_SYS_STATUS_BOOTMEDIUM_SERFLASH) 0010 = Parallel Flash (RCX_SYS_STATUS_BOOTMEDIUM_PARFLASH) Other values are reserved
28	Reserved, set to 0
29	System Volume (RCX_SYS_STATUS_NO_SYSVOLUME) Indicates if the device uses a file system 0 = File system available 1 = No file system available

30	Flash File System (RCX_SYS_STATUS_SYSVOLUME_FFS) Indicates if a flash (remanent) file system is available 0 = No flash file system available 1 = Flash file system available
31	Loadable Modules (RCX_SYS_STATUS_NXO_SUPPORTED) Indication if the firmware supports loadable modules (NXOs) 0 = Loadable modules (NXO) not supported 1 = Loadable modules (NXO) supported

Table 67: System Channel: System Status Field Description

System Error

The system error field *ulSystemError* holds information about general netX firmware errors (0 = no error).

The system error field works in conjunction with the NSF_ERROR bit in the *netX System Flags* register. NSF_ERROR is used to indicate an error while the error number is shown in the system error field. Error codes are listed in section 7.

Boot Error

The boot error field *ulBootError* is used by the 2nd Stage Boot Loader to indicate errors during system startup or firmware loading.

Boot loader errors are described in the 2nd Stage Bootloader manual and listed in section 7.

Time Since Startup

Time since startup *ulTimeSinceStart* is used to shows the elapsed time since the last system start (e.g. Power-On / hardware reset). The time is given in seconds [s] and can be used to detect unexpected system re-starts.

CPU Load

The CPU load field *usCpuLoad* indicates the current netX CPU usage. The value is updated every second and has a resolution of 0.01% (e.g. a value of 10000 is equal to 100%).

Hardware Features

The hardware features field is used to indicate additional hardware components available, assembled on a netX device.

Variable: *ulHWFeatures*

31	…	11	10	9	8	7	6	5	4	3	2	1	0	Bit Number

External Memory Type
0000 = None
0001 = MRAM 64*16 Bit (1 Mbit/128 KB)

Reserved, set to 0

External Memory Access
00 = No access
01 = External access (host)
10 = Internal access
11 = External and internal access

Real-Time Clock
00 = No RTC
01 = RTC internal
10 = RTC external
11 = RTC emulated

Clock State
0 = Time not valid
1 = Time valid

Unused, set to zero

Table 68: System Channel: Hardware Features Field

Bit No.	Definition / Description
0..3	External Memory Type A netX device can provide an external memory which is independent of the DPM. 0000 = No external memory available (RCX_SYSTEM_EXTMEM_TYPE_NONE) 0001 = MRAM 128Kbyte available (RCX_SYSTEM_EXTMEM_TYPE_MRAM_128K)
4..5	Reserved, set to 0
6..7	External Memory Access The external memory is accessible by the host application and the netX firmware. 00 = No access (RCX_SYSTEM_EXTMEM_ACCESS_NONE) 01 = External access by host (RCX_SYSTEM_EXTMEM_ACCESS_EXTERNAL) 10 = Internal access by netX (RCX_SYSTEM_EXTMEM_ACCESS_INTERNAL) 11 = External and internal (RCX_SYSTEM_EXTMEM_ACCESS_BOTH) **Note** If RCX_SYSTEM_EXTMEM_ACCESS_BOTH is defined, the size of the memory is divided by 2 while 1st half of the RAM is owned by the host application and the 2nd half of the RAM is owned by netX firmware.
8..9	Real-Time Clock These bits defining if a real-time clock is equipped on the netX device. By default all netX are offering a standard (none real-time) clock. 00 = No RTC (RCX_SYSTEM_HW_RTC_TYPE_NONE) 01 = RTC internal (RCX_SYSTEM_HW_RTC_TYPE_INTERNAL) 10 = RTC external (RCX_SYSTEM_HW_RTC_TYPE_EXTERNAL) 11 = RTC emulated (RCX_SYSTEM_HW_RTC_TYPE_EMULATED)
10	Clock State Indicates if the clock is set or not 0 = Time not valid, not initialized or battery fault 1 = Time valid, clock was set (RCX_SYSTEM_HW_RTC_STATE)
11..31	Reserved, set to 0

Table 69: System Channel: Hardware Feature Description Field

5.2.6 System Mailbox

The system mailbox allows access to the system channel and the general netX firmware (operating system services) via packet based commands. It is present as soon as the 2nd Stage Loader or a netX firmware is running.

A host application uses the mailbox system to download a protocol stack firmware, a fieldbus configuration or to determine the actual DPM layout if necessary (see section 4.1 for details).

The mailbox handling is described in section 4.1.

System Send Mailbox: *NETX_SYSTEM_SEND_MAILBOX*			
Direction: Host System ⇨ netX			
Offset	**Type**	**Name**	**Description**
0x0100	uint16_t	usPackagesAccepted	Packages Acceptable Number of packets that can be accepted by the firmware
0x0102	uint16_t	usReserved	Reserved, set to 0
0x0104 … 0x017F	uint8_t	abSendMbx[124]	Send Mailbox Buffer Buffer to insert the send packet

Table 70: System Channel: Send Mailbox

System Receive Mailbox: *NETX_SYSTEM_RECV_MAILBOX*			
Offset	**Type**	**Name**	**Description**
0x0180	uint16_t	usWaitingPackages	Waiting Packages Counter of packets waiting to be read by the host
0x182	uint16_t	usReserved	Reserved, set to 0
0x0184 … 0x01FF	uint8_t	abRecvMbx[124]	Receive Mailbox Buffer Buffer containing a packet received from the firmware

Table 71: System Channel: Receive Mailbox

Structure Reference: NETX_SYSTEM_SEND_MAILBOX

```
typedef struct NETX_SYSTEM_SEND_MAILBOXtag
{
  uint16_t   usPackagesAccepted;
  uint16_t   usReserved;
  uint8_t    abSendMbx[124];
} NETX_SYSTEM_SEND_MAILBOX;
```

Structure Reference: NETX_SYSTEM_RECV_MAILBOX

```
typedef struct NETX_SYSTEM_RECV_MAILBOXtag
{
  uint16_t   usWaitingPackages;
  uint16_t   usReserved;
  uint8_t    abRecvMbx[124];
} NETX_SYSTEM_RECV_MAILBOX;
```

5.3 Handshake Channel

In the default layout, the *Handshake Channel* follows the *System Channel*. It holds the handshake registers of all other channels in the DPM, providing the data transfer synchronization mechanism between the host system and the netX firmware.

Note: Offsets are given relative to the start offset of the channel start address.

There are three types of handshake registers.

- **System Handshake Registers**
  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 Registers**
  are used to synchronize cyclic and non-cyclic data exchange over IO data images and mailboxes for communication channels
- Application Handshake Registers
  are not supported yet

For compatibility reason, the handshake channel itself has a handshake register defined in the handshake channel.

Handshake Channel: *NETX_HANDSHAKE_CHANNEL*				
Offset	**Type**	**Element**	**Description**	
0x0000	NETX_HANDSHAKE_CELL	tSysFlags	System Channel	
	Offset	**Type**	**Element**	**Description**
	0x0000	uint8_t	abData[2]	reserved, set to 0
	0x0002	uint8_t	bNetxFlags	netX flag register
	0x0003	uint8_t	bHostFlags	Host flag register
Offset	**Type**	**Element**	**Description**	
0x0004	NETX_HANDSHAKE_CELL	tHskFlags	Handshake Channel (contains the fieldbus synchronization flags)	
	Offset	**Type**	**Element**	**Description**
	0x0004	uint16_t	usNSyncFlags	netX flag register
	0x0006	uint16_t	usHSyncFlags	Host flag register
Offset	**Type**	**Element**	**Description**	
0x0008	NETX_HANDSHAKE_CELL	tCommFlags0	Communication Channel 0	
	Offset	**Type**	**Element**	**Description**
	0x0008	uint16_t	usNetxFlags	netX flag register
	0x000A	uint16_t	usHostFlags	Host flag register
Offset	**Type**	**Element**	**Description**	
0x000C	NETX_HANDSHAKE_CELL	tCommFlags1	Communication Channel 1	
	Offset	**Type**	**Element**	**Description**
	0x000C	uint16_t	usNetxFlags	netX flag register
	0x000E	uint16_t	usHostFlags	Host flag register

Offset	Type	Element	Description	
0x0010	NETX_HANDSHAKE_CELL	tCommFlags2	Communication Channel 1	
	Offset	**Type**	**Element**	**Description**
	0x0010	uint16_t	usNetxFlags	netX flag register
	0x0012	uint16_t	usHostFlags	Host flag register
Offset	**Type**	**Element**	**Description**	
0x0014	NETX_HANDSHAKE_CELL	tCommFlags3	Communication Channel 3	
	Offset	**Type**	**Element**	**Description**
	0x0014	uint16_t	*usNetxFlags*	netX flag register
	0x0016	uint16_t	*usHostFlags*	Host flag register
Offset	**Type**	**Element**	**Description**	
0x0018	NETX_HANDSHAKE_CELL	tAppFlags0	Application Channel 0 (unused)	
Offset	**Type**	**Element**	**Description**	
0x001E	NETX_HANDSHAKE_CELL	tAppFlags1	Application Channel 1 (unused)	
Offset	**Type**	**Element**	**Description**	
0x0020	uint32_t	aulReserved[56]	Reserved, set to 0	

Table 72: Handshake Channel: Handshake Channel Layout

Structure Reference: *NETX_HANDSHAKE_CHANNEL*

```
typedef struct NETX_HANDSHAKE_CHANNELtag
{
  NETX_HANDSHAKE_CELL        tSysFlags;   /* system handshake flags         */
  NETX_HANDSHAKE_CELL        tHskFlags;   /* synchronization flags          */
  NETX_HANDSHAKE_CELL        tCommFlags0; /* channel 0 handshake flags      */
  NETX_HANDSHAKE_CELL        tCommFlags1; /* channel 1 handshake flags      */
  NETX_HANDSHAKE_CELL        tCommFlags2; /* channel 2 handshake flags      */
  NETX_HANDSHAKE_CELL        tCommFlags3; /* channel 3 handshake flags      */
  NETX_HANDSHAKE_CELL        tAppFlags0;  /* not supported yet              */
  NETX_HANDSHAKE_CELL        tAppFlags1;  /* not supported yet              */
  uint32_t                   aulReserved[56];
} NETX_HANDSHAKE_CHANNEL;
```

Structure Reference: ***NETX_HANDSHAKE_CELL***

```
typedef union NETX_HANDSHAKE_CELLtag
{
  struct
  {
    volatile uint8_t abData[2];        /* Data value, not belonging to handshake */
    volatile uint8_t bNetxFlags;       /* Device status flags (8Bit Mode) */
    volatile uint8_t bHostFlags;       /* Device command flags (8Bit Mode) */
  } t8Bit;

  struct
  {
    volatile uint16_t usNetxFlags;     /* Device status flags (16Bit Mode) */
    volatile uint16_t usHostFlags;     /* Device command flags (16Bit Mode)*/
  } t16Bit;
  volatile uint32_t ulValue;           /* Handshake cell value */
} NETX_HANDSHAKE_CELL;
```

5.4 Communication Channel

The *Communication Channel* contains data structures and information about the fieldbus protocol stack running on a channel.

Note: Offsets are given relative to the start offset of the channel start address.

Structure of the *Communication Channel*

Communication Channel: *NETX_DEFAULT_COMM_CHANNEL / NETX_8K_DPM_COMM_CHANNEL*			
Offset	**Type**	**Name**	**Description**
0x0000	NETX_HANDSHAKE_BLOCK	tReserved	Reserved See section 5.4.1 for details
0x0008	NETX_CONTROL_BLOCK	tControl	Common Control Block See section 5.4.2 for details
0x0010	NETX_COMMON_STATUS_BLOCK	tCommonStatus	Common Status Block See section 5.4.3 for details
0x0050	NETX_EXTENDED_STATUS_BLOCK	tExtendedStatus	Extended Status Block See section 5.4.4 for details
0x0200	NETX_SEND_MAILBOX_BLOCK	tSendMbx	Send Mailbox See section 5.4.5 for details
0x0840	NETX_RECV_MAILBOX_BLOCK	tRecvMbx	Receive Mailbox See section 5.4.5 for details
0x0E80	uint8_t	abPd1Output[64]	High Priority Output Data Image 1 (not supported, yet)
0x0EC0	uint8_t	abPd1Input[64]	High Priority Input Data Image 1 (not supported, yet)
0x0F00	uint8_t	abReserved1[256]	Reserved, set to 0
	16 KB Layout (default layout)		
0x1000	uint8_t	abPd0Output[5760]	Output Data Image 0 See section 5.4.8 for details
0x2680	uint8_t	abPd0Input[5760]	Input Data Image 0 See section 5.4.8 for details
	8 KByte Layout		
0x1000	uint8_t	abPd0Output[1536]	Output Data Image 0 See section 5.4.8 for details
0x1600	uint8_t	abPd0Input[1536]	Input Data Image 0 See section 5.4.8 for details

Table 73: COMM Channel: Communication Channel Layout

```
#define NETX_IO_DATA_SIZE         5760  /* I/O data size in bytes for 16KByte DPM */
#define NETX_IO_DATA_SIZE_8K_DPM  1536  /* I/O data size in bytes for 8KByte DPM  */
#define NETX_HP_IO_DATA_SIZE      64    /* Default size of the high prio I/O data */
```

Structure Reference: Communication Channel (16Kbyte Layout)

```
typedef struct NETX_DEFAULT_COMM_CHANNELtag
{
  NETX_HANDSHAKE_BLOCK           tReserved;
  NETX_CONTROL_BLOCK             tControl;
  NETX_COMMON_STATUS_BLCOK       tCommonStatus;
  NETX_EXTENDED_STATUS_BLOCK     tExtendedStatus;
  NETX_SEND_MAILBOX_BLOCK        tSendMbx;
  NETX_RECV_MAILBOX_BLOCK        tRecvMbx;
  uint8_t                        abPd1Output[NETX_HP_IO_DATA_SIZE];
  uint8_t                        abPd1Input[NETX_HP_IO_DATA_SIZE];
  uint8_t                        abReserved1[256];
  uint8_t                        abPd0Output[NETX_IO_DATA_SIZE];
  uint8_t                        abPd0Input[NETX_IO_DATA_SIZE];
} NETX_DEFAULT_COMM_CHANNEL;
```

Structure Reference: Communication Channel (8Kbyte Layout)

```
typedef struct NETX_8K_DPM_COMM_CHANNELtag
{
  NETX_HANDSHAKE_BLOCK           tReserved;
  NETX_CONTROL_BLOCK             tControl;
  NETX_COMMON_STATUS_BLCOK       tCommonStatus;
  NETX_EXTENDED_STATUS_BLOCK     tExtendedStatus;
  NETX_SEND_MAILBOX_BLOCK        tSendMbx;
  NETX_RECV_MAILBOX_BLOCK        tRecvMbx;
  uint8_t                        abPd1Output[NETX_HP_IO_DATA_SIZE];
  uint8_t                        abPd1Input[NETX_HP_IO_DATA_SIZE];
  uint8_t                        abReserved1[256];
  uint8_t                        abPd0Output[NETX_IO_DATA_SIZE_8K_DPM];
  uint8_t                        abPd0Input[NETX_IO_DATA_SIZE_8K_DPM];
} NETX_8K_DPM_COMM_CHANNEL;
```

5.4.1 Channel Handshake Block

The *Channel Handshake Block* is a reserved area which is intended to hold the channel handshake register and may be used in the future.

Note: Currently the handshake registers of all channels are located in the *Handshake Channel* of the DPM.

The *Channel Handshake Block* inside a channel is not yet supported and therefore set to zero.

System Handshake Block: *NETX_HANDSHAKE_CELL*			
Offset	**Type**	**Name**	**Description**
0x0000	uint16_t	t16Bit.usNetxFlags	netX channel handshake register
0x0003	uint16_t	t16Bit.usHostFlags	Host channel handshake register

Table 74: COMM Channel: System Handshake Block

Structure Reference - Channel Handshake Block

```
/*****************************************************************************/
/*! Handshake cell definition                                                */
/*****************************************************************************/
typedef __RCX_PACKED_PRE union NETX_HANDSHAKE_CELLtag
{
  __RCX_PACKED_PRE struct
  {
    volatile uint8_t abData[2];        /*!< Data value, not belonging to handshake */
    volatile uint8_t bNetxFlags;       /*!< Device status flags (8Bit Mode) */
    volatile uint8_t bHostFlags;       /*!< Device command flags (8Bit Mode) */
  } __RCX_PACKED_POST t8Bit;

  __RCX_PACKED_PRE struct
  {
    volatile uint16_t usNetxFlags;     /*!< Device status flags (16Bit Mode) */
    volatile uint16_t usHostFlags;     /*!< Device command flags (16Bit Mode)*/
  } __RCX_PACKED_POST t16Bit;
  volatile uint32_t ulValue;            /*!< Handshake cell value */
} NETX_HANDSHAKE_CELL;
```

5.4.2 Common Control Block

The control block of the communication channel holds a command cell (*ulApplicationCOS*), which can be passed to the protocol stack by using the COS (Change of State) mechanism and a watchdog element, allowing a netX firmware to supervise the host application and vice versa.

Control Block: *NETX_CONTROL_BLOCK*			
Offset	**Type**	**Name**	**Description**
0x0008	uint32_t	ulApplicationCOS	Application Change Of State - READY - BUS ON - INITIALIZATION - LOCK CONFIGURATION
0x000C	uint32_t	ulDeviceWatchdog	Device Watchdog Watchdog counter necessary for the handling and supervision (see page 102)

Table 75: COMM Channel: Communication Control Block

Structure Reference - Communication Control Block

```
typedef struct NETX_CONTROL_BLOCKtag
{
  uint32_t   ulApplicationCOS;
  uint32_t   ulDeviceWatchdog;
} NETX_CONTROL_BLOCK;
```

Application Change of State Register

The *Application Change of State* (*ulApplicationCOS*) is a bit field. The host application uses this field in order to send commands to the communication channel synchronized by the COS mechanism described in section 4.3.

***ulApplicationCOS* – Host writes, netX reads**

31	…	9	8	7	6	5	4	3	2	1	0

- 0: RCX_APP_COS_APPLICATION_READY
- 1: RCX_APP_COS_BUS_ON
- 2: RCX_APP_COS_BUS_ON_ENABLE
- 3: RCX_APP_COS_INITIALIZATION
- 4: RCX_APP_COS_INITIALIZATION_ENABLE
- 5: RCX_APP_COS_LOCK_CONFIGURATION
- 6: RCX_APP_COS_LOCK_CONFIGURATION_ENABLE
- 7: RCX_APP_COS_DMA
- 8: RCX_APP_COS_DMA_ENABLE
- 9 … 31: unused, set to zero

Table 76: COMM Channel: Application Change of State

Application Change of State Flags (Application ⇨ netX System)

Bit No.	Description
0	**APPLICATION READY** Flag (RCX_APP_COS_APPLICATION_READY) Application ready is used to signal a protocol stack a host application is working with the DPM. 0 = Host application not Ready 1 = Host application Ready
1	**BUS ON** Flag (RCX_APP_COS_BUS_ON) The *Bus On* flag is used to signal a protocol stack to start/stop communication on the fieldbus network. 0 = *Bus OFF*, network communication is inactive (stopped) 1 = *Bus ON,* network communication is active (should be activated) **NOTE** This flag is used in conjunction with the protocol stack configuration which can be set to "manual start of the network communication".
2	**BUS ON ENABLE** Flag (RCX_APP_COS_BUS_ON_ENABLE) The *Bus On Enable* flag defines if the RCX_APP_COS_BUS_ON flag will be evaluate and executed by the protocol stack. 0 = RCX_APP_COS_BUS_ON flag evaluation disabled 1 = RCX_APP_COS_BUS_ON flag evaluation enabled
3	**INITIALIZATION** Flag (RCX_APP_COS_INITIALIZATION) The *Initialization* flag is used to re-initialize a protocol stack. If the command is recognized, all network connections are closed immediately and restarted by using the available configuration. 0 = No re-initialization 1 = Re-initialization activated **NOTE** If the database is locked by RCX_APP_COS_LOCK_CONFIGURATION, re-initializing the channel is not allowed and rejected by the protocol stack.
4	**INITIALIZATION ENABLE** Flag (RCX_APP_COS_INITIALIZATION_ENABLE) The *Initialization Enable* flag is used to enable the evaluation of RCX_APP_COS_INITIALIZATION flag. 0 = RCX_APP_COS_INITIALIZATION flag evaluation disabled 1 = RCX_APP_COS_INITIALIZATION flag evaluation enabled
5	**LOCK CONFIGURATION** Flag (RCX_APP_COS_LOCK_CONFIGURATION) If this bit is set, the protocol stack configuration is locked and the host system does not allow the firmware to reconfigure the communication channel. 0 = Configuration is unlocked, reconfiguration of the stack allowed 1 = Configuration is locked, reconfiguration is not allowed
6	**LOCK CONFIGURATION ENABLE** Flag (RCX_APP_COS_LOCK_CONFIGURATION_ENABLE) The *Lock Configuration Enable* flag is used to enable the evaluation of the RCX_APP_COS_LOCK_CONFIGURATION flag. 0 = RCX_APP_COS_LOCK_CONFIGURATION flag evaluation is disabled 1 = RCX_APP_COS_LOCK_CONFIGURATION flag evaluation is enabled
7	**DMA MODE** Flag (RCX_APP_COS_DMA) The host system sets this flag in order to turn on DMA (Direct Memory Access) transfer of the cyclic process data image between the host and the netX hardware. 0 = DMA mode disabled 1 = DMA mode enabled **Note:** DMA mode only available on PCI and PCIe based hardware (e.g. CIFX50 / CIFX50e)

8	**DMA MODE ENABLE** Flag (RCX_APP_COS_DMA_ENABLE) The *DMA Mode Enable* flag is used to enable the evaluation of the RCX_APP_COS_DMA flag. 0 = RCX_APP_COS_DMA flag evaluation is disabled 1 = RCX_APP_COS_DMA flag evaluation is enabled
9 … 31	Reserved, set to 0

Table 77: COMM Channel: Application Change of State Description

Device Watchdog

The device watchdog counter *ulDeviceWatchdog* is one part of the communication channel watchdog mechanism. The second part *ulHostWatchdog* is located in the status block of the channel.

5.4.3 Common Status Block

The common status block contains information common to all fieldbus protocol stacks and is always present.

Common Status Block: *NETX_COMMON_STATUS_BLOCK*				
Offset	**Type**	**Name**	**Description**	
0x0010	uint32_t	ulCommunicationCOS	Communication Change of State - READY / RUN - RESET REQUIRED / NEW CONFIG AVAILABLE - CONFIG LOCKED	
0x0014	uint32_t	ulCommunicationState	Communication State - OFFLINE / STOP / IDLE / OPERATE	
0x0018	uint32_t	ulCommunicationError	Communication Error Protocol Stack error number	
0x001C	uint16_t	usVersion	Version Version Number of this structure (e.g. 0x0002)	
0x001E	uint16_t	usWatchdogTime	Watchdog Time Configured watchdog time given in milliseconds [ms]	
0x0020	uint8_t	bPDInHskMode	Handshake Mode Configured input process data handshake mode	
0x0021	uint8_t	bPDInSource	Input Handshake Event Source Reserved, set to zero	
0x0022	uint8_t	bPDOutHskMode	Handshake Mode Configured output process data handshake mode	
0x0023	uint8_t	bPDOutSource	Output Handshake Event Source Reserved, set to zero	
0x0024	uint32_t	ulHostWatchdog	Host Watchdog Host watchdog counter used for watchdog handling	
0x0028	uint32_t	ulErrorCount	Error Count Total Number of Detected Errors Since Power-Up or Reset (see page 109)	
0x002C	uint8_t	bErrorLogInd	Number of Entries in the internal error log Not supported yet	
0x002D	uint8_t	bErrorPDInCnt	Input process data handshake error counter	
0x002E	uint8_t	bErrorPDOutCnt	Output process data handshake error counter	
0x002F	uint8_t	bErrorSyncCnt	Synchronization handshake error counter	
0x0030	uint8_t	bSyncHskMode	Synchronization Handshake Mode	
0x0031	uint8_t	bSyncSource	Synchronization Source	
0x0032	uint16_t	ausReserved[3]	Reserved Set to 0	
0x0038	**union uStackDepended**		Common state information master protocol stacks	
	NETX_MASTER_STATUS	tMasterStatusBlock	Common master protocol stack state information	
	uint32_t	aulReserved[6]	Reserved, set to zero for slave protocols	

Table 78: COMM Channel: Common Status Block

Structure Reference - Common Status Block

```
typedef struct NETX_COMMON_STATUS_BLOCKtag
{
  uint32_t   ulCommunicationCOS;
  uint32_t   ulCommunicationState;
  uint32_t   ulCommunicationError;
  uint16_t   usVersion;
  uint16_t   usWatchdogTime;
  uint8_t    bPDInHskMode;
  uint8_t    bPDInSource;
  uint8_t    bPDOutHskMode;
  uint8_t    bPDOutSource;
  uint32_t   ulHostWatchdog;
  uint32_t   ulErrorCount;
  uint8_t    bErrorLogInd;
  uint8_t    bErrorPDInCnt;
  uint8_t    bErrorPDOutCnt;
  uint8_t    bErrorSyncCnt;
  uint8_t    bSyncHskMode;
  uint8_t    bSyncSource;
  uint16_t   ausReserved[3];
  union
  {
    NETX_MASTER_STATUS  tMasterStatusBlock;  /* for master implementation    */
    uint32_t            aulReserved[6];      /* otherwise reserved           */
  } uStackDepended;
} NETX_COMMON_STATUS_BLOCK;
```

Communication Change of State Register

The *Communication Change of State* register is a bit field, containing information about the current operating status of the communication channel and its firmware.

The state information is exchanged between the netX firmware and the host by using the COS (Change of State) mechanism described in section 4.3.

ulCommunicationCOS **– netX writes, Host reads**

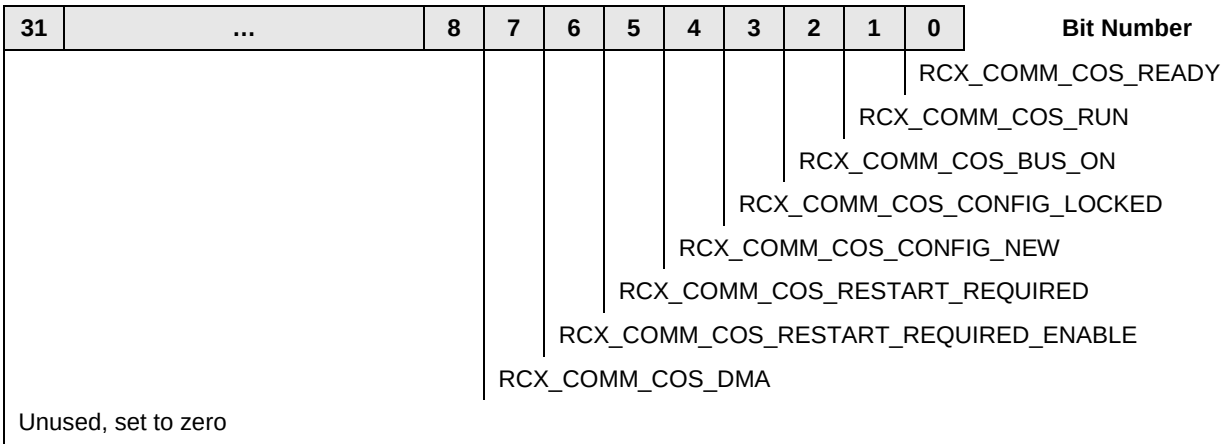


Table 79: COMM Channel: Communication State of Change Register

Communication Change of State Flags (netX System ⇨ Application)

Bit No.	Definition / Description
0	**READY** Flag (RCX_COMM_COS_READY) The *READY* flag indicates if the protocol stack on the given channel is started properly. In this state the stack is ready to accept a configuration or other commands from the host application. 0 = not *READY* (protocol stack not started/working) 1 = *READY* (protocol stack is started)
1	**RUNNING** Flag (RCX_COMM_COS_RUN) The *RUNNING* flag indicates a configured protocol stack, able to start a network communication. Only if both, the *READY* and *RUNNING* flag are set, the protocol stack is started and configured. 0 = not *RUNNING* (not configured) 1 = *RUNNING* (protocol stack is configured) **NOTE** The fieldbus configuration defines if a *READY* and *RUNNING* protocol stack will automatically start the network communication.
2	**BUS ON** Flag (RCX_COMM_COS_BUS_ON) The *BUS ON* flag indicates the actual state of the fieldbus network communication. 0 = *Bus OFF* (fieldbus communication not started) 1 = *Bus ON* (fieldbus communication started) The fieldbus configuration defines if the network communication will start automatically or if it has to be started by the host application (see *COMMON_CONTROL_BLOCK->ulApplicationCOS*). Also a wrong configuration may prevent the start of the network communication.

3	**CONFIGURATION LOCKED** Flag (RCX_COMM_COS_CONFIG_LOCKED) The *CONFIGURATION LOCKED* flag indicates if the fieldbus configuration is protected and the protocol stack is not allowed to execute a re-initialization with another configuration. 0 = Configuration not locked 1 = Configuration is locked Locking is controlled by the host application (see *COMMON_CONTROL_BLOCK->`ulApplicationCOS`*). **Note:** An application has to make sure to disable the configuration locking before executing a channel reset or re-initialization (see *COMMON_CONTROL_BLOCK->`ulApplicationCOS`*)
4	**CONFIGURATION NEW** Flag (RCX_COMM_COS_CONFIG_NEW) The *CONFIGURATION NEW* flag is set by the protocol stack to indicate that a new configuration became available, which has not been activated yet. 0 = No new configuration available 1 = A new configuration is available This flag may be set together with the *RESTART REQUIRED* flag.
5	**RESTART REQUIRED** Flag (RCX_COMM_COS_RESTART_REQUIRED) The *RESTART REQUIRED* flag is set by the protocol stack as an indication of a changed configuration, either received by a host download or an upload via the fieldbus network. A new configuration will only be activated if the host restarts the protocol stack in such a case. 0 = No restart required, no new configuration 1 = Restart required, new configuration available This flag is used together with the *RESTART REQUIRED ENABLE* flag.
6	**RESTART REQUIRED ENABLE** Flag (RCX_COMM_COS_RESTART_REQUIRED_ENABLE) The *RESTART REQUIRED ENABLE* flag enables the evaluation of the *RESTART REQUIRED* flag by the host. 0 = Restart disabled 1 = Restart enabled
7	**DMA Mode** Flag (RCX_COMM_COS_DMA) The protocol stack sets this flag in order to signal the host application that DMA (Direct Memory Access) mode is turned on and used to transfer the cyclic process data images between the host system and the netX firmware. 0 = DMA mode off (default) 1 = DMA mode on **Note:** DMA mode is only available on PCI and PCIe based hardware (e.g. CIFX50 / CIFX50e)
8 … 31	Reserved, set to 0

Table 80: COMM Channel: Communication State of Change Description

Communication State

The *Communication State* field *ulCommunicationState* contains information about the current network state.

Note: Depending on fieldbus protocol, not all of the defined states are always available or may have different meanings.

Value	Definition	General Description
0x00000000	Unknown	Current network state is unknown
0x00000001	Offline	No valid configuration / no network communication
0x00000002	Stop	Communication stopped by the user application or an error during the network communication.
0x00000003	Idle	Protocol stack is configured but has not reached operating state. No cyclic data exchanged on the bus system
0x00000004	Operate	Network communication is active, data exchange on the network is activated
Other values are reserved		

Table 81: COMM Channel: Communication State

Communication Channel Error

ulCommunicationError holds the current error code of the communication channel (protocol stack). An error is also indicated by the *NCF_ERROR* flag in the communication channel handshake register.

If the cause of the error is resolved, the communication error field is set to 0 again.

Note: Error codes can be protocol specific and are described in the corresponding manual. Default system errors are listed in section 7.

Structure Version

The version field *usVersion* holds the actual version number of the *Common Status Block* structure. The structure number is used to indicate changes in the structure.

Value	Definition / Description
0x0000	Undefined
0x0001	First version of the structure layout
0x0002	Second version of the structure layout (**current**) The layout was extended by the following data *bPDInHskMode*, *bPDInSource*, *bPDOutHskMode*, *bPDOutSource* *bErrorLogInd*, *bErrorPDInCnt*, *bErrorPDOutCnt*, bErrorSyncCnt bSyncHskMode, bSyncSource

Table 82: COMM Channel: Common Status Block Structure Version

Watchdog Timeout

The watchdog timeout field *usWatchdogTime* holds the configured watchdog timeout value in milliseconds [ms]. This value is set by the fieldbus configuration (default = 1000ms).

The application can use the value to setup their watchdog trigger interval accordingly. If the application has activated the watchdog it must trigger the watchdog at least once during the given watchdog time. Otherwise the protocol stack will interrupt all network connections immediately regardless of their current state.

A watchdog fault will be indicated by the *NCF_ERROR* flag and a corresponding error is inserted in *ulCommunicationError*.

Handshake Mode / Sources and Error Counters

Input and output data images can be driven in different handshake modes, while each mode also allows configuring a so called synchronization source (not supported yet) and offers an error handshake counter.

The following elements are used to indicate the actual configured mode / synchronization source and handshake error for each image.

For details of the handshake mechanism refer to section 4.2.

Image	Type	Variable	Description
INPUT	uint8_t	bPDInHskMode	Input data image Handshake mode Indicates the actual configured handshake mode for the input data image
	uint8_t	bPDInSource	Input data synchronization source definition Is intended to be used as an indicator of the actual configured input data synchronization source. This must be supported by the fieldbus protocol stack.
	uint8_t	bErrorPDInCnt	Input data handshake errors Depending on the configured handshake mode, it is possible the protocol stack is not able to signal new input data, because the host application has not acknowledged a previous state and therefore access to the image is not allowed by the stack. In this case, the protocol stack would increment this counter to signal a missed update.
OUTPUT	uint8_t	bPDOutHskMode	Output data image Handshake mode Indicates the actual configured handshake mode for the output data image
	uint8_t	bPDOutSource	Output data synchronization source definition Is intended to be used as an indicator of the actual configured output data synchronization source. This must be supported by the fieldbus protocol stack.
	uint8_t	bErrorPDOutCnt	Output data handshake errors Depending on the configured handshake mode, it is possible the protocol stack needs new output data for the next network transfer. If the host application is not quick enough to deliver the data, the protocol stack would increment this counter to signal the missing update.

Table 83: COMM Channel: Handshake Mode

Host Watchdog

The host watchdog field *ulHostWatchdog* is used together with the *Common Control Block* *ulDeviceWatchdog* field for the handling of the netX watchdog functionality.

For details on the watchdog function, refer to section 6.5.4 and 5.4.3.

Error Count

This *ulErrorCount* 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 (not supported yet)

The error log indicator field *bErrorLogInd* is created for later use, when the netX firmware supports an internal error logging array. In this case, the field will show the number of entries in the logging array and if all entries are read from the log, the field is set to zero.

Extended Synchronization

A protocol stack may offer additional synchronization options, not depending on the input / output data transfer synchronization and handled by dedicated synchronization flags.

Note: The extended synchronization mechanism is described in an own manual and must be supported by the fieldbus protocol stack.

These options could be the synchronization with the network bus cycle or an external hardware trigger. For such options the following three elements are defined to indicate the actual configuration, the source of the sync signal and an error counter.

Type	Variable	Description
uint8_t	bErrorSyncCnt	Number of synchronization handshake errors Depending on the configuration the error counter is used if the sync information could not be updated because of a missing acknowledgement.
uint8_t	bSyncHskMode	Synchronization Handshake Mode Configured mode of the synchronization (host controlled / device controlled)
uint8_t	bSyncSource	Synchronization Source Definition of the sync source (bus cycle / hardware trigger etc.)

Table 84: COMM Channel: Extended Synchronization

5.4.3.1 Master State Information

The master state information structure *NETX_MASTER_STATUS* offers common information for all master protocol stacks.

Note: This structure is not available for slave protocols and set to zero.

Master Status: *NETX_MASTER_STATUS*			
Start Offset	**Type**	**Name**	**Description**
0x0038	uint32_t	ulSlaveState	Common Slave State 0 = RCX_SLAVE_STATE_UNDEFINED 1 = RCX_SLAVE_STATE_OK 2 = RCX_SLAVE_STATE_FAILED
0x003C	uint32_t	ulSlaveErrLogInd	Number of entries in the internal error log array (not supported yet) 0 = no error entry
0x0040	uint32_t	ulNumOfConfigSlaves	Number of configured slave devices in the master configuration. 0 = no configured slaves
0x0044	uint32_t	ulNumOfActiveSlaves	Number of activated slave devices, the master has an open connection to. 0 = no active slaves
0x0048	uint32_t	ulNumOfDiagSlaves	Number of slaves reporting diagnostic issues 0 = no slaves with diagnostic information
0x004C	uint32_t	ulReserved	Reserved, set to 0

Table 85: COMM Channel: Master State Information

Master Status Structure Reference

```
typedef struct NETX_MASTER_STATUStag
{
  uint32_t  ulSlaveState;          /* slave state                  */
  uint32_t  ulSlaveErrLogInd;      /* slave error log Indicator    */
  uint32_t  ulNumOfConfigSlaves;   /* number of configured slaves  */
  uint32_t  ulNumOfActiveSlaves;   /* number of activated slaves   */
  uint32_t  ulNumOfDiagSlaves;     /* number of faulted slaves     */
  uint32_t  ulReserved;            /* */
} NETX_MASTER_STATUS;
```

Common Slave State

The *Common Slave State* field *ulSlaveState* is a collective indication on whether the master is in cyclic data exchange to all configured slaves or not.

If there is at least one slave missing or one of the slaves has pending diagnostic information, the status is changed to *RCX_SLAVE_STATE_FAILED*.

For protocols that only support packet based (non-cyclic) communication, the slave state is set to RCX_SLAVE_STATE_OK as soon as a valid configuration is found.

Slave Error Log Indicator

Not supported yet and reserved for later use if the firmware supports an internal error log array. The field will indicate the number of entries in the log array and is set to zero if all log entries have been read by the host application.

Number of Configured Slaves

ulNumOfConfigSlaves indicates number of slave devices configured in the master configuration.

Number of Active Slaves

ulNumOfActiveSlaves indicates the number of slaves, the master has an active communication to. Ideally this number is equal to the number of configured slaves, if all configured slaves are connected to the networks and working without problems.

For certain fieldbus systems, it could be possible that a slave is shown as active, but still has a problem in terms of a diagnostic issue.

Number with Diagnostic Information

ulNumOfDiagSlaves indicates how many slaves are missing on the network or reporting a diagnostic issue.

Slave diagnostic is reset if it was read by the master or host application. If all configured slaves are on the network and no more diagnostic information is pending, the field is set to zero.

5.4.4 Extended Status Block

The *Extended Status Block* contains protocol stack specific information, not common to all fieldbus protocols, given in *abExtendedStatus[]* and, if supported by the protocol stack, additional definitions in a structured descriptor table *tExtStateField*.

The idea behind the descriptor table is the free definition of memory areas inside the input/output image of the process data. This allows an application to parse the descriptor table, if available, and locating additional information inside the input /output image areas.

An example for such additional information is the *List of Configured Slaves (Bit Field).*

Overview Extended State Field Structure

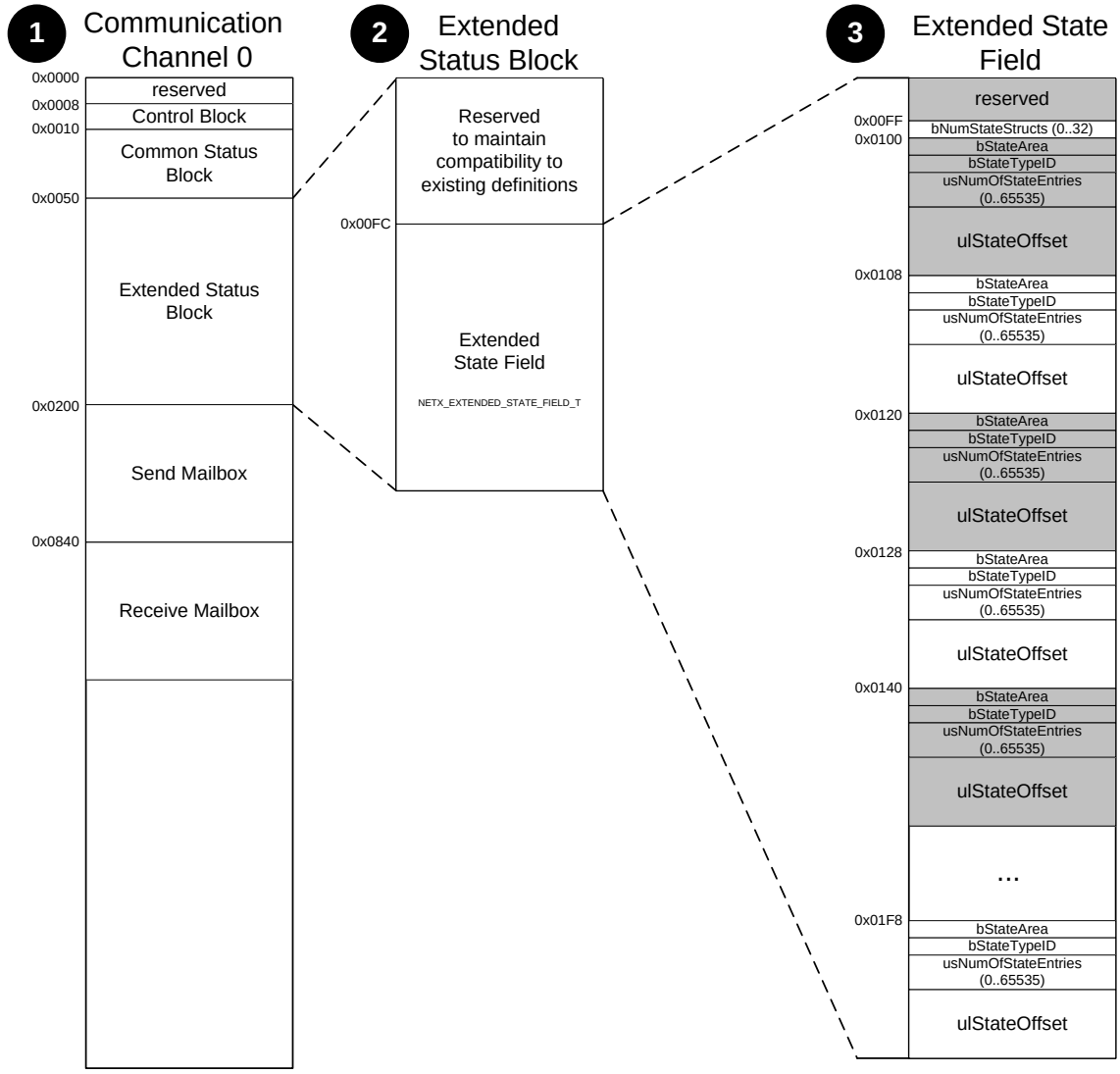


Figure 16: COMM Channel: Overview - Extended State Field Structure

❶ DPM structure of the communication channel

❷ Extended status block inside the communication channel

❸ Organization of the extended state field structure

Extended Status Block

The *Extended Status Block* is a byte array of 432 byte. This array is divided into the *Extended Status Area*, which is protocol specific, and the *Extended State Field.*

Extended Status Block: *NETX_EXTENDED_STATUS_BLOCK*			
Offset	**Type**	**Name**	**Description**
0x0050	uint8_t	abExtendedStatus[NETX_EXT_STATE_SIZE];	Extended Status Block Protocol stack specific status area (max. 432byte)
… 0x01FF	Unused Space is Set to Zero		

Table 86: COMM Channel: Extended Status Block Definition

Structure Reference - Extended Status

```
typedef struct NETX_EXTENDED_STATUS_BLOCKtag
{
  uint8_t abExtendedStatus[NETX_EXT_STATE_SIZE];         /* Extended status buffer */
} NETX_EXTENDED_STATUS_BLOCK;
```

Extended Status Area

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

Note: Depending on the protocol stack, the status block is supported or not and it is defined in the specific protocol stack manual.

Structure Reference - Extended Status Block / Extended State Field Definition

Extended State Field Definition: *NETX_EXTENDED_STATE_FIELD_DEFINITION_T*			
Offset	**Type**	**Name**	**Description**
0x0050	uint8_t	abReserved[172]	Extended Status Area Protocol stack specific status area
0x00FC	NETX_EXTENDED_STATE_FIELD_T	tExtStateField	Extended State Field Structure
… 0x01FF	Unused Space is Set to Zero		

Table 87: COMM Channel: Extended Status Block Structure

```
typedef struct NETX_EXTENDED_STATE_FIELD_DEFINITION_Ttag
{
  uint8_t  abReserved[172];                    /* Protocol specific information area */
  NETX_EXTENDED_STATE_FIELD_T   tExtStateField; /* Extended status structures */
} NETX_EXTENDED_STATE_FIELD_DEFINITION_T;
```

Extended State Field Structure

The *Extended State Field Structure* is a collection of descriptor definitions, where each entry describes the content and the size of additional memory areas in the DPM.

This location is used to maintain various protocol, device or implementation specific state fields. The *Status Structures* contain definitions in terms of type, number of valid entries and start offset of these fields in the communication channel.

If the status information is configured to be located in the IO data image of a channel, the status information and the IO data image are consistent and updated together.

Structure Reference - Extended Status Block / Extended State Field Structure

Extended State Field: *NETX_EXTENDED_STATE_FIELD_T*			
Offset	**Type**	**Name**	**Description**
0x00FC	uint8_t	abReserved[3]	Reserved, set to zero
0x00FF	uint8_t	bNumOfStateStruct	Number of Status Structures Number of valid structure definitions in the following *atStateStruct[]* array
0x0100	NETX_EXTENDED_STATE_STRUCT_T	atStateStruct[24]	Status structure with a maximum of RCX_EXT_STS_MAX_STRUCTURES (24) elements
… 0x01FF	Unused Space is Set to Zero		

Table 88: COMM Channel: Extended State Field Structure

```
typedef struct NETX_EXTENDED_STATE_FIELD_Ttag
{
  uint8_t    bReserved[3];         /* 3 Bytes preserved. Not used.  */
  uint8_t    bNumStateStructs;     /* Number of following state structures */
  NETX_EXTENDED_STATE_STRUCT_T  atStateStruct[RCX_EXT_STS_MAX_STRUCTURES];
} NETX_EXTENDED_STATE_FIELD_T;
```

Number of Status Structures

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

Extended State Structure

Each element of the *Extended State Structure* array (*atStateStruct[]*) describes exactly one memory region.

Structure Reference - Extended Status Block / Extended State Structure

Extended Status Structure: *NETX_EXTENDED_STATE_STRUCT_T*			
Offset	**Type**	**Name**	**Description**
0x0100	uint8_t	bStateArea	State Area Defines the memory location where the state information can be found
0x0101	uint8_t	bStateTypeID	State Type Identification Type of the state information
0x0102	uint16_t	usNumOfStateEntries	Number of State Entries
0x0104	uint32_t	ulStateOffset	State Offset Start of the information in the given state area
… 0x01FF	Unused Space is Set to Zero		

Table 89: COMM Channel: Extended State Structure

```
typedef struct NETX_EXTENDED_STATE_STRUCT_Ttag
{
  uint8_t    bStateArea;           /* Location of the ext. state information */
  uint8_t    bStateTypeID;         /* Type of the state information */
  uint16_t   usNumOfStateEntries;  /* Number of state entries of the type bStateTypeID */
  uint32_t   ulStateOffset;        /* Byte start offset in the defined I/O area*/
} NETX_EXTENDED_STATE_STRUCT_T;
```

State Area

bStateArea defines the memory location where this state information can be found.

Value	**Definition**	**State field located**
0	RCX_EXT_STS_STD_INPUT_BLK_ID	In standard input data area
1	RCX_EXT_STS_HI_INPUT_BLK_ID	In high priority input data area (not supported yet)
8	RCX_EXT_STS_STD_OUTPUT_BLK_ID	In standard output area
9	RCX_EXT_STS_HI_OUTPUT_BLK_ID	In high priority output data area (not supported yet)
Other values are reserved		

Table 90: COMM Channel: Extended State Area

State Type ID

The *State Type ID* (*bStateTypeID*) indicates the type of the status information. It implicitly defines the content and the size of the information. This could be a list of one or more bits or even bytes per IO data unit, corresponding to the state definition of the specific protocol.

The following types of state information are defined. The complete list of supported types can be found in the protocol API manual of the used protocol.

Value	Definition	Description
1	RCX_EXT_STS_SLAVE_CONFIGURED	Configured slave bit field
2	RCX_EXT_STS_SLAVE_ACTIV	Active slave bit field
3	RCX_EXT_STS_SLAVE_DIAGNOSTIC	Slave diagnostic bit field
Other values are reserved		

Table 91: COMM Channel: Extended State Type ID

Note: Not all of the status types are supported by every protocol stack.

Number of State Entries

usNumOfStateEntries holds the number of bytes provided in the status information field. This value is zero, if no state entries are provided and the field can hold up to 65535 entries.

State Offset

ulStateOffset holds the start offset of the status information in the defined memory area (input/output image). The offset is related to the beginning of the defined data area and given in bytes.

Note: The state information is always aligned to 32 bit offsets (round up to the next double word).

Example

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

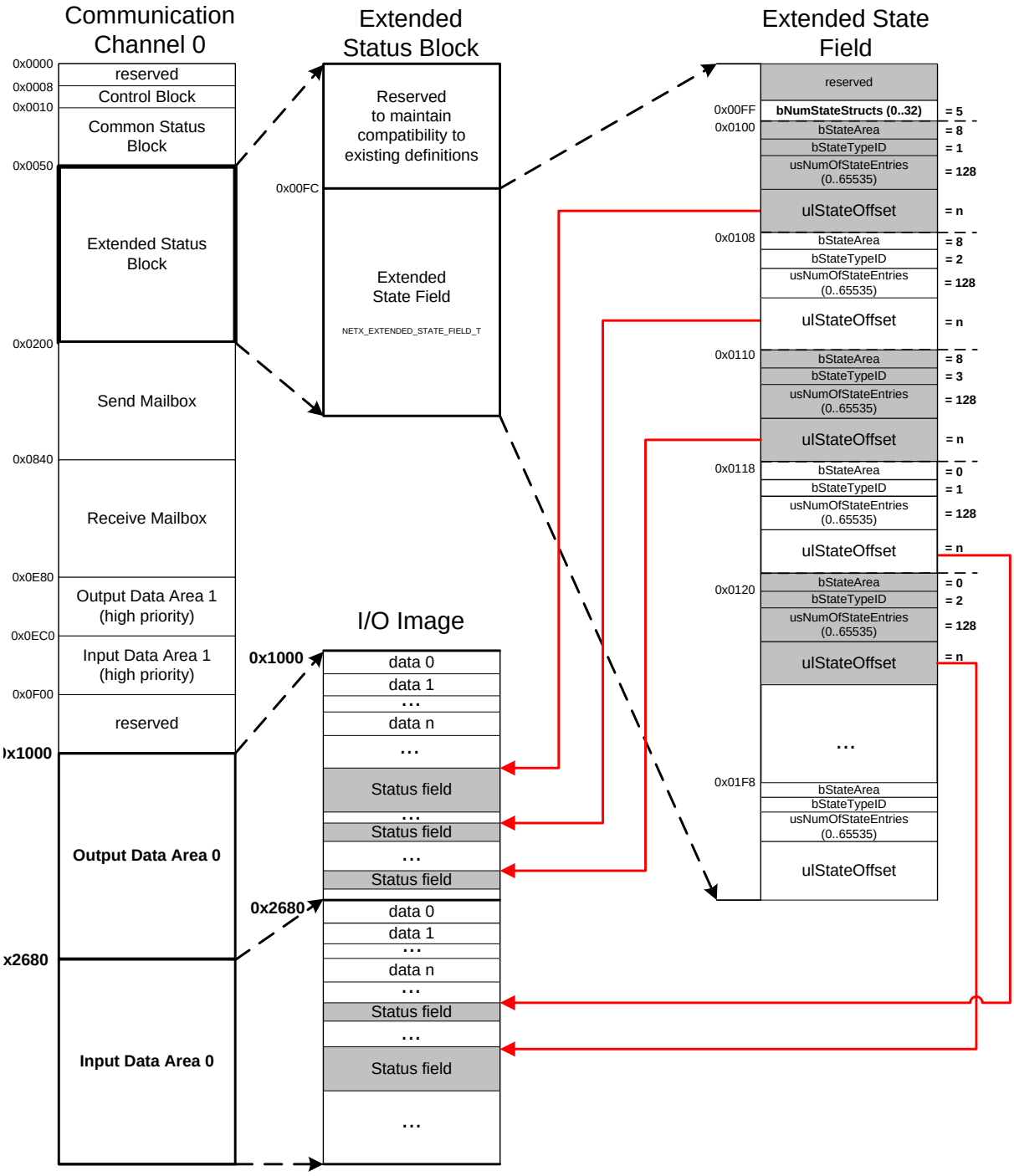


Figure 17: COMM Channel: Example Extended Status Structures

The first entry in the *Extended State Field* structure indicates that there is a status field located in the output data image (*bStateArea* = 8) and *ulStateOffset* points to a location within the output image. *bStateTypeID* holds the type of information located in the output data image (list of configured/activated/faulted slaves). The second entry, points to the location in the output data image, and so on.

5.4.5 Channel Mailbox

The channel mailbox follows the definition for a mailbox system described in section 4.1. Send and receive mailbox areas are used to exchange non-cyclic, packet based data with the communication channel (netX firmware / fieldbus protocols).

The **send** mailbox is used to transfer data **to** the netX firmware or **to** the protocol stack. The **receive** mailbox is used to receive packet based data **from** the netX firmware or **from** the protocol stack.

A send/receive mailbox is always available in the communication channel and the default user data size inside the send and receive mailbox is 1596 byte.

The handling of the mailbox system is described in section 4.1.

Channel Send Mailbox: *NETX_SEND_MAILBOX_BLOCK*			
Direction: Host System ⇨ netX			
Offset	**Type**	**Name**	**Description**
0x0200	uint16_t	usPackagesAccepted	Packages Acceptable Number of packets that can be accepted by the firmware
0x0202	uint16_t	usReserved	Reserved, set to 0
0x0204 … 0x023F	uint8_t	abSendMbx[1596]	Send Mailbox Buffer Buffer to insert the send packet

Table 92: COMM Channel: Channel Mailbox - Send Mailbox

Channel Receive Mailbox: *NETX_RECV_MAILBOX_BLOCK*			
Offset	**Type**	**Name**	**Description**
0x0840	uint16_t	usWaitingPackages	Waiting Packages Counter of packets waiting to be read by the host
0x0842	uint16_t	usReserved	Reserved, set to 0
0x0844 … 0x0E7F	uint8_t	abRecvMbx[1596]	Receive Mailbox Buffer Buffer containing a packet received from the firmware

Table 93: COMM Channel: Channel Mailbox - Receive Mailbox

Channel Mailboxes Structure Reference

```
typedef struct NETX_SEND_MAILBOX_BLOCKtag
{
  uint16_t   usPackagesAccepted;
  uint16_t   usReserved;
  uint8_t    abSendMbx[1596];
} NETX_SEND_MAILBOX_BLOCK;
```

```
typedef struct NETX_RECV_MAILBOX_BLOCKtag
{
  uint16_t   usWaitingPackages;
  uint16_t   usReserved;
  uint8_t    abRecvMbx[1596];
} NETX_RECV_MAILBOX_BLOCK;
```

5.4.6 High Priority Input/Output Data Image

These areas are currently not supported.

The high priority input and output areas are intended to divide standard cyclic I/O data from I/O data which should be transferred with a higher priority.

Both blocks are reserved and always present in the default memory map.

High Priority Input / Output Data Image			
Offset	**Type**	**Name**	**Description**
0x0E80	uint8_t	`abPd1Output[64]`	High Priority Output Data Image (not supported yet)
0x0EC0	uint8_t	`abPd1Input[64]`	High Priority Input Data Image (not supported yet)

Table 94: COMM Channel: High Priority Input / Output Data Image

5.4.7 Reserved Area

This area is reserved for later use and always available in the default memory map.

Reserved Area			
Offset	**Type**	**Name**	**Description**
0x0F00	uint8_t	`abReserved1[256]`	Reserved Set to 0

Table 95: COMM Channel: Reserved Area

5.4.8 Input / Output Process Data Image

The input and output data blocks are used by fieldbus protocols that support cyclic data transfer. 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.

Note: In case of a network fault (e.g. disconnected network cable), a slave firmware keeps the last state of the input and output data and clears the *Communicating* flag in netX communication flags (see section 3.3).
In this case the input data should not be evaluated, while output data can be written.

The handling and synchronization of the input/output data areas is described in section 4.2.

Default Memory Map

The size of the input and output data image in the default memory map is 5760 byte each.

Input / Output Process Data Image			
Offset	**Type**	**Name**	**Description**
0x1000	uint8_t	`abPd0Output[5760]`	Output Data Image Cyclic Data **To** The Network
0x2680	uint8_t	`abPd0Input[5760]`	Input Data Image Cyclic Data **From** The Network

Table 96: COMM Channel: Input/Output Process Data Image

8 KByte Memory Layout

The size of the input and output data image for the 8 Kbyte layout is 1536 bytes each.

Input / Output Process Data Image (8 KByte)			
Offset	**Type**	**Name**	**Description**
0x1000	uint8_t	`abPd0Output[1536]`	Output Data Image Cyclic Data **To** The Network
0x1600	uint8_t	`abPd0Input[1536]`	Input Data Image Cyclic Data **From** The Network

Table 97: COMM Channel: Input/Output Process Data Image (8 KByte)

5.5 Application Channel

The application channel is reserved for user specific implementations and is not yet supported.

6 System Behavior and Services

6.1 Timing Considerations

The netX firmware and the handling of the handshake registers must meet some general timing requirements to allow a host system a correct detection of state changes and therefore a correct monitoring of the firmware behavior.

During a reset of the netX chip / firmware, the DPM will be re-initialized and this leads into invalid data in the DPM for a certain time.

The following list will give an overview of times which have to be considered.

ROM Loader		**Time**
Start-up / Restart	Time until DPM content valid	< 10ms
2nd Stage Loader	Time until DPM content valid	< 500 ms
Protocol Stack	Time until DPM content valid - without configuration - with fieldbus configuration	 < 1s < 10s
Reset: netX chip / Firmware		
Start / Re-start	Time until DPM content is valid, if a 2nd Stage Bootloader and/or netX firmware has initialized the DPM.	min. 100ms
Reset activation	Clearing NSF_READY after HSF_RESET issued	min. 100 - 500ms
Ready after reset	Setting NSF_READY after recovering from a reset	0.5s - max. 6s
Fieldbus Communication		
General	COS command / signal handling must be recognized, therefore changes should be stable for a minimum of time and before the DPM content becomes invalid in case of a reset.	typical 20ms
Application Ready	Time until channel signals READY	typical 1000ms
BUS_ON / BUS_OFF	Time until a fieldbus protocol stack signals an activated/deactivated fieldbus communication (if available)	typical 5000ms
Channel Initialization	Re-configuration of a protocol stack	typical 10000ms
Packet Send / Receive	Sending / receiving packets via a mailbox system	typical 1000ms
Communication Channel Watchdog		
Watchdog	Watchdog trigger cycle	>= 20ms

Note: The given times are valid if the netX dual port memory is interfaced via a data / address bus. When using an SPI / USB or another serial connection, data transfer times of the physical connection have to be taken in account which may increase these times.

6.2 netX 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

The netX chip contains a ROM loader. After power-on reset, the ROM loader is started and its main task is to initialize the internal netX controller and its components (e.g. optional non-volatile boot devices such as serial Flash, parallel Flash, and DPM etc.). If a boot device is found, the ROM loader checks for an executable binary code residing in the boot media and starts it. Otherwise it depends on the hardware settings how the ROM loader proceeds.

Note: The netX chip boot process and the requirements for creating a bootable binary code is described in the netX Bootstrap Specification.

Step 2: Start of the 2nd Stage Loader

In general the first boot binary of a netX firmware is a 2nd stage loader. This loader should handle all hardware specific initializations of the target hardware, which keeps a netX protocol stack firmware independent of it. The 2nd stage loader creates the so-called *System Device* / *System Channel* in the dual-port memory area before it starts to look for non-volatile boot devices and a file system containing the resulting netX firmware.

If available, the firmware file will be started. Otherwise, the 2nd stage loader, will wait until a firmware is downloaded by the host application using the *System Mailbox.*

Step 3: Start of the netX Firmware

If the found netX firmware is started by the 2nd stage loader, the firmware takes the hardware specific information from the 2nd stage loader and creates the final layout of the DPM (System Channel / Handshake Channel / *Communication Channels).*

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.

6.3 Hardware LEDs

A standard netX based hardware offers several LEDs which can be used to identify the actual state of the firmware.

The SYS LED is always present (only one per netX device) 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.

6.3.1 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
Yellow / Green	Flashing Alternating	2nd Stage Bootloader its active
Off	N/A	netX has no Power Supply or Hardware Defect Detected

Table 98: SYS LED

6.3.2 Communication Channel LEDs

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

6.4 Reset Handling

A reset can be executed either for the whole netX chip (*Hardware Reset* executed by the netX chip) or the netX firmware (*System Reset / Boot Start*).

A *Hardware Reset* re-starts the netX chip and the complete boot procedure starts with the ROM loader execution. This includes an internal memory check and other functions to insure the integrity of the netX chip itself.

System Reset and *Boot Start* are handled by the netX firmware while the host application uses commands and flags defined in the netX DPM.

Note: During a HW reset and during the time when the 2nd stage loader starts the firmware, the content of the dual port memory can be invalid (0xFFFF / 0x0BAD) for a short period of time. If the cookie in the beginning of the DPM can be read, the boot process is finished (see section 5.2.1 / `abCookie[]`).

6.4.1 Hardware Reset

A hardware (chip) reset can only be processed via the *netX Register Block* and depends on the netX chip type.

- netX 500/100/50
  Register block is located at the end of the DPM and the accessible DPM address range must be 64 Kbyte.
- netX51/52
  Register block is either located at the beginning or the end of the DPM. Without a firmware (ROM loader only) the register block is mapped to the beginning of the DPM and the 2nd Stage Loader / netX firmware will map the block to the end of the DPM.
  The accessible DPM address range must be 64 Kbyte if mapped to the end of the DPM.

To execute a netX chip reset, a reset pattern must be written to the netX reset register.

```
Reset pattern:
0x00000000, 0x00000001,0x00000003, 0x00000007, 0x0000000F, 0x0000001F,0x0000003F,
0x0000007F, 0x000000FF
```

The reset is activated when the last value of the pattern is written.

Note: For a complete description on how to reset a specific netX chip, please consult the corresponding *netX Programming Reference Guide*.

6.4.2 System Reset

A *System Reset*, executed by a loaded netX firmware or 2nd Stage Bootloader, forces the running firmware to close all resources and to do a re-start of the firmware.

The internal handling depends on the netX target layout. If the target offers a FLASH memory and 2nd Stage Bootloader and the netX firmware is stored in the FLASH, a *System Reset* will activate the netX ROM loader which loads the 2nd Stage Loader which then loads the netX firmware.

Without a FLASH on the netX target or if both programs are not stored in FLASH (RAM based device), the firmware just jumps to the program entry point and starts over.

System Reset Procedure:

- Writing *RCX_SYS_RESET_COOKIE* (0x55AA55AA) to the `ulSystemCommandCOS` (see *System Control Block* section 5.2.4)
- Setting the *HSF_RESET* flag in `bHostFlags` (see *System Handshake Register* section 3.2)
- The netX firmware clears the *NSF_READY* flag in `bNetxFlags` (see *System Handshake Register* section 3.2), indicating that the system wide reset is in progress.
- The *NSF_READY* flag is set after a successful reset. If a startup error is detected, the *NSF_ERROR* flag is set (see `bNetxFlags`) and an error code is written to `ulSystemError` (see the *System Status Block* section 5.2.5) indicating the problem.

6.4.3 Boot Start

Boot Start is an additional option to the *System Reset*, forcing a 2nd Stage Bootloader not to start an available firmware and to stay in the bootloader (e.g. for system maintenance).

It is activated by additionally setting the *HSF_BOOTSTART* flag in the `bHostFlags` before executing the *System Reset*.

Note:	*Boot Start* only works if 2nd stage Bootloader and firmware is stored in a FLASH. On a netX target without a FLASH (RAM based), the firmware starts over without activating the 2nd Stage Loader.

6.5 Communication Channel Services

6.5.1 Channel Initialization

The *Channel Initialization* is processed by the netX firmware and forces a protocol stack, running behind a *Communication Channel,* to re-evaluate the actual fieldbus configuration and to restart network communication.

A *Channel Initiaization* does not influence other *Communication Channels* and is used if a new configuration database is downloaded to the device or if a new packet based configuration (see packet services) should be activated.

Note: If the configuration database is locked, re-initializing the communication channel is not allowed (see *RCX_COMM_COS_CONFIG_LOCKED* state page 103. and *RCX_APP_COS_LOCK_CONFIGURATION* command page 100).

Channel Initialization Procedure:

The initialization is handled in two steps:

- Set the initialization command in the *ulApplicationCOS* register by setting the *RCX_APP_COS_INIT* and *RCX_APP_COS_INITIALIZATION_ENABLE* flag at the same time (see *Common Control Block* section 5.4.2).
- Signal the new COS command by using the *HCF_HOST_COS_CMD* flag in *usHostFlags* (see *Application COS Handling* section 4.3.2

Communication Channel Flags:

- During the channel initialization the *RCX_COMM_COS_READY* and the *RCX_COMM_COS_RUN* flag, located in the *ulCommunicationCOS* register, are cleared.
- The *RCX_COMM_COS_READY* flag stays cleared for at least 20 ms before it is set again. This indicats that the initialization has been finished.
- The *RCX_COMM_COS_RUN* flag is set, if a valid configuration was found. Otherwise it stays cleared and an initialization error may be inserted into the *ulCommunicationError* indicated by the NCF_ERROR flag in *usNetXFlags*.

After the initialization process has finished, the protocol stack checks the *ulApplicationCOS* register for further setting/commands from the application and depending on the "new" configuration settings, the network communication will be restored automatically or waits until the application starts the network communication again.

In this state the aplication has to proceed with normal handshake flag operation which means checking handshake registers and setting additional application COS command like:

- BUS On (see RCX_APP_COS_BUS_ON / RCX_APP_COS_BUS_ON_ENABLE)
- Lock configuration (see *RCX_APP_COS_LOCK_CONFIG* / *RCX_APP_COS_LOCK_CONFIG_ENABLE*)
- Start DMA transfer (see *RCX_APP_COS_DMA* / *RCX_APP_COS_DMA_ENABLE*)

6.5.2 Start / Stop Communication

A communication channel (protocol stack) has the option to start network communication after power up automatically or manually by an application. This behaviour can be defined in the fieldbus configuration.

Possible Start-up Configuration Settings (e.g. SYCON.net):

Start of Bus Communication	Description
Automatically by device	The fieldbus communication is started as soon as the configuration is loaded and evaluated by the protocol stack.
Controlled by application	The protocol stack loads and evaluates the configuration bus waits until the alpplication start it.

If the protocol stack is configured in *Automatically by Device* mode it will open the network connections automatically as soon as the configuration is loaded. In the *Controlled by Application* mode, the protocol stack loads the configuration and than waits until the application sends a *Bus On* command.

Note: Which start-up mode is used is up to the application developer. In general *Controlled Start* method gives a better control over the network communication.

The actual state of the bus communication is indicated by the protocol stack in the *RCX_COMM_COS_BUS_ON* flag, located in the `ulCommounicationCOS` field.

Communication State Flag	Value	State
RCX_COMM_COS_BUS_ON	0	Bus communication is disabled
	1	Bus communication is enabled

The host application can use the *RCX_APP_COS_BUS_ON* flag in the *Application Change of State* field `ulApplicationCOS` to start and stop the bus communication. This also implies the executing of the COS handling defined for application commands (see *Application COS Handling* in section 4.3.2.)

Application State Flag	Value	State
RCX_APP_COS_BUS_ON	0	Disable bus communication
	1	Enable bus communication

In addtion to the *RCX_COMM_COS_BUS_ON* flag, which indicates the actual bus setting, the protocol stack uses the *NCF_COMMUNICATION* flag in the `usNetXFlags` register to indicate if it has an active bus communication to other devices on the network.

If the bus communication is disabled, the *NCF_COMMUNICATION flag* will be cleared an all further attempts to re-open a connection are rejected until bus communication is started again.

6.5.3 Lock / Unlock Configuration

The **Lock / Unlock Configuration** mechanism is used to prevent the configuration settings of a communication channel from being deleted or changed during run-time.

Locking and unlocking of the configuration can be achieved by an application by using the **Lock Configuration** flag RCX_APP_COS_LOCK_CONFIGURATION in ulApplicationCOS and executing the COS handling defined for application commands (see *Application COS Handling* in section 4.3.2).

Application State Flag	Value	State
RCX_APP_COS_LOCK_CONFIGURATION	0	Unlock configuration
	1	Lock configuration

The communication channel indicates the actual state by the **Configuration Locked** flag RCX_COMM_COS_CONFIG_LOCKED in ulCommunicationCOS.

Communication State Flag	Value	State
RCX_COMM_COS_CONFIG_LOCKED	0	Configuration unlocked
	1	Configuration locked

Any configuration tool shall reject those attempts when the *Configuration Locked* flag RCX_COMM_COS_CONFIG_LOCKED is set in ulCommunicationCOS.

6.5.4 Channel Watchdog

Each *Communication Channel* offers a dedicated watchdog handling, allowing the netX firmware to monitor the correct processing of the host application and vice versa.

Two fields are used for the watchdog handling:

- `ulDeviceWatchdog` (see *Common Control Block* section 5.4.2)
- `ulHostWatchdog` (see *Common Status Block* section 5.4.3)^

The handling itself simply consist of copy function which must be cyclically exceuted by the host application, where the application has to copy the content from the `ulHostWatchdog` field to the `ulDeviceWatchdog` field.

The first copy automatically activates the watchdog supervising by the firmware and from this point the application has to repeate the copy function once during the configured watchdog time. If the application does not process the copy and the watchdog time expires, a watchdog hit will be signaled to the *Communication Channel*.

Note: The actual configured watchdog time can be read from `usWatchdogTime` field located in the *Common Status Block*. If the watchdog time is set to 0, teh watchdog will not be activated.

Communication Channel watchdog behavior in case of a watchdog hit:

- Close all network connections
- Clearing the *NCF_COMMUNICATING* flag
- Set *RCX_E_WATCHDOG_TIMEOUT* error into the `ulCommunicationError` field
- Set *NCF_ERROR* flag in the `usNetXFlags` register

NOTE After a watchdog hit, the *Communication Channel* must be newly initialized (reset) before it will start again.

Watchdog States and Processing

Watchdog State	*ulHostWatchdog*	*ulDeviceWatchdog*	Description
Start-up Value	0x00000001	0x00000000	Default initialization after power-up or reset - Watchdog supervision not active
Activation / Processing	The application has to copy the content of *ulHostWatchdog* to *ulDeviceWatchdog* field. The first copy automatically starts the watchdog supervising (*ulDeviceWatchdog != 0)*		
	0x00000001	**0x00000001**	Watchdog supervision activated
Checking	Every system cycle (2ms), the firmware checks: 1. *ulDeviceWatchdog != 0* (watchdog is activated 2. *ulHostWatchdog == ulDeviceWatchdog* (watchdog is handled by the application In this case: - The watchdog timer is restarted with the configured watchdog value - The value of *ulDeviceWatchdog* is copied to *ulHostWatchdog* - *ulHostWatchdog* is incremented by 1 (0 is skipped on overflows)		
	0x00000002	0x00000001	Watchdog check was successful
Watchdog Hit	If the application does not copy *ulHostWatchdog* to *ulDeviceWatchdog* and the watchdog time expires, a watchdog hit is signaled to the communication channel.		
	ulDeviceWatchdog == ulHostWatchdog *and* *watchdog timeout has expired*		Communication Channel will be signaled
Deactivation	The watchdog supervising will be deactivated as soon as the host application writes 0 to *ulDeviceWatchdog*.		
	n	**0x00000000**	Application has deactivates the watchdog
	0x00000001	0x00000000	netX firmware will: - stop the internal watchdog timer - re-initialize *ulHostWatchdog* to 1

Because of the copy functionality and the incrementation by the netX firmware, *ulHostWatchdog* can also be used by the application to supervise the netX firmware processing.

Note: The minimum cycle time of the firmware for checking the watchdog values is 2ms and the minimal configureable watchdog time is 20ms.

6.6 Packet Services

Most of the DPM data and functions can be executed by sending so-called command packets to the netX firmware (either via the system mailbox or a channel mailbox).

These packets are defined in an own manual (see *netX DPM Packet Services*).

7 Error codes

A netX firmware contains several components which are able to signal errors. Therefore error numbers and places where the errors are signaled depend on the component.

The following table should help to identify the component signaling an error.

Component	Format / Value Range	Location	Description
all	0x0000000	*all* - RCX_S_OK - RCX_SYS_SUCCESS	No error
2nd Stage Bootloader	0x00000001 ..0x000000C	*System Status Block* Variable: *ulBootError*	See section 5.2.5
netX System (System) (rcX operating system)	0x00000001 ..0x00007FFF	*System Status Block* Variable: *ulSystemError*	See section 5.2.5
netX System (General) (General errors)	0xC0000001..0xC0000182 0xC02B0001..0xC02Byyyy	*System Status Block* Variable: *ulSystemError* or Packet error in `ulState`	See section 5.2.5 See section 4.1.1
netX System - Protocol stack error	0xC0xxyyyy xx = Protocol stack identifier yyyy = Error number	*System Status Block* Variable: *ulSystemError* or Packet error in `ulState`	See section 5.2.5 See section 4.1.1

By default, an error code of 0 = *RCX_S_OK* / *RCX_SYS_SUCCESS* indicates no failure or error. In case a system error occurs, `ulSystemError` will be set.

Additionally the *NSF_ERROR* flag in the *netX System Flags* is set (see section 3.2 and 5.2.5 for details).

7.1 Second Stage Bootloader Errors

Value	Description
0x00000000	No Error
0x00000001	There was no bootable file found inside the flash disk or parallel flash
0x00000002	No flash disk could be determined by boot loader Possible cause: Defect of the serial flash
0x00000003	Timing parameters for target could not be determined Possible causes: ▪ No SDRAM parameters found in security memory / device label or .NXF file ▪ SRAM / Flash parameters in .NXF file are missing
0x00000004	Boot header in .NXF file is corrupt.
0x00000005	Application checksum in .NXF file is invalid Possible causes: ▪ Invalid RAM Parameters ▪ Defective .NXF file ▪ Flash error
0x00000006	Error opening file on flash disk Possible cause: Defective flash volume
0x00000007	Error reading file from flash disk Possible cause: Defective flash volume
0x00000008	Command mode of boot loader was forced by user Possible causes: ▪ Rdy/Run Pins of netX are configured to Extension bus boot mode ▪ HSF_Bootstart bit was set in DPM
0x00000009	Firmware does not match device (firmware validation failed)
0x0000000A	File does not fit in SQIROM (XiP) area, while trying to update firmware
0x0000000B	Error copying firmware from to SQIROM (XiP) area
0x0000000C	Error during license check (netX 100/500 only) e. g. i2c transaction error

Table 99: 2ne Stage Bootloader Errors

7.2 netX System Errors (System)

System errors can be found in the rcX_User.h header files.

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
Other values are reserved	

Table 100: System Error Codes (System)

7.3 netX System Errors (General)

General system errors can be found in the rcX_User.h header files.

Value	Definition / Description
0x00000000	RCX_S_OK No error
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
System Middleware errors	
0xC002000C	RCX_E_TIMER_APPL_PACKET_SENT Timer Application Packet Sent Failed
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
0xC02B0009	RCX_E_UNKNOWN_PORT_INDEX Port Index Unknown
0xC02B000A	RCX_E_ROUTER_TABLE_FULL Router Table Full
0xC02B000B	RCX_E_NO_SUCH_ROUTER_IN_TABLE No Router Table Found
0xC02B000C	RCX_E_INSTANCE_NOT_NULL Instance Not Null

0xC02B000D	RCX_E_COMMAND_INVALID Invalid / Unknown Command
0xC02B000E	RCX_E_TASK_INVALID Invalid / Unknown Task
0xC02B000F	RCX_E_TASK_NOT_A_USER_TASK Not a User Task, Access Denied
0xC02B0010	TLR_E_RCX_LOG_QUE_NOT_SETTABLE Logical Queue Not Setable / Available
0xC02B0011	TLR_E_RCX_LOG_QUE_NOT_INVALID Logical Queue Invalid
0xC02B0012	TLR_E_RCX_LOG_QUE_NOT_SET Logical Queue Handle Not Set
0xC02B0013	TLR_E_RCX_LOG_QUE_ALREADY_USED Logical Queue Already In Use
0xC02B0014	TLR_E_RCX_TSK_NO_DEFAULT_QUEUE Task Has No Default Queue
0xC02B0015	TLR_E_RCX_MODULE_INVALID Firmware Module Is Invalid, CRC-32 Check Failed
0xC02B0016	TLR_E_RCX_MODULE_NOT_FOUND Firmware Module Not Found / Not Downloaded
0xC02B0017	TLR_E_RCX_MODULE_RELOC_ERROR Unable To Relocate Firmware Module / Relocation Table Invalid
0xC02B0018	TLR_E_RCX_MODULE_NO_INIT_TBL Firmware Module Init Table Missing
0xC02B0019	TLR_E_RCX_MODULE_NO_ENTRY_POINT Firmware Module Entry Point / Code missing
0xC02B001A	TLR_E_RCX_ACCESS_DENIED_IN_LOCKED_STATE Access Denied / Locked
0xC02B001B	RCX_E_INVALID_FIRMWARE_SIZE Firmware Size Invalid
0xC02B001D	RCX_E_SEC_FAILED Security EEPROM Access Failed
0xC02B001E	RCX_E_SEC_DISABLED Security EEPROM Disabled
0xC02B001F	TLR_E_RCX_INVALID_EXTENSION Invalid Extension Field
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_CHARACTER Invalid Character Found
0xC02B0024	RCX_E_PACKET_OUT_OF_SEQUNCE Packet Out Of Sequence
0xC02B0025	RCX_E_NOT_POSSIBLE_IN_CURRENT_STATE Not Allowed In Current State
0xC02B0026	RCX_E_SECURTY_EEPROM_INVALID_ZONE Security EEPROM Invalid Zone
0xC02B0028	RCX_E_SECURTY_EEPROM _NOT_AVAILABLE Security EEPROM Not Available
0xC02B0029	RCX_E_SECURTY_EEPROM_INVALID_CHECKSUM Security EEPROM Invalid Checksum
0xC02B002A	RCX_E_SECURTY_EEPROM_ZONE_NOT_WRITEABLE Security EEPROM Zone Not Writeable

0xC02B002B	RCX_E_SECURTY_EEPROM_READ_FAILED Security EEPROM Read Failed
0xC02B002C	RCX_E_SECURTY_EEPROM_WRITE_FAILED Security EEPROM Write Failed
0xC02B002D	RCX_E_SECURTY_EEPROM_ZONE_ACCESS_DENIED Security EEPROM Access Denied
0xC02B002E	RCX_E_SECURTY_EEPROM_EMULATED Security EEPROM Emulated
0xC02B002F	RCX_E_FILE_NAME_INVALID File Name Invalid
0xC02B0030	RCX_E_FILE_SEQUENCE_ERROR File / Packet Sequence Invalid
0xC02B0031	RCX_E_FILE_SEQUENCE_END_ERROR File Sequence End Error
0xC02B0032	RCX_E_FILE_SEQUENCE_BEGIN_ERROR File Sequence Begin Error
0xC02B0033	RCX_E_FILE_UNEXPECTED_BLOCK_SIZE Unexpected File Block Size
0xC02B0034	RCX_E_HIL_FILE_HEADER_CRC_ERROR Hilscher File Header CRC Error
0xC02B0035	RCX_E_HIL_FILE_HEADER_MODULE_SIZE_DIFFERS Hilscher File Header Wrong Module Size
0xC02B0036	RCX_E_HIL_FILE_HEADER_MD5_CHECKSUM_ERROR Hilscher File Header MD5 Checksum Error
0xC02B0037	RCX_E_PACKET_WOULD_BE_TO_LONG_FOR_MTU Packet Too Long For MTU
0xC02B0038	RCX_E_INVALID_BLOCK Invalid Block
0xC02B0039	RCX_E_INVALID_STRUCT_NUMBER Invalid Structure Number
0xC02B4352	TLR_E_RCX_FILE_CRC_REPEATEDLY_WRONG File Transfer was tried repeatedly with a wrong CRC
0xC02B4B54	TLR_E_RCX_CONFIGURATION_LOCKED Configuration Locked
0xC02B4D52	TLR_E_RCX_SECURITY_EEPROM_ZONE_NOT_READABLE Security EEPROM Zone Not Readable
0xC02B524C	TLR_E_RCX_FILE_TRANSFER_IN_USE File transfer in use
0xC02F0001	TLR_E_ROUTER_PACKET_TOO_BIG Packet is too big

Table 101: System Error Codes (General)

7.4 Protocol Stack Errors

These errors are protocol stack specific and defined in a file named TLR_Results.h (not part of the default DPM definition).

8 Appendix

8.1 List of figures

8.2 List of tables

8.3 Legal notes

Copyright

Important notes

Utmost care was/is given in the preparation of the documentation at hand consisting of a user's manual, operating manual and any other document type and accompanying texts. However, errors cannot be ruled out. Therefore, we cannot assume any guarantee or legal responsibility for erroneous information or liability of any kind. You are hereby made aware that descriptions found in the user's manual, the accompanying texts and the documentation neither represent a guarantee nor any indication on proper use as stipulated in the agreement or a promised attribute. It cannot be ruled out that the user's manual, the accompanying texts and the documentation do not completely match the described attributes, standards or any other data for the delivered product. A warranty or guarantee with respect to the correctness or accuracy of the information is not assumed.

We reserve the right to modify our products and the specifications for such as well as the corresponding documentation in the form of a user's manual, operating manual and/or any other document types and accompanying texts at any time and without notice without being required to notify of said modification. Changes shall be taken into account in future manuals and do not represent an obligation of any kind, in particular there shall be no right to have delivered documents revised. The manual delivered with the product shall apply.

Under no circumstances shall Hilscher Gesellschaft für Systemautomation mbH be liable for direct, indirect, ancillary or subsequent damage, or for any loss of income, which may arise after use of the information contained herein.

Liability disclaimer

The hardware and/or software was created and tested by Hilscher Gesellschaft für Systemautomation mbH with utmost care and is made available as is. No warranty can be assumed for the performance or flawlessness of the hardware and/or software under all application conditions and scenarios and the work results achieved by the user when using the hardware and/or software. Liability for any damage that may have occurred as a result of using the hardware and/or software or the corresponding documents shall be limited to an event involving willful intent or a grossly negligent violation of a fundamental contractual obligation. However, the right to assert damages due to a violation of a fundamental contractual obligation shall be limited to contract-typical foreseeable damage.

It is hereby expressly agreed upon in particular that any use or utilization of the hardware and/or software in connection with

- Flight control systems in aviation and aerospace;
- Nuclear fission processes in nuclear power plants;
- Medical devices used for life support and
- Vehicle control systems used in passenger transport

shall be excluded. Use of the hardware and/or software in any of the following areas is strictly prohibited:

- For military purposes or in weaponry;
- For designing, engineering, maintaining or operating nuclear systems;
- In flight safety systems, aviation and flight telecommunications systems;
- In life-support systems;
- In systems in which any malfunction in the hardware and/or software may result in physical injuries or fatalities.

You are hereby made aware that the hardware and/or software was not created for use in hazardous environments, which require fail-safe control mechanisms. Use of the hardware and/or software in this kind of environment shall be at your own risk; any liability for damage or loss due to impermissible use shall be excluded.

Warranty

Hilscher Gesellschaft für Systemautomation mbH hereby guarantees that the software shall run without errors in accordance with the requirements listed in the specifications and that there were no defects on the date of acceptance. The warranty period shall be 12 months commencing as of the date of acceptance or purchase (with express declaration or implied, by customer's conclusive behavior, e.g. putting into operation permanently).

The warranty obligation for equipment (hardware) we produce is 36 months, calculated as of the date of delivery ex works. The aforementioned provisions shall not apply if longer warranty periods are mandatory by law pursuant to Section 438 (1.2) BGB, Section 479 (1) BGB and Section 634a (1) BGB [Bürgerliches Gesetzbuch; German Civil Code] If, despite of all due care taken, the delivered product should have a defect, which already existed at the time of the transfer of risk, it shall be at our discretion to either repair the product or to deliver a replacement product, subject to timely notification of defect.

The warranty obligation shall not apply if the notification of defect is not asserted promptly, if the purchaser or third party has tampered with the products, if the defect is the result of natural wear, was caused by unfavorable operating conditions or is due to violations against our operating regulations or against rules of good electrical engineering practice, or if our request to return the defective object is not promptly complied with.

Costs of support, maintenance, customization and product care

Please be advised that any subsequent improvement shall only be free of charge if a defect is found. Any form of technical support, maintenance and customization is not a warranty service, but instead shall be charged extra.

Additional guarantees

Although the hardware and software was developed and tested in-depth with greatest care, Hilscher Gesellschaft für Systemautomation mbH shall not assume any guarantee for the suitability thereof for any purpose that was not confirmed in writing. No guarantee can be granted whereby the hardware and software satisfies your requirements, or the use of the hardware and/or software is uninterruptable or the hardware and/or software is fault-free.

It cannot be guaranteed that patents and/or ownership privileges have not been infringed upon or violated or that the products are free from third-party influence. No additional guarantees or promises shall be made as to whether the product is market current, free from deficiency in title, or can be integrated or is usable for specific purposes, unless such guarantees or promises are required under existing law and cannot be restricted.

Confidentiality

The customer hereby expressly acknowledges that this document contains trade secrets, information protected by copyright and other patent and ownership privileges as well as any related rights of Hilscher Gesellschaft für Systemautomation mbH. The customer agrees to treat as confidential all of the information made available to customer by Hilscher Gesellschaft für Systemautomation mbH and rights, which were disclosed by Hilscher Gesellschaft für Systemautomation mbH and that were made accessible as well as the terms and conditions of this agreement itself.

The parties hereby agree to one another that the information that each party receives from the other party respectively is and shall remain the intellectual property of said other party, unless provided for otherwise in a contractual agreement.

The customer must not allow any third party to become knowledgeable of this expertise and shall only provide knowledge thereof to authorized users as appropriate and necessary. Companies associated with the customer shall not be deemed third parties. The customer must obligate authorized users to confidentiality. The customer should only use the confidential information in connection with the performances specified in this agreement.

The customer must not use this confidential information to his own advantage or for his own purposes or rather to the advantage or for the purpose of a third party, nor must it be used for commercial purposes and this confidential information must only be used to the extent provided for in this agreement or otherwise to the extent as expressly authorized by the disclosing party in written form. The customer has the right, subject to the obligation to confidentiality, to disclose the terms and conditions of this agreement directly to his legal and financial consultants as would be required for the customer's normal business operation.

Export provisions

The delivered product (including technical data) is subject to the legal export and/or import laws as well as any associated regulations of various countries, especially such laws applicable in Germany and in the United States. The products / hardware / software must not be exported into such countries for which export is prohibited under US American export control laws and its supplementary provisions. You hereby agree to strictly follow the regulations and to yourself be responsible for observing them. You are hereby made aware that you may be required to obtain governmental approval to export, reexport or import the product.

9 Glossary

Term	Description	Section
Change of State (COS) Mechanism	Method to synchronize command / state exchange between a host application and the netX firmware	4.3
Change of State (COS) Enable Flag Mechanism	Part of the COS mechanism, used to selectively set flags without interfering other flags inside the *Change of State* register	4.3.3
Channel System Channel Communication Channel	Logical interface / path from the DPM to dedicated parts of the netX firmware System Channel => netX Firmware / System services Communication Channel=> Protocol Stack / System services	2.3 5.2 5.4
Dual-Port Memory (DPM)	Shared memory between the netX firmware and the host application, representing the physical interface to a netX based device.	2
Firmware	Binary program code executed on the netX chip ARM CPU. It contains the protocol stack and other components.	2
Handshake Handshake Flags Handshake Block	The handshake mechanism is used to synchronize data access to data located in the DPM and therefore ensuring consistent data exchange via DPM.	2.3.2 / 3.1 / 5.2.3 / 5.3 / 5.4.1
Host Host System Host Application	Program that runs on a host controller (outside the netX chip) and uses the DPM to communicate and control the netX target.	
I/O Area I/O Status	Process data image of a communication channel using handshake mechanism to synchronize access Holds the cyclic process data / state information of a network I/O Status Additional information regarding the state of input and output process data in the IO data image	2.4.9 / 4.2.1 / 5.4.8 5.4.4
Lock Configuration	Function to protect the configuration settings against changes	5.4.2
Mailbox System Mailbox Channel Mailbox	Used to exchange non-cyclic data (packets) between the host application and the netX firmware (protocol stack). Each channel (system/Communication) offers an own mailbox.	4.14.1
Packet Definition Request / Confirmation Indication / Response	A packet definition is an additional attribute of none-cyclic commands/answers defining the originator of a packet. Request / Confirmation: Request defines a command packet created by the host application and sent to the netX firmware. While a confirmation is the answer of the netX firmware. Indication / Response: Indications are command packets created by the netX firmware and sent to the host application. A response is the answer packet of the host application returned to the netX firmware.	4.1.5
Packet Packet Structure	None cyclic, packet based commands/answers, exchanged via the mailbox system	4.1.1
Process Data Image	see I/O Data Area	
Protocol Stack	A protocol stack is the functional part (state machine) of a fieldbus network application. It is an element of the netX firmware, while the firmware can also include multiple protocol stacks (via different communication channels).	

Term	Description	Section
Reset Hardware Reset System Reset Channel Initialization	A netX system offers different types of resets. Hardware Reset Reset of the netX chip and therefore for the whole system System Reset Firmware handled reset of the netX target Channel Initialization Initialization / re-initialization of a specific *Communication Channel*	6.4
Security Memory Security EEPROM	Non-volatile memory use to store hardware specific and product related information	5.2.1
Watchdog Host Watchdog Device Watchdog	Possibility to monitor the correct processing of the host application and/or netX firmware. Including the possibility to automatically shut down a fieldbus network communication if the host application stucks or crashes.	6.5.4
xC Port	Denotation of the netX chip internal communication controllers	2

Table 102: Glossary

10 Contact

Headquarters

Germany
Hilscher Gesellschaft für
Systemautomation mbH
Rheinstrasse 15
65795 Hattersheim
Phone: +49 (0) 6190 9907-0
Fax: +49 (0) 6190 9907-50
E-Mail: info@hilscher.com
Support
Phone: +49 (0) 6190 9907-99
E-Mail: de.support@hilscher.com

Subsidiaries

China
Hilscher Systemautomation (Shanghai) Co. Ltd.
200010 Shanghai
Phone: +86 (0) 21-6355-5161
E-Mail: info@hilscher.cn
Support
Phone: +86 (0) 21-6355-5161
E-Mail: cn.support@hilscher.com

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

India
Hilscher India Pvt. Ltd.
Pune, Delhi, Mumbai
Phone: +91 8888 750 777
E-Mail: info@hilscher.in

Italy
Hilscher Italia S.r.l.
20090 Vimodrone (MI)
Phone: +39 02 25007068
E-Mail: info@hilscher.it
Support
Phone: +39 02 25007068
E-Mail: it.support@hilscher.com

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

Korea
Hilscher Korea Inc.
Seongnam, Gyeonggi, 463-400
Phone: +82 (0) 31-789-3715
E-Mail: info@hilscher.kr

Switzerland
Hilscher Swiss GmbH
4500 Solothurn
Phone: +41 (0) 32 623 6633
E-Mail: info@hilscher.ch
Support
Phone: +49 (0) 6190 9907-99
E-Mail: ch.support@hilscher.com

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