**Programming reference guide**

# CIFX API

**Hilscher Gesellschaft für Systemautomation mbH**

**www.hilscher.com**

DOC121201PR09EN | Revision 9 | English | 2020-05 | Released | Public

# Table of contents



# 1 Introduction

## 1.1 About this document

This manual describes the *CIFX/COMX/netX Application Programming Interface* (*CIFX API*) and the containing functions, offered for all Hilscher standard devices based on netX controller hardware.

Aim of the API is to provide applications a target and fieldbus independent programming interface to netX based hardware running a standard Hilscher fieldbus protocol or firmware which meet the Hilscher netX dual port memory (netX DPM) definitions, described in the '*netX Dual Port Memory Interface*' manual (see reference [1]).

The API is designed to give the user easy access to all of the communication board functionalities.

In addition, Hilscher also offers a free of charge *cifX Toolkit* (C-source code based) which allows to write own drivers based on the Hilscher netX DPM definitions including the *CIFX API* functions (the toolkit is described in a separate *cifX/netX Toolkit* manual, see reference [2]*).*

## 1.2 List of revisions

| Rev | Date | Name | Chapter | Revision |
|---|---|---|---|---|
| 5 | 2018-08-28 | RMA | 2.3<br>4.7.10<br>4.8.22<br>4.8.23 | Section *Availability of API functions*: List of API functions updated.<br>Section *xSysdeviceDownload*: Table of available download modes added.<br>Section *xChannelIORead*: Note added.<br>Section *xChannelIOWrite*: Note added. |
| 6 | 2019-04-16 | ALM / LCO | 2.3<br>4.3<br>4.5.3<br>4.7.5 | Section *Availability of API functions*: xSysdeviceResetEx added.<br>Section *System Device functions*: xSysdeviceResetEx added.<br>System Control Block updated.<br>Section *xSysdeviceResetEx* added. |
| 7 | 2019-08-13 | | 4.7.5 | Section *xSysdeviceResetEx*: Description of update start expanded. |
| 8 | 2019-12-02 | ALM / LCO | 4.7.5<br>4.8.18<br>4.8.19 | Section *xSysdeviceResetEx*: Updated description of remanet data deletion<br>Section *xChannelControlBlock*: Update concening use of offset parameter for big-endian targets.<br>Section *xChannelCommonStatusBlock*: Update concening use of offset parameter for big-endian targets. |
| 9 | 2020-05-28 | LCO | 4.6, 4.7.3-4.7.6, 4.7.10, 4.8.27.4, 4.8.27.6<br>4.8.27.4 - 4.8.27.6<br>7 | Function call syntax fixed.<br>Spelling of notifications fixed.<br>Section *Error codes* updated. |

*Table 1: List of revisions*

## 1.3 Terms, abbreviations and definitions

| Term | Description |
|---|---|
| netX | Hilscher highly integrated network controller |
| rcX | Hilscher **R**eal Time **C**ommunication System for net**X** |
| cifX | **C**ommunication **I**nter**f**ace based on netX |
| comX | **Co**mmunication **M**odule based on netX |
| API | **A**pplication **P**rogramming **I**nterface |
| DPM | **D**ual-**P**ort **M**emory<br>Physical memory area, connected to a host processor.<br>Standard interface to Hilscher communication boards like CIFX/COMX or netX evaluation boards<br>(Attention: DPM may also be used as a shortcut for PROFIBUS-**DP M**aster field bus protocol). |
| SHM | **Sh**ared **M**emory<br>System memory area shared between different processes inside a software application |
| SPI | **S**erial **P**eripheral **I**nterface |
| RPC | **R**emote **P**rocedure **C**alls |
| DPM Manual | Description of the standard Hilscher DPM layout and functionality |
| netX Transport | Diagnostics and remote access functions to netX based remote devices via serial interfaces |
| CIFX Toolkit | C source-code based implementation of the standard Hilscher DPM access functions |
| SDO | **S**ervice **D**ata **O**bject |
| PDO | **P**rocess **D**ata **O**bject |

*Table 2: Terms, abbreviations and definitions*

## 1.4 References to documents

This document refers to the following documents:

[1] Hilscher Gesellschaft für Systemautomation mbH: DPM Manual,
netX Dual-Port Memory Interface DPM Manual, Revision 17, English, 2020.
Description of the standard Hilscher DPM layout and functionality

[2] Hilscher Gesellschaft für Systemautomation mbH: Toolkit Manual,
cifX/netX Toolkit - DPM, Revision 11, English, 2019.
C source-code based implementation of the standard Hilscher DPM access functions

[3] Hilscher Gesellschaft für Systemautomation mbH: Program Reference Guide,
netX Diagnostic and Remote Access, Fundamentals Revision 3, English, 2019.

Hilscher Gesellschaft für Systemautomation mbH: Program Reference Guide,
netX Diagnostic and Remote Access, Host Device Revision 3, English, 2013.

Hilscher Gesellschaft für Systemautomation mbH: Program Reference Guide,
netX Diagnostic and Remote Access, Target Device Revision 3, English, 2010.

*Table 3: References to documents*

# 2 Application note

## 2.1 Component overview for host applications

Hilscher offers the *CIFX API* on different platforms and as different applications (DLL / library or C source code). Usually the API comes with an operating system driver or with the CIFX Toolkit.

The use of the API also implies the physical hardware connection to the netX hardware. While a device driver uses memory functions to access the DPM, the toolkit also allows the implementation of alternative hardware access functions like *DPM via SPI* or custom access functions like *DPM via a custom USB* protocol.

The *netX Transport DLL* is a special implementation of the *CIFX API*, including hardware access via serial interfaces (e. g. USB, serial or Ethernet connections). This component is able to convert *CIFX API* function calls either into appropriate packet based commands (rcX packets, described in reference [1]) or into a dedicated binary format which enables the execution of the API functions on a remote system (similar to RPC - remote procedure calls).

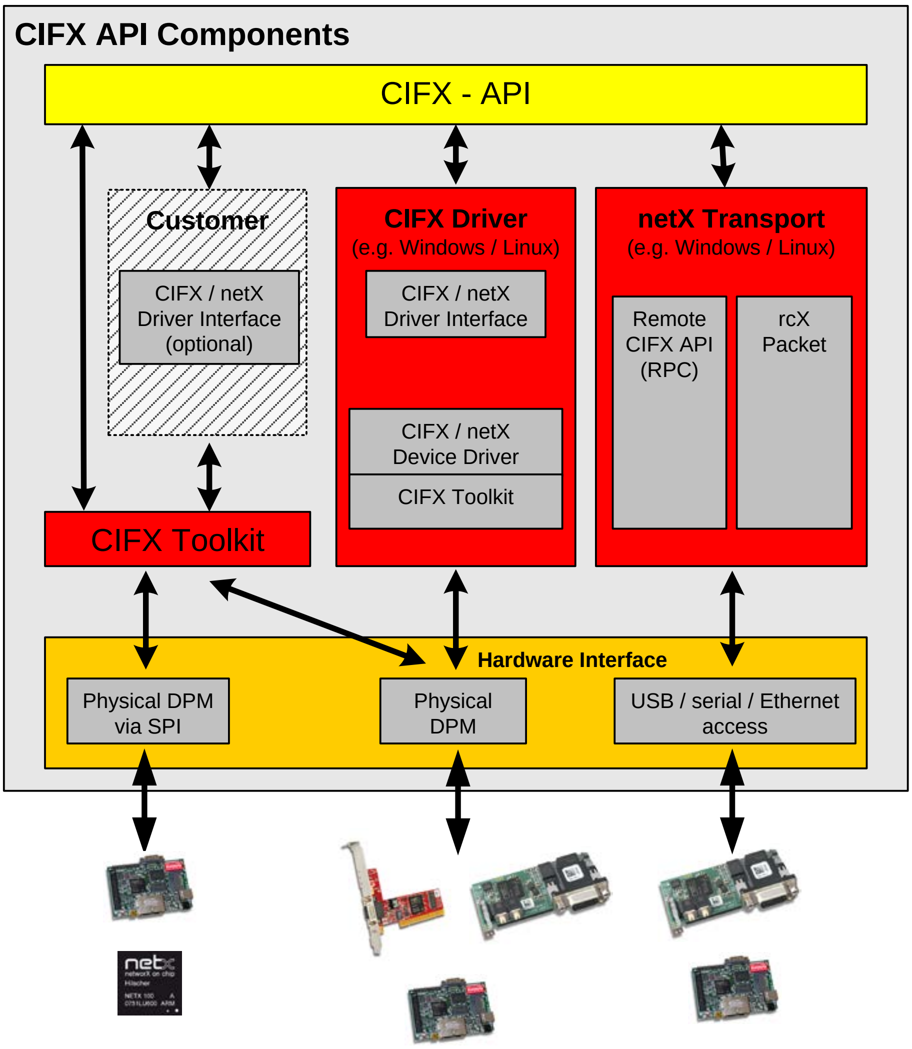



*Figure 1: CIFX API components*

**Note:** Not all of the physical connections can be made available on every host system (e.g. SPI on Windows).

- Physical DPM interface
  The DPM hardware interface, with direct access to the memory, offers the entire defined API functions.
- Physical DPM via SPI
  The netX hardware offers a SPI interface accepting encoded memory access functions (memory read / write functions) to the DPM. These SPI commands are decoded by the hardware / firmware and executed on the internal DPM of the netX device.
- USB / serial / Ethernet access
  Another possibility to access a netX device is the use of serial interfaces (USB/serial/Ethernet). In this case, *CIFX API* function calls are converted (described in the *netX Diagnostic and Remote Access* manual, see reference [3]), transferred and processed by the remote netX device.
  This type of communication does not offer all of the *CIFX API* functions. Especially functions related to mapped memory areas like *xChannelPLC...()* functions.

## 2.2 Component overview for netX applications

Hilscher also provides the application development directly on the netX communication controller. The *CIFX API* is also available in this environment.

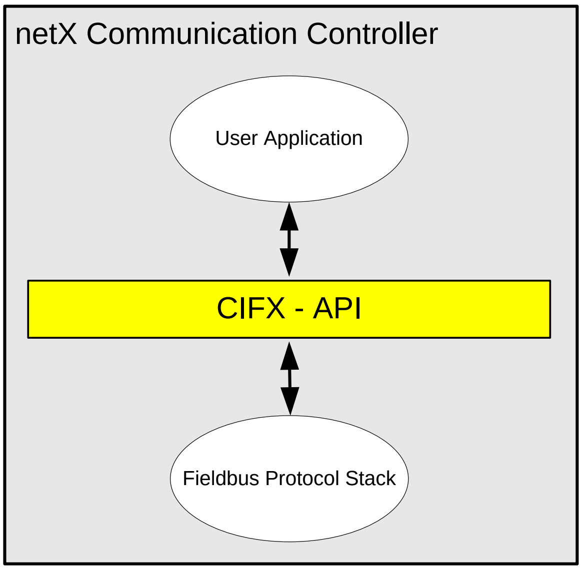



*Figure 2: Component Overview for netX applications*

## 2.3 Availability of API functions

Following table lists the available API functions regarding to the used environment and the physical hardware connection.

### Driver functions

| Function | Brief description | DPM | DPM via SPI | USB / serial / Ethernet | netX |
|---|---|---|---|---|---|
| | | | | RPC \| Packet | |
| xDriverOpen | Opens the driver | X | X | X \| X | X |
| xDriverClose | Closes the driver | X | X | X \| X | X |
| xDriverGetInformation | Retrieves driver information | X | X | X \| X | X |
| xDriverGetErrorDescription | Retrieves an error code description | X | X | X \| X | X |
| xDriverEnumBoards | Enumerates available boards/devices | X | X | X \| X | X |
| xDriverEnumChannels | Enumerates available channels on a specific board | X | X | X \| X | X |
| xDriverRestartDevice | Restarts a device | X | X | O \| O | O |
| xDriverMemoryPointer | Gets/Releases a pointer to the dual port memory. **Only to be used for debugging purpose.** | X | O | O \| O | X |

*Table 4: List of API functions – Driver functions*

### System Device functions

| Function | Brief description | DPM | DPM via SPI | USB / serial / Ethernet | netX |
|---|---|---|---|---|---|
| | | | | RPC \| Packet | |
| xSysdeviceOpen | Opens a connection a system device | X | X | X \| X | X |
| xSysdeviceClose | Closes a connection to a system device | X | X | X \| X | X |
| xSysdeviceInfo | Gets System device information | X | X | X \| X | X |
| xSysdeviceReset | Performs a device reset | X | X | X \| X | X |
| xSysdeviceResetEx | Performs a device reset with parameters | X | X | X \| X | X |
| xSysdeviceBootstart | Performs a device boot start | X | X | O \| O | O |
| xSysdeviceGetMBXState | Retrieves the system mailbox state | X | X | X \| X | X |
| xSysdeviceGetPacket | Reads a pending packet | X | X | X \| X | X |
| xSysdevicePutPacket | Sends a packet | X | X | X \| X | X |
| xSysdeviceDownload | Downloads a file/configuration/firmware | X | X | X \| X | X |
| xSysdeviceFindFirstFile | Finds the first file in the given directory | X | X | X \| X | X |
| xSysdeviceFindNextFile | Finds the next file entry in the given directory | X | X | X \| X | X |
| xSysdeviceUpload | Uploads a file/configuration/firmware | X | X | X \| X | X |
| xSysdeviceExtendedMemory | Gets a pointer to the extended memory area | X (1) | O | O \| O | O |

*Table 5: List of API functions – System Device functions*

### Communication Channel functions

| Function | Brief description | DPM | DPM via SPI | USB / serial / Ethernet<br>RPC \| Packet | netX |
|---|---|---|---|---|---|
| xChannelOpen | Opens a communication channel | X | X | X \| X | X |
| xChannelClose | Closes a communication channel | X | X | X \| X | X |
| **Asynchronous services (Packets)** | | | | | X |
| xChannelGetMBXState | Retrieves the channels mailbox state | X | X | X \| X | X |
| xChannelGetPacket | Reads packet from the channel mailbox | X | X | X \| X | X |
| xChannelPutPacket | Sends a packet to the channel mailbox | X | X | X \| X | X |
| xChannelGetSendPacket | Reads back the packet sent | X | X | X \| O | X |
| **Device Administrational/Informational functions** | | | | | X |
| xChannelDownload | Downloads a file/configuration to the channel | X | X | X \| X | X |
| xChannelReset | Resets the channel | X | X | X \| X | X (3) |
| xChannelInfo | Retrieves channel specific information | X | X | X \| O | X |
| xChannelWatchdog | Activates/Deactivates/Trigger Watchdog | X | X | X \| O | X |
| xChannelHostState | Sets the Application state flag<br>(signal application is running or not) | X | X | X \| O | X |
| xChannelBusState | Sets the bus state flag<br>(start or stop fieldbus communication) | X | X | X \| X | X |
| xChannelControlBlock | Access the Channels control block | X | X | X \| X | X |
| xChannelCommonStatusBlock | Access to the common status block | X | X | X \| X | X |
| xChannelExtendedStatusBlock | Access to the extended status block | X | X | X \| X | X |
| xChannelUserBlock | Access user block (not implemented yet!) | X | X | X \| X | X |
| **Cyclic Data services (I/O's)** | | | | | |
| xChannelIORead | Instructs the device to place the latest data into the DPM and passes them to the user | X | X | X \| O | X |
| xChannelIOWrite | Copies the data to the DPM and waits for the firmware to retrieve them | X | X | X \| O | X |
| xChannelIOReadSendData | Reads back the last send data | X | X | X \| O | X |
| **Cyclic Data services (I/O's, PLC optimized)** | | | | | |
| xChannelPLCMemoryPtr | Gets a pointer to the IO Block | X | O (2) | O \| O | X |
| xChannelPLCActivateRead | Instruct the firmware to place the latest input data into the dual port (no wait for completion) | X | O (2) | O \| O | X |
| xChannelPLCActivateWrite | Instruct the firmware to retrieve the latest output data from the dual port (no wait for completion) | X | O (2) | O \| O | X |
| xChannelPLCIsReadReady | Checks if the last Read Activation has finished | X | O (2) | O \| O | X |
| xChannelPLCIsWriteReady | Checks if the last Write Activation has finished | X | O (2) | O \| O | X |

| Function | Brief description | DPM | DPM via SPI | USB / serial / Ethernet<br>RPC \| Packet | netX |
|---|---|---|---|---|---|
| **DMA services** | | | | | |
| xChannelDMAState | Activates/Deactivates DMA mode | X (1) | O | O \| O | O |
| **Bus synchronous operation** | | | | | |
| xChannelSyncState | Waits for a synchronization event or trigger/acknowledge a sync event | X | X (4) | O \| O | O |
| **Notification services (only available in Interrupt mode)** | | | | | |
| xChannelRegisterNotification | Registers a notification callback | X | X (4) | O \| O | O |
| xChannelUnregisterNotification | Un-register a notification callback | X | X (4) | O \| O | O |

*Table 6: List of API functions – Communication Channel functions*

**Notes:**

(1) PCI / PCIe hardware only.

(2) Special implementation for SPI necessary.

(3) A system reset will reset the whole device.

(4) Notifications are only available in interrupt mode.

# 3 API basics

As described before, the *CIFX API* is the common, fieldbus protocol independent, function interface to Hilscher CIFX/COMX and netX based devices.

It is based on the Hilscher netX DPM (dual-port memory) definition and abstracts the access to the netX based hardware and the Hilscher netX protocol firmware running on the netX.

The API offers a set of functions grouped into *'Driver'* related, '*System Device'* related and *'Communication Channel'* related functions.

Each of the group covers device specific functions by providing a set of API functions necessary for the specific handling.

**Functional Groups in the CIFX-API**

- Driver-related functions
  Administration of multiple devices in a standardized way
- *System Device*-related functions
  General device functions (e.g. system reset, download, device information)
- *Communication Channel*-related functions
  Fieldbus protocol stack handling

## 3.1 DPM layout, devices and channels

The DPM layout divides the interface to a netX device into several areas (channels) where each channel has its own structure, predefined information and functionality and can be handled independently from other channels.

A standard netX firmware offers up to 8 channels with three different channel definitions.

**General dual-port memory layout**

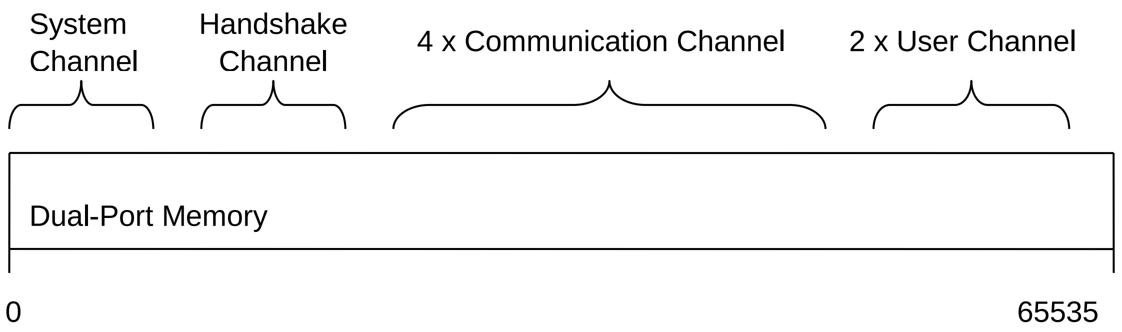



*Figure 3: General dual-port memory layout*

**Channel definition**

- *System Channel* (also named *System Device)*
  The main channel is the '*System Channel*' also named '*System Device*'. It is always available and used for administration functions belonging to the whole device, like hardware reset, firmware download etc.
- *Communication Channels*
  Communication channels representing a fieldbus connection (a fieldbus protocol stack) with its information and functionalities.
  Up to four communication channels are possible.
- *User Channels*
  User channels are the third type of channels and designed for user applications, running on the netX chip in parallel to fieldbus protocol stack (two user channels are possible).

The *Handshake Channel* is necessary for special device functionalities like interrupt handling and DPM access synchronization and therefore it has no user API functions offering access to this channel.

*CIFX API* function names correspond to the function groups and channel definitions.

- **xSysdevice..........()** Functions and functionalities corresponding to the system channel functionalities
- **xChannel.............()** Functions and functionalities corresponding communication channel and its needs
- **xDriver…..............()** Functions to handle multiple devices in a common way

## 3.2 Basic fieldbus data handling

NetX devices are providing two basic mechanisms to transfer user data between a fieldbus protocol stack and user applications.

First one is the cyclic process data transfer mechanism (*Transfer of the Process Data Image*) and the second one is the asynchronous data transfer mechanism (*Packet Oriented Data Transfer*). Other information like configuration, diagnostic and device specific administration functions are also based on the asynchronous data transfer mechanism.

- **Asynchronous Data Transfer (Packet Oriented Data Transfer)**
  Data are transferred by using a data structure named rcX packet. Packet transfer between a host system and the cifX hardware takes place via a, so called, mailbox system. This method is used to transfer of SDO, administration, configuration and diagnostic data.
- **Cyclic Process Data Transfer (Process Data Image Transfer)**
  Data are located in a process data image. This method is used for I/O based protocols (PDO transfer).
  Input and output data are located in separate memory areas which can be handled independently.

**Note:** A complete description of the fieldbus data handling can be found in the *netX Dual Port Memory Interface* manual

## 3.3 Fieldbus-specific information and functions

Beside the general device data in the DPM each fieldbus protocol stack comes with its own specific data and functions. This specific information can be found in the corresponding protocol API manuals.

**Note:** Fieldbus specific information and functionalities can be found in the corresponding protocol API manuals (e. g. *PROFIBUS DP Master Protocol API 15 EN.pdf*).

# 4 CIFX API (Application Programming Interface)

The API offers functions grouped into *'Driver'*, '*System Channel'* and *'Communication Channel'* related functions.

## 4.1 API header files

The API comes with a single C header file.

API definition file: *cifXUser.h*
Error definitions: *cifXError.h*

## 4.2 Driver functions

The driver related functions are used to handle a driver and offers functions to identify the connected hardware.

| Function | Description |
|---|---|
| xDriverOpen | Opens the driver, allowing access to every driver function |
| xDriverClose | Closes an open connection to the driver |
| xDriverGetInformation | Retrieves driver information (e.g. Version) |
| xDriverGetErrorDescription | Retrieves an English description of a cifX driver error code |
| xDriverEnumBoards | Enumerate through all boards/devices the driver is managing |
| xDriverEnumChannels | Enumerate through all channels located on a specific board |
| xDriverRestartDevice | Restart a device |
| xDriverMemoryPointer | Get/Release a pointer to the dual port memory.<br>**This function must only be used for debugging purpose!** |

*Table 7: Driver functions*

## 4.3 System Device functions

Each communication board owns a *System Device* allowing generic access to the device. This '*System Device*' only offers a small mailbox and system global status information and should not be used to communicate with a protocol stack directly.

| Function | Description |
|---|---|
| xSysdeviceOpen | Opens a connection to a boards system device |
| xSysdeviceClose | Closes a connection to a system device |
| xSysdeviceInfo | Gets System device specific information (e.g. mailbox size) |
| xSysdeviceReset | Performs a device reset |
| xSysdeviceResetEx | Performs a device reset with parameters |
| xSysdeviceBootstart | Performs a device boot start.<br>netX 10/50/51/52/100/500: This will activate the Second Stage Boot Loader. The Second Stage Boot Loader does not start the firmware.<br>netX 90/4000/4100: This will activate the Maintenance Firmware.<br>**Note:** Only possible on Flash-based devices. |
| xSysdeviceGetMBXState | Retrieves the system mailbox state |
| xSysdeviceGetPacket | Retrieves a pending packet from the system mailbox |
| xSysdevicePutPacket | Sends a packet to the system mailbox |
| xSysdeviceDownload | Downloads a file/configuration/firmware to the device |
| xSysdeviceFindFirstFile | Finds the first file entry in the given directory |
| xSysdeviceFindNextFile | Finds the next file entry in the given directory |
| xSysdeviceUpload | Uploads a file/configuration/firmware from the device |
| xSysdeviceExtendedMemory | Gets a pointer to an available extended memory area |

*Table 8: System Device functions*

## 4.4 Communication Channel functions

Each protocol stack is represented as a *Communication Channel*.

Communication channels owning a set of functions, allowing every possible interaction with the protocol stack.

The *CIFX API* functions are protocol stack independent and used for all available Hilscher netX based protocol stacks. Only the data content is protocol specific and must be interpreted by the user application.

### Communication Channel functions

| Function | Description |
|---|---|
| xChannelOpen | Opens a connection to a communication channel |
| xChannelClose | Closes a connection |
| **Asynchronous services (Packets)** | |
| xChannelGetMBXState | Retrieve the channels mailbox state |
| xChannelGetPacket | Retrieve a pending packet from the channel mailbox |
| xChannelPutPacket | Send a packet to the channel mailbox |
| xChannelGetSendPacket | Read back the last sent packet |
| **Device Administrational/Informational functions** | |
| xChannelDownload | Download a file/configuration to the channel |
| xChannelReset | Reset the channel |
| xChannelInfo | Retrieve channel specific information |
| xChannelWatchdog | Activate/Deactivate/Trigger the channel Watchdog |
| xChannelHostState | Set the application state flag in the application COS flags, to signal the hardware if an application is running or not |
| xChannelBusState | Set the bus state flag in the application COS state flags, to start or stop fieldbus communication. |
| xChannelControlBlock | Access the channel control block |
| xChannelCommonStatusBlock | Access to the common status block |
| xChannelExtendedStatusBlock | Access to the extended status block |
| xChannelUserBlock | Access user block (not implemented yet!) |
| **Cyclic Data services (I/O's)** | |
| xChannelIORead | Instructs the device to place the latest data into the DPM and passes them to the user |
| xChannelIOWrite | Copies the data to the DPM and waits for the firmware to retrieve them |
| xChannelIOReadSendData | Reads back the last send data |
| **Cyclic Data services (I/O's, PLC optimized)** | |
| xChannelPLCMemoryPtr | Get a pointer to the I/O memory block |
| xChannelPLCActivateRead | Instruct the firmware to place the latest input data into the I/O memory block (no wait for completion) |
| xChannelPLCActivateWrite | Instruct the firmware to retrieve the latest output data from the I/O memory block (no wait for completion) |
| xChannelPLCIsReadReady | Checks if the last read activation has finished |
| xChannelPLCIsWriteReady | Checks if the last write activation has finished |

| Function | Description |
|---|---|
| **DMA services** | |
| xChannelDMAState | Activate/Deactivate DMA mode |
| **Bus synchronous operation** | |
| xChannelSyncState | Wait for synchronization events or trigger / acknowledge a sync event |
| **Notification services (only available in Interrupt mode)** | |
| xChannelRegisterNotification | Register a notification callback |
| xChannelUnregisterNotification | Un-register a notification callback |

*Table 9: Communication Channel functions*

# 4.5 Structure definitions

**Note:** All structures are byte packed, for easy portability and data exchange via the DPM.

## 4.5.1 Driver information

When querying the driver information the following structure is expected in the function call.

| Element | Type | Description |
|---|---|---|
| abDriverVersion | uint8_t[32] | Human readable driver name and version |
| ulBoardCnt | uint32_t | Number of handled boards |

*Table 10: Driver information structure (DRIVER_INFORMATION)*

## 4.5.2 Board information

The board information structure is used, when enumerating boards.

| Element | Type | Description |
|---|---|---|
| lBoardErrror | uint32_t | Global board error (currently not used always 0) |
| abBoardName | uint8_t[16] | This is the name of the board which can be used for opening a channel or the system device on it. |
| abBoardAlias | uint8_t[16] | This is an alternate, user-definable name for the device |
| ulBoardID | uint32_t | Unique driver created board identifier |
| ulSystemError | uint32_t | Boot-up/System error, when trying to handle device |
| ulPhysicalAddress | uint32_t | Physical address of the device's DPM |
| ulIrqNumber | uint32_t | Interrupt number assigned to the device |
| bIrqEnabled | uint32_t | Defines if the interrupt is used by the driver, or if the driver works in polling mode for this device |
| ulChannelCnt | uint32_t | Number of available channels |
| ulDpmTotalSize | uint32_t | Total size of the dual port in bytes |
| tSystemInfo | SYSTEM_CHANNEL_SYSTEM_INFO_BLOCK (see below) | |

*Table 11: Board information structure (BOARD_INFORMATION)*

### 4.5.3 System channel information

The following structures are returned on calls to *xSysdeviceInfo()* depending on the passed command parameter:

**Command: CIFX_INFO_CMD_SYSTEM_INFORMATION**

| Element | Type | Description |
|---|---|---|
| ulSystemError | uint32_t | Boot-up/System error, when trying to handle device |
| ulDpmTotalSize | uint32_t | Total size of the dual port in bytes |
| ulMBXSize | uint32_t | Size of the system mailbox in bytes |
| ulDeviceNumber | uint32_t | Device number (as found on the matrix label) |
| ulSerialNumber | uint32_t | Serial number (as found on the matrix label) |
| ulOpenCnt | uint32_t | Number of times this device is open |

*Table 12: System channel information (SYSTEM_CHANNEL_SYSTEM_INFORMATION)*

**Command: CIFX_INFO_CMD_SYSTEM_INFO_BLOCK**

| Element | Type | Description |
|---|---|---|
| abCookie | uint8_t[4] | System channel identifier MUST be "netX" |
| ulDpmTotalSize | uint32_t | Total size of the dual port in bytes |
| ulDeviceNumber | uint32_t | Device number (as found on the matrix label) |
| ulSerialNumber | uint32_t | Serial number (as found on the matrix label) |
| ausHwOptions | uint16_t[4] | Array of hardware options for all four possible ports of the netX |
| usManufacturer | uint16_t | Manufacturer ID |
| usProductionDate | uint16_t | Production date code |
| ulLicenseFlags1 | uint32_t | Hilscher dedicated license flags (e.g. fieldbus license) |
| ulLicenseFlags2 | uint32_t | Hilscher dedicated license flags (e.g. additional information) |
| usNetxLicenseID | uint16_t | Special netX user license information |
| usNetxLicenseFlags | uint16_t | Dedicated netX user license information |
| usDeviceClass | uint16_t | Hardware device class (e.g. CIFX / COMX etz.) |
| bHwRevision | uint8_t | Hardware revision |
| bHwCompatibility | uint8_t | Hardware compatibility list |
| bDevIdNumber | uint8_t | Device identification number (rotary switch) |
| bReserved | uint8_t | unused/reserved |
| usReserved | uint16_t | unused/reserved |

*Table 13: System channel info block (SYSTEM_CHANNEL_SYSTEM_INFO_BLOCK)*

### Command: CIFX_INFO_CMD_SYSTEM_CHANNEL_BLOCK

| Element | Type | Description |
|---|---|---|
| abInfoBlock | uint8_t[8][16] | Channel information in the system channel |

*Table 14: System Channel - Channel Info Block (SYSTEM_CHANNEL_CHANNEL_INFO_BLOCK)*

### Area definitions in cifXUser.h

CIFX_MAX_NUMBER_OF_CHANNEL_DEFINITION = 8

CIFX_SYSTEM_CHANNEL_DEFAULT_INFO_BLOCK_SIZE = 16

**Note:** To evaluate the content of the *abInfoBlock* array, refer to the netX DPM Interface Manual and the rcX_User.h, structure NETX_CHANNEL_INFO_BLOCK.

### Command: CIFX_INFO_CMD_SYSTEM_CONTROL_BLOCK

| Element | Type | Description |
|---|---|---|
| ulSystemCommandCOS | uint32_t | System channel host COS flags |
| ulSystemControl | uint32_t | netX 10/50/51/52/100/500: Reserved, not used. |
| | | netX 90/4000/4100: System channel control. |

*Table 15: System channel control block (SYSTEM_CHANNEL_SYSTEM_CONTROL_BLOCK)*

### Command: CIFX_INFO_CMD_SYSTEM_STATUS_BLOCK

| Element | Type | Description |
|---|---|---|
| ulSystemCOS | uint32_t | System channel device COS flags |
| ulSystemStatus | uint32_t | Actual system state |
| ulSystemError | uint32_t | Actual system error |
| ulBootError | uint32_t | Error code from the second stage boot loader (only valid if Cookie="BOOT") |
| ulTimeSinceStart | uint32_t | Time since system start in seconds |
| usCpuLoad | uint16_t | CPU load in 0,01% units (10000 => 100%) |
| usReserved | uint16_t | Reserved for later use |
| ulHWFeatures | Uint32_t | Information about hardware features (e.g. MRAM / RTC) |
| abReserved | uint8_t[36] | unused/reserved |

*Table 16: System channel status block (SYSTEM_CHANNEL_SYSTEM_STATUS_BLOCK)*

### 4.5.4 Communication channel information

The following structure is returned on calls to *xChannelInfo()* or when enumerating channels on a Board using *xDriverEnumChannels()*:

| Element | Type | Description |
|---|---|---|
| abBoardName | uint8_t[16] | This is the name of the board which can be used for opening a channel or the system device on it. |
| abBoardAlias | uint8_t[16] | This is an alternate, user-definable name for the device |
| ulDeviceNumber | uint32_t | Device number (as found on the matrix label) |
| ulSerialNumber | uint32_t | Serial number (as found on the matrix label) |
| usFWMajor | uint16_t | Major version number of firmware |
| usFWMinor | uint16_t | Minor version number of firmware |
| usFWBuild | uint16_t | Build number of firmware |
| usFWRevision | uint16_t | Revision version number of firmware |
| bFWNameLength | uint8_t | Length of firmware name |
| abFWName | uint8_t[63] | Firmware name |
| usFWYear | uint16_t | Build year of firmware |
| bFWMonth | uint8_t | Build month of firmware (1..12) |
| bFWDay | uint8_t | Build day of firmware (1..31) |
| ulChannelError | uint32_t | Communication channel error from the "Common Status Block" |
| ulOpenCnt | uint32_t | Number of calls to xChannelOpen for this channel |
| ulPutPacketCnt | uint32_t | Number of successful transmitted packets |
| ulGetPacketCnt | uint32_t | Number of successfully received packets |
| ulMailboxSize | uint32_t | Mailbox size in Bytes |
| ulIOInAreaCnt | uint32_t | Number of I/O Input areas |
| ulIOOutAreaCnt | uint32_t | Number of I/O output areas |
| ulHskSize | uint32_t | RCX_HANDSHAKE_SIZE_8BIT (0x01) or<br>RCX_HANDSHAKE_SIZE_16BIT (0x02) |
| ulNetxFlags | uint32_t | Actual netX communication flags (usNetxCommFlag) |
| ulHostFlags | uint32_t | Actual host communication flags (usHostCommFlags) |
| ulHostCOSFlags | uint32_t | Actual applicaton COS flags (ulApplicationCOS of Control Block) |
| ulDeviceCOSFlags | uint32_t | Actual communication COS flags (ulCommunicationCOS of Common Status Block) |

*Table 17: Channel information structure*

# 4.6 Driver-related functions

## 4.6.1 xDriverOpen

This function opens a connection / handle to the cifX driver.

### Function call

```
int32_t xDriverOpen( CIFXHANDLE* phDriver)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| phDriver | CIFXHANDLE* | returned handle to the driver |

*Table 18: xDriverOpen arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### 4.6.2 xDriverClose

This function closes a connection / handle to the cifX driver.

#### Function call

```
int32_t xDriverClose(     CIFXHANDLE hDriver)
```

#### Arguments

| Argument | Data type | Description |
|---|---|---|
| hDriver | CIFXHANDLE | Handle returned by xDriverOpen |

*Table 19: xDriverClose arguments*

#### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### 4.6.3 xDriverGetInformation

This function retrieves all driver specific information, like version number, build date, etc.

#### Function call

```
int32_t xDriverGetInformation(  CIFXHANDLE      hDriver,
                                uint32_t        ulSize,
                                void*           pvDriverInfo)
```

#### Arguments

| Argument | Data type | Description |
|---|---|---|
| hDriver | CIFXHANDLE | Handle returned by xDriverOpen |
| ulSize | uint32_t | Size of the passed structure |
| pvDriverInfo | void* | Pointer to a DRIVER_INFORMATION structure, to place returned values in. |

*Table 20: xDriverGetInformation arguments*

#### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

#### Example

```
DRIVER_INFORMATION tDriverInfo = {0};
int32_t lRet = xDriverGetInformation(NULL, sizeof(tDriverInfo), &tDriverInfo);
if( lRet == CIFX_NO_ERROR)
{
}
```

## 4.6.4 xDriverGetErrorDescription

Look up function for driver errors. The function returns a human-readable error description (English only).

### Function call

```
int32_t xDriverGetErrorDescription(  int32_t    lError,
                                     char*      szBuffer,
                                     uint32_t   ulBufferLen)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| lError | int32_t | Error value returned by any driver function |
| szBuffer | String | Pointer to a ASCII string buffer, to place returned text in |
| ulBufferLen | uint32_t | length of the string buffer for returned data |

*Table 21: xDriverGetErrorDescription arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

### Example

```
// Read driver error description
char szError[1024] ={0};
xDriverGetErrorDescription( lError,  szError, sizeof(szError));
```

## 4.6.5 xDriverEnumBoards

Enumerate all currently handled boards/cards of the driver.

### Function call

```
int32_t xDriverEnumBoards( CIFXHANDLE      hDriver,
                           uint32_t        ulBoard,
                           uint32_t        ulSize,
                           void*           pvBoardInfo)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hDriver | CIFXHANDLE | Handle to the driver (returned by xDriverOpen) |
| ulBoard | uint32_t | Board number to return info for.<br>This must be incremented from zero until an error is returned to query all boards |
| ulSize | uint32_t | length of the Structure passed in pvBoardInfo |
| pvBoardInfo | void* | Pointer to returned BOARD_INFORMATION structure |

*Table 22: xDriverEnumBoards arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### Example

```
int32_t lBoardRet;
do {
  BOARD_INFORMATION tBoardInfo = {0};
  lBoardRet = xDriverEnumBoards(NULL, ulBoardIdx++, sizeof(tBoardInfo), &tBoardInfo);

  if(lBoardRet == CIFX_NO_ERROR)
  {
  }
} while(lBoardRet == CIFX_NO_ERROR);
```

## 4.6.6 xDriverEnumChannels

Enumerate all available channels on a board/card.

### Function call

```
int32_t xDriverEnumChannels(    CIFXHANDLE      hDriver,
                                uint32_t        ulBoard,
                                uint32_t        ulChannel,
                                uint32_t        ulSize,
                                void*           pvChannelInfo)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hDriver | CIFXHANDLE | Handle to the driver (returned by xDriverOpen) |
| ulBoard | uint32_t | Board number to return info for (constant during channel enumeration). |
| ulChannel | uint32_t | Channel number to enumerate.<br>This must be incremented from zero until an error is returned to query all channels |
| ulSize | uint32_t | length of the Structure passed in pvBoardInfo |
| pvChannelInfo | void* | Pointer to returned CHANNEL_INFORMATION structure |

*Table 23: xDriverEnumChannels arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### Example

```
int32_t lChannelRet;
do {
  CHANNEL_INFORMATION tChannelInfo = {0};
  lChannelRet = xDriverEnumChannels(NULL, ulBoardIdx, ulChannelIdx++,
                                   sizeof(tChannelInfo), &tChannelInfo);
  if(lChannelRet == CIFX_NO_ERROR)
  {

  }
} while(lChannelRet == CIFX_NO_ERROR);
```

## 4.6.7 xDriverRestartDevice

The function can be used to restart a netX board. The driver processes the same functions like on a power on reset (reset the hardware and download the boot loader, firmware and configuration files etc.).

A restart is necessary on PCI based netX boards if a running firmware should be updated or changed. Because on such boards the firmware is not stored in a Flash file system and updating the firmware while it is running in RAM is not possible.
On Windows based systems a restart can also be performed using the *Windows Device Manager* to deactivate/activate the board.

**Note:** A restart is only performed if no application has an open handle to the board or one of its communication channels.

### Function call

```
int32_t APIENTRY xDriverRestartDevice(    CIFXHANDLE hDriver,
                                          char*      szBoardName,
                                          void*      pvData);
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hDriver | CIFXHANDLE | Handle to the driver (returned by *xDriverOpen*) |
| szBoardName | String | Identifier for the Board.<br>(e.g. "cifX<Board Number>") |
| pvData | void* | For further extensions can be NULL |

*Table 24: xDriverRestartDevice arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.6.8 xDriverMemoryPointer

Return a pointer to the dual port memory of a board/channel. This function should only be used for debugging purposes, because the function only maps the card memory into the processes memory area.

### Function call

```
int32_t xDriverMemoryPointer (  CIFXHANDLE      hDriver,
                                uint32_t        ulBoard,
                                uint32_t        ulCmd,
                                void*           pvMemoryInfo)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hDriver | CIFXHANDLE | Handle to the driver (returned by xDriverOpen) |
| ulBoard | uint32_t | Board number to return pointer for. |
| ulCmd | uint32_t | Maps the dual port memory for direct access from an application<br>**1 = CIFX_MEM_PTR_OPEN**<br>Map a user specific memory area<br>2 = CIFX_MEM_PTR_USR -> not supported<br>Release the dual port pointer (same memory structure MUST be passed)<br>**3 = CIFX_MEM_PTR_CLOSE** |
| pvMemoryInfo | void* | Pointer to returned MEMORY_INFORMATION structure<br>**Note:** The Parameter ulChannel must be inserted! |

*Table 25: xDriverMemoryPointer arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### Description of the MEMORY_INFORMATION structure

| Value | Data type | Description |
|---|---|---|
| pvMemoryID | void* | Identifier of the memory area |
| ppvMemoryPtr | void** | Memory pointer |
| pulMemorySize | uint32_t* | Complete size of the mapped memory |
| ulChannel | uint32_t* | Requested channel number |
| pulChannelStartOffset | uint32_t* | Start offset of the requested channel |
| pulChannelSize | uint32_t* | Memory size of the requested channel |

*Table 26: Memory information (MEMORY_INFORMATION structure)*

### MEMORY_INFORMATION structure

```
/*****************************************************************************/
/*! Memory Information structure                                             */
/*****************************************************************************/
typedef __CIFx_PACKED_PRE struct MEMORY_INFORMATIONtag
{
  void*         pvMemoryID;           /*!< Identification of the memory area      */
  void**        ppvMemoryPtr;         /*!< Memory pointer                         */
  uint32_t*     pulMemorySize;        /*!< Complete size of the mapped memory     */
  uint32_t      ulChannel;            /*!< Requested channel number               */
  uint32_t*     pulChannelStartOffset;/*!< Start offset of the requested channel  */
  uint32_t*     pulChannelSize;       /*!< Memory size of the requested channel   */
} __CIFx_PACKED_POST MEMORY_INFORMATION;
```

### Example

```
//=============================================================================
// Test memory pointer
//
//
//=============================================================================
void TestMemoryPointer( void)
{
  unsigned char abBuffer[100] = {0};

  // Open channel
  uint32_t        ulMemoryID            = 0;
  unsigned char*  pabDPMMemory          = NULL;
  uint32_t        ulMemorySize          = 0;
  uint32_t        ulChannelStartOffset  = 0;
  uint32_t        ulChannelSize         = 0;
  long            lRet                  = CIFX_NO_ERROR;

  MEMORY_INFORMATION tMemory = {0};
  tMemory.pvMemoryID            = &ulMemoryID;            // Identification of the memory area
  tMemory.ppvMemoryPtr          = (void**)&pabDPMMemory;  // Memory pointer
  tMemory.pulMemorySize         = &ulMemorySize;          // Complete size of the mapped memory
  tMemory.ulChannel             = CIFX_NO_CHANNEL;        // Requested channel number
  tMemory.pulChannelStartOffset = &ulChannelStartOffset;  // Start offset of the requested channel
  tMemory.pulChannelSize        = &ulChannelSize;         // Memory size of the requested channel

  // Open a DPM memory pointer
  lRet = xDriverMemoryPointer( NULL, 0, CIFX_MEM_PTR_OPEN, &tMemory);
  if(lRet != CIFX_NO_ERROR)
  {
    // Failed to get the memory mapping
    ShowError( lRet);
  } else
  {
    // We have a memory mapping                                             1
    // Read 100 Bytes
    memcpy( abBuffer, pabDPMMemory, sizeof(abBuffer));

    memcpy( pabDPMMemory, abBuffer, sizeof(abBuffer));
  }

  // Return the DPM memory pointer
  lRet = xDriverMemoryPointer( NULL, 0, CIFX_MEM_PTR_CLOSE, &tMemory);
  ShowError( lRet);

}
```

# 4.7 System Device-specific functions

The system device is an additional device created by the device driver for each card. The corresponding data area in the DPM is called system channel. All global board information is located in this channel and all functions of the system device are related to the whole card.

For example the processing of a system reset, downloading a channel firmware etc. Downloads are processed via an own mailbox system which is independently from the communication channels.

The device driver uses the system channel for administrative functions (e.g. card start-up) or to process a card reset.

Usually an application has not to work with the system channel as long as it is designed to work with a specific communication channel or fieldbus system.

## 4.7.1 xSysdeviceOpen

Open a connection to a system device on the passed board.

### Function call

```
int32_t  xSysdeviceOpen(  CIFXHANDLE      hDriver,
                          char*           szBoard,
                          CIFXHANDLE*     phSysdevice);
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hDriver | CIFXHANDLE | Handle of the driver |
| szBoard | String | Identifier for the Board. Can by cifX<Board Number> or the associated alias. |
| phSysdevice | CIFXHANDLE* | Returned handle to the system device, to be used on all other sysdevice functions |

*Table 27: xSysdeviceOpen arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### 4.7.2 xSysdeviceClose

Close a connection to a system device.

#### Function call

```
int32_t xSysdeviceClose( CIFXHANDLE hSysdevice)
```

#### Arguments

| Argument | Data type | Description |
|---|---|---|
| hSysdevice | CIFXHANDLE | Handle of the system device that is to be closed. |

*Table 28: xSysdeviceClose arguments*

#### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.7.3 xSysdeviceInfo

Query information about the opened system device.

### Function call

```
int32_t xSysdeviceInfo(    CIFXHANDLE      hSysdevice,
                           uint32_t        ulCmd,
                           uint32_t        ulSize,
                           void*           pvInfo)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hSysdevice | CIFXHANDLE | Handle of the system device |
| ulCmd | uint32_t | Available Commands:<br>**1 = CIFX_INFO_CMD_SYSTEM_INFORMATION**<br>**2 = CIFX_INFO_CMD_SYSTEM_INFO_BLOCK**<br>**3 = CIFX_INFO_CMD_SYSTEM_CHANNEL_BLOCK**<br>**4 = CIFX_INFO_CMD_SYSTEM_CONTROL_BLOCK**<br>**5 = CIFX_INFO_CMD_SYSTEM_STATUS_BLOCK** |
| ulSize | uint32_t | Size of the passed system info buffer |
| pvInfo | void* | Pointer to SYSTEM_CHANNEL_INFORMATION structure for returned data |

*Table 29: xSysdeviceInfo arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.7.4 xSysdeviceReset

This function performs a firmware restart. Depending on the hardware and the implementation in the firmware, this is a software restart or a complete hardware reset.

Usually a software reset is performed.

**Note:** All channels will be reset.

### Function call

```
int32_t xSysdeviceReset(   CIFXHANDLE hSysdevice,
                           uint32_t   ulTimeout)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hSysdevice | CIFXHANDLE | Handle of the system device |
| ulTimeout | uint32_t | Timeout in ms to wait for reset to complete |

*Table 30: xSysdeviceReset arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.7.5 xSysdeviceResetEx

This function performs a firmware restart and allows to define a restart mode (*ulMode*), describing further functions executed by the firmaware, during the restart.

Depending on the hardware and the implementation in the firmware, this is a software restart or a complete hardware reset.

**Note:** All channels will be reset.

### Function call

```
int32_t xSysdeviceResetEx(  CIFXHANDLE     hSysdevice,
                            uint32_t       ulTimeout,
                            uint32_t       ulMode)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hSysdevice | CIFXHANDLE | Handle of the system device |
| ulTimeout | uint32_t | Timeout in ms to wait for reset to complete |
| ulMode | uint32_t | Reset Mode & Parameter<br>See following tables: Table 32 and Table 33. |

*Table 31: xSysdeviceResetEx arguments*

### ulMode for netX 90/4000/4100 only

| Bit No. | Definition / Description |
|---|---|
| 31..9 | empty / undefined |
| 8 | Delete complete remanent data area after reset.<br>1 = Remanent data area will be deleted.<br>0 = Remanent data area will not be deleted.<br>This option is only available for the modes BOOTSTART and UPDATESTART. In mode BOOTSTART, the remanent area is deleted when the maintenance firmware starts up. In mode UPDATESTART, the remanent is only deleted after a successful firmware installation. |
| 7..4 | Reset parameter for mode “update start”<br>Arguments that will be evaluated during “update start”.<br>0x0 ... 0xF = Specifies the firmware variant to be installed. 0x0 corresponds to VAR0, 0x1 corresponds to VAR1, etc. |

| | |
|---|---|
| 3..0 | **Reset Mode** |
| | `0 = CIFX_RESETEX_SYSTEMSTART`<br>**System start**<br>The system start will perform a reset (coldstart) of the device and will start the installed firmware again. |
| | `1 = reserved` |
| | `2 = CIFX_RESETEX_BOOTSTART`<br>**Boot start**<br>**Note:** The boot start is usable for Flash-based devices only.<br>The boot start will perform a reset of the device and will start the maintenance firmware. The boot start can be used to activate the maintenance firmware without starting an update process (idle mode). |
| | `3 = CIFX_RESETEX_UPDATESTART`<br>**Update start**<br>The update start will perform a reset of the device and will start the maintenance firmware.<br>If a valid update file is available, it will be automatically processed and installed.<br>If no update file is available or if the update file is not valid, the maintenance firmware will change into error mode, changes the SYS LED (to yellow on) and sets an error code (e.g. ERR_HIL_NOT_AVAILABLE, 0xC0001152). |
| Other values are reserved | |

*Table 32: xSysdeviceResetEx: Argument ulMode (netX 90/4000/4100)*

### ulMode for netX 10/50/51/52/100/500 only

| Bit No. | Definition / Description |
|---|---|
| 31..9 | empty / undefined |
| 8..4 | Without function. |
| 3..0 | Reset Mode |
| | `0 = CIFX_RESETEX_SYSTEMSTART`<br>**System start**<br>The system start will perform a reset (coldstart) of the device and will start the installed firmware again. |
| | `1 = reserved` |
| | `2 = CIFX_RESETEX_BOOTSTART`<br>**Boot start**<br>**Note:** The boot start is usable for Flash-based devices only.<br>This will activate the Second Stage Boot Loader (if available in the device). The Second Stage Boot Loader does not start the firmware. |
| | `3 = CIFX_RESETEX_UPDATESTART`<br>**Update start**<br>Function not available. |
| Other values are reserved | |

*Table 33: xSysdeviceResetEx: Argument ulMode (netX 10/50/51/52/100/500)*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.7.6 xSysdeviceBootstart

Performs a boot start on the hardware. This is necessary if the Second Stage Boot Loader (netX 10/50/51/52/100/500) or the Maintenance Firmware (netX 90/4000/4100) has to be activated while an executable Firmware is available.

**Note:** All channels will be reset.

**Note:** This function is only available on so called Flash-based devices where the Second Stage Boot Loader or the Maintenance Firmware is stored in the Flash of the hardware.

**Note:** For Maintenance Firmware (netX 90/4000/4100): The boot start activates the maintenance firmware without starting an update process (idle mode). To perform a firmware update, use the `xSysdeviceResetEx` function with “Reset Mode” parameter set to value `CIFX_RESETEX_UPDATESTART`.

### Function call

```
int32_t xSysdeviceBootstart(    CIFXHANDLE hSysdevice,
                                uint32_t   ulTimeout)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hSysdevice | CIFXHANDLE | Handle of the system device |
| ulTimeout | uint32_t | Timeout in ms to wait for reset to complete |

*Table 34: xSysdeviceBootstart arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error is returned. For a list, see section *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.7.7 xSysdeviceGetMBXState

Retrieve the current load of the system device mailbox. This Function can be used to read the actual state of the channels send and receive mailbox, without accessing the mailbox itself.

**Note:** Mailboxes are used to pass asynchronous data back and forth between the hardware and the host system. The amount of concurrent active asynchronous commands is limited by the hardware.

### Function call

```
int32_t xSysdeviceGetMBXState(  CIFXHANDLE      hSysdevice,
                                uint32_t*       pulRecvPktCount,
                                uint32_t*       pulSendPktCount)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hSysdevice | CIFXHANDLE | Handle of the system device. |
| pulRecvPktCount | uint32_t* | Number of packets waiting to be received by Host |
| pulSendPktCount | uint32_t* | Number of packets the Host is able to send at once. |

*Table 35: xSysdeviceGetMBXState arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.7.8 xSysdevicePutPacket

Insert an asynchronous command (packet) into the system device send mailbox to send it to the hardware. This function uses the system device mailbox.

### Function call

```
int32_t xSysdevicePutPacket(    CIFXHANDLE      hSysdevice,
                                CIFX_PACKET*    ptSendPacket,
                                uint32_t        ulTimeout)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the system device. |
| ptSendPacket | CIFX_PACKET* | Packet to be send. Total data length is acquired through the ulLen element inside the structure. |
| ulTimeout | uint32_t | Time in ms to wait for the mailbox to get free.<br>0 means, do not wait |

*Table 36: xSysdevicePutPacket arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.7.9 xSysdeviceGetPacket

Retrieve an already waiting, asynchronous data packet from the system device receive mailbox.

### Function call

```
int32_t xSysdeviceGetPacket(    CIFXHANDLE      hSysdevice,
                                uint32_t        ulSize,
                                CIFX_PACKET*    ptRecvPacket,
                                uint32_t        ulTimeout)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hSysdevice | CIFXHANDLE | Handle of the system device. |
| ulSize | uint32_t | Size of the passed receive packet buffer |
| ptRecvPacket | CIFX_PACKET* | Buffer to returned packet |
| ulTimeout | uint32_t | Time in ms to wait for a receive message.<br>0 means, do not wait |

*Table 37: xSysdeviceGetPacket arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.7.10 xSysdeviceDownload

Downloading files to the board via the system device.

Due to the limited size of the mailbox these downloads are slower than using the channels mailbox and should only be used if the channel's firmware is not running yet.

Download mode definitions can be found in the cifXUser.h file.

### Available download modes

| Definition | Mode | Description |
|---|---|---|
| DOWNLOAD_MODE_FIRMWARE | 1 | Download a firmware file (file extension *.nxf) |
| DOWNLOAD_MODE_CONFIG | 2 | Download a fieldbus configuration file (file extension *.nxd) |
| DOWNLOAD_MODE_FILE | 3 | Download any file (file extension *.*) |
| DOWNLOAD_MODE_BOOTLOADER | 4 | Special download mode only supported by the Second Stage Boot Loader to update itself in Flash memory |
| DOWNLOAD_MODE_LICENSECODE | 5 | Download a license file (file extension *.nxl)<br>License files containing license flags which are stored on the hardware. |
| DOWNLOAD_MODE_MODULE | 6 | Download a firmware module file (file extension *.nxo)<br>Firmware modules are dynamically loadable and require a so-called base firmware |

*Table 38: Download modes*

**Note:** **xSysdeviceDownload() is not working if called with *DOWNLOAD_MODE_FIRMWARE*** on so called "**RAM-based devices**" like on CIFX PCI/PCIe cards or devices where the firmware is not stored into Flash.

**It is not possible** to use this function to download a firmware, because of the circumstance that a firmware running in RAM is not able to update itself in RAM at the same time.

### Function call

```
int32_t xSysdeviceDownload(    CIFXHANDLE               hSysdevice,
                               uint32_t                 ulChannel,
                               uint32_t                 ulMode,
                               char*                    pszFileName,
                               uint8_t*                 pabFileData,
                               uint32_t                 ulFileSize,
                               PFN_PROGRESS_CALLBACK    pfnCallback,
                               PFN_RECV_PKT_CALLBACK    pfnRecvPktCallback,
                               void*                    pvUser)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hSysdevice | CIFXHANDLE | Handle of the system device, the download is performed on. |
| ulChannel | uint32_t | Number of the channel, to receive the file |
| ulMode | uint32_t | Download mode (see above) |
| pszFileName | char* | Short file name of the passed data on the device. |
| pabFileData | uint8_t* | File data to download. |
| ulFileSize | uint32_t | Length of the file in bytes |
| pfnCallback | PFN_PROGRESS_CALLBACK | Callback function to indicate the download progress. Passing NULL will suppress callbacks. |
| pfnRecvPktCallback | PFN_RECV_PKT_CALLBACK | Callback function to receive unhandled packets during this function. Passing NULL will suppress callbacks and discard received packets that do not belong to the file download. |
| pvUser | void* | User parameter which is passed on every callback |

*Table 39: xSysdeviceDownload arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.7.11 xSysdeviceFindFirstFile

Start enumerating a directory on the device. This call will deliver the first directory/file entry on the device if available.

### Function call

```
int32_t xSysdeviceFindFirstFile(     CIFXHANDLE                hSysdevice,
                                     uint32_t                  ulChannel,
                                     CIFX_DIRECTORYENTRY*      ptDirectoryInfo,
                                     PFN_RECV_PKT_CALLBACK     pfnRecvPktCallback,
                                     void*                     pvUser)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hSysdevice | CIFXHANDLE | Handle of the system device. |
| ulChannel | uint32_t | Channel number |
| ptDirectoryInfo | CIFX_DIRECTORYENTRY* | Returned first directory entry.<br>The szFilename entry can be used to start enumerating on a special file.<br>Must be a zero length string to enumerate the whole directory. |
| pfnRecvPktCallback | PFN_RECV_PKT_CALLBACK | Callback function to receive unhandled packets during this function. Passing NULL will suppress callbacks and discard received packets that do not belong to the file search. |
| pvUser | void* | User parameter which is passed on every callback |

*Table 40: xSysdeviceFindFirstFile arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.7.12 xSysdeviceFindNextFile

Continue enumerating a directory on the device. This function must be called with a previously returned directory entry structure from *xSysdeviceFindFirstFile()*.

### Function call

```
int32_t xSysdeviceFindNextFile (     CIFXHANDLE               hSysdevice,
                                     uint32_t                 ulChannel,
                                     CIFX_DIRECTORYENTRY*     ptDirectoryInfo,
                                     PFN_RECV_PKT_CALLBACK    pfnRecvPktCallback,
                                     void*                    pvUser)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hSysdevice | CIFXHANDLE | Handle of the system device. |
| ulChannel | uint32_t | Channel number |
| ptDirectoryInfo | CIFX_DIRECTORYENTRY* | Returned directory entry. |
| pfnRecvPktCallback | PFN_RECV_PKT_CALLBACK | Callback function to receive unhandled packets during this function. Passing NULL will suppress callbacks and discard received packets that do not belong to the file search. |
| pvUser | void* | User parameter which is passed on every callback |

*Table 41: xSysdeviceFindNextFile arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.7.13 xSysdeviceUpload

Upload a given file from the device.

### Function call

```
int32_t xSysdeviceUpload( CIFXHANDLE             hSysdevice,
                          uint32_t               ulChannel,
                          uint32_t               ulMode,
                          char*                  pszFilename,
                          uint8_t*               pabFileData,
                          uint32_t*              pulFileSize,
                          PFN_PROGRESS_CALLBACK  pfnCallback,
                          PFN_RECV_PKT_CALLBACK  pfnRecvPktCallback,
                          void*                  pvUser)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hSysdevice | CIFXHANDLE | Handle of the system device. |
| ulChannel | uint32_t | Channel number of the file |
| ulMode | uint32_t | Upload Mode (see section *xSysdeviceUpload* on page 45) |
| pszFilename | char* | Name of the file to upload (must conform to 8.3 filename rules) |
| pabFileData | uint8_t* | Buffer to place uploaded data in |
| pulFileSize | uint32_t * | [in] Size of the buffer,<br>[out] Number of uploaded bytes |
| pfnCallback | PFN_PROGRESS_CALLBACK | Callback function to indicate the download progress. Passing NULL will suppress callbacks. |
| pfnRecvPktCallback | PFN_RECV_PKT_CALLBACK | Callback function to receive unhandled packets during this function. Passing NULL will suppress callbacks and discard received packets that do not belong to the file upload. |
| pvUser | void* | User parameter which is passed on every callback |

*Table 42: xSysdeviceUpload arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.7.14 xSysdeviceExtendedMemory

netX based PCI hardware is able to offer a second PCI memory window used to accesses additional hardware memory, independent of the existing Hilscher dual-port-memory resource.

Depending on the netX hardware, the type of memory resource could differ. Current hardware offers a MRAM (Magneto resistive Random Access Memory) resource.

The type of additional memory, assembled on the hardware, is defined by information in the hardware security memory. The information is used by the boot loader and firmware to detect and initialize access to the additional memory and the information is also stored in the NETX_SYSTEM_STATUS_BLOCK (see *ulHWFeatures*) to be accessible by a user application.

The *xSysdeviceExtendedMemory()* function offers a command parameter to allowing reading information and getting/returning the pointer to the extended memory.

### Function call

```
int32_t xSysdeviceExtendedMemory(    CIFXHANDLE                          hSysdevice,
                                     uint32_t                            ulCmd,
                                     CIFX_EXTENDED_MEMORY_INFORMATION*   ptExtMemData);
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hSysdevice | CIFXHANDLE | Handle of the system device. |
| ulCmd | uint32_t | Extended Memory Commands:<br>**1 = CIFX_GET_EXTENDED_MEMORY_INFO**<br>**2 = CIFX_GET_EXTENDED_MEMORY_POINTER**<br>**3 = CIFX_FREE_EXTENDED_MEMORY_POINTER** |
| ptExtMemData | CIFX_EXTENDED_MEMORY_INFORMATION* | Pointer to a extended memory structure, to store /pass information between driver and application |

*Table 43: xSysdeviceExtendedMemory arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### Description of the CIFX_EXTENDED_MEMORY_INFORMATION Structure

| Value | Data type | Description |
|---|---|---|
| pvMemoryID | void* | Identifier of the memory area |
| pvMemoryPtr | void* | Memory pointer to the extended memory area |
| ulMemorySize | uint32_t | Size of the extended memory area |
| ulMemoryType | uint32_t | Type of the extended memory area (e.g. MRAM) |

**ulMemoryInformation**

| | | | | | | | | | Extended Memory | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **31..16** | **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** | |
| | | | | | | | | | | | | | **RAM Type:**<br>0 = None<br>1 = MRAM 64*16 Bit (1 MBit/128 KB) | | | | |
| | | | | | | | | | | | Reserved | | | | | | |
| | | | | | | | | | **Access Type:**<br>00 = No access<br>01 = external access (host)<br>10 = internal access<br>11 = external and internal access reserved | | | | | | | | |
| Unused set to 0 | | | | | | | | | | | | | | | | | |

*Table 44: ulMemoryInformation*

**Note:** *ulMemoryType* defines the type of the assembled/offered memory by the hardware. The type is defined in the hardware security memory.

Available definitions (see rcX_User.h)

| | |
|---|---|
| #define RCX_SYSTEM_EXTMEM_TYPE_MSK | 0x0000000F |
| #define RCX_SYSTEM_EXTMEM_TYPE_NONE | 0x00000000 |
| #define RCX_SYSTEM_EXTMEM_TYPE_MRAM_128K | 0x00000001 |
| | |
| #define RCX_SYSTEM_EXTMEM_ACCESS_MSK | 0x000000C0 |
| #define RCX_SYSTEM_EXTMEM_ACCESS_NONE | 0x00000000 |
| #define RCX_SYSTEM_EXTMEM_ACCESS_EXTERNAL | 0x00000040 |
| #define RCX_SYSTEM_EXTMEM_ACCESS_INTERNAL | 0x00000080 |
| #define RCX_SYSTEM_EXTMEM_ACCESS_BOTH | 0x000000C0 |

**Note:** RCX_SYSTEM_EXTMEM_ACCESS_EXTERNAL defines exclusive access by a host application while RCX_SYSTEM_EXTMEM_ACCESS_INTERNAL defines exclusive access by the firmware.
RCX_SYSTEM_EXTMEM_ACCESS_BOTH defines access for the firmware and host application. In this case, first half of the memory is reserved for the host application, starting at offset 0 and the second half of the memory is used by the firmware, starting at offset memory size / 2.

### CIFX_EXTENDED_MEMORY_INFORMATION structure

```
/*****************************************************************************/
/*! Extended memory information structure                                   */
/*****************************************************************************/
typedef __CIFx_PACKED_PRE struct CIFX_EXTENDED_MEMORY_INFORMATIONtag
{
  void*          pvMemoryID;           /*!< Identification of the memory area       */
  void*          pvMemoryPtr;          /*!< Memory pointer                          */
  uint32_t       ulMemorySize;         /*!< Memory size of the Extended memory area */
  uint32_t       ulMemoryType;         /*!< Memory type information                 */

} __CIFx_PACKED_POST CIFX_EXTENDED_MEMORY_INFORMATION;
```

### Example

```
//=============================================================================
// Test memory pointer
//
//
//=============================================================================
void TestExtendedMemoryPointer( void)
{
  CIFXHANDLE    hSysdevice    = NULL;
  int32_t       lRet          = CIFX_NO_ERROR;
  uint8_t       abBuffer[100] = {0};

  printf("\n--- Test Extended Memory Pointer ---\r\n");

  lRet = xSysdeviceOpen( NULL, "CIFX0", &hSysdevice);

  if ( CIFX_NO_ERROR != lRet)
  {
    ShowError( lRet);
  } else
  {
    CIFX_EXTENDED_MEMORY_INFORMATION tExtMemory = {0};

    // Open a DPM memory pointer
    lRet = xSysdeviceExtendedMemory( hSysdevice, CIFX_GET_EXTENDED_MEMORY_INFO,
                                     &tExtMemory);
    if(lRet != CIFX_NO_ERROR)
    {
      // Failed to get the memory mapping
      ShowError( lRet);
    } else
    {
      /* Get an extended memory pointer */
      lRet = xSysdeviceExtendedMemory( hSysdevice, CIFX_GET_EXTENDED_MEMORY_POINTER,
                                       &tExtMemory);
      if(lRet != CIFX_NO_ERROR)
      {
        // Failed to get the memory mapping
        ShowError( lRet);
      }else
      {
        // We have a memory mapping
        uint8_t* pbExtMem = (uint8_t*)tExtMemory.pvMemoryPtr;

        while( 1 == 1)
        {
          // Read 100 Bytes
          memcpy( abBuffer, pbExtMem, sizeof(abBuffer));

          printf("Read data from the extended memory (%d bytes):\n",
                 sizeof(abBuffer));
          DumpData( abBuffer, sizeof(abBuffer));

          printf("Increment the read data:\n");
```

```
          for ( uint32_t ulIdx =0; ulIdx < sizeof(abBuffer); ulIdx++)
          {
            abBuffer[ulIdx] +=1;
          }

          printf("Write data back to the extened memory:\n");
          memcpy( pbExtMem, abBuffer, sizeof(abBuffer));

          printf("Type (A) for again and (S) to stop the extended read/write test:\n");
          if( 'S' == (toupper (_getch())) )
          {
            break;
          }
        }
        lRet = xSysdeviceExtendedMemory( hSysdevice, CIFX_FREE_EXTENDED_MEMORY_POINTER,
                                         &tExtMemory);
        if(lRet != CIFX_NO_ERROR)
        {
          // Failed to free the memory mapping
          ShowError( lRet);
        }
      }
    }

    /* Close the system device */
    lRet = xSysdeviceClose( hSysdevice);
    if ( CIFX_NO_ERROR != lRet)
    {
      ShowError( lRet);
    }
  }

  // Test done
  printf("\n Extended Memory Pointer test done\r\n");

}
```

# 4.8 Channel-specific functions

Channels (Communication Channels) are the access to a specific fieldbus system running on the netX hardware. Each channel has its own memory area in the DPM and can be handled independently from other channels.

## 4.8.1 xChannelOpen

Open a connection to a communication / user channel on the given board.

### Function call

```
int32_t xChannelOpen(      CIFXHANDLE      hDriver,
                           char*           szBoard,
                           uint32_t        ulChannel,
                           CIFXHANDLE*     phChannel)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hDriver | CIFXHANDLE | Handle to the driver (returned by xDriverOpen) |
| szBoard | char* | Identifier for the Board. Can by cifX<BoardNumber> or the associated alias. |
| ulChannel | uint32_t | Channel number to open |
| phChannel | CIFXHANDLE* | Returned handle to the channel, to be used on all other channel functions |

*Table 45: xChannelOpen arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### 4.8.2 xChannelClose

Close a connection to a communication channel.

#### Function call

```
int32_t xChannelClose(    CIFXHANDLE hChannel)
```

#### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel that is to be closed. |

*Table 46: xChannelClose arguments*

#### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### 4.8.3 xChannelDownload

Download a file via the communication channel.

Download modes are defined and described in the xSysdeviceDownload() function (see section *xSysdeviceUpload* on page 45)

**Note:** **xChannelDownload() is not working if called with** ***DOWNLOAD_MODE_FIRMWARE*** on so called "**RAM based devices**" like on CIFX PCI/PCIe cards or devices where the firmware is not stored into Flash.

**It is not possible** to use this function to download a firmware, because of the circumstance that a firmware running in RAM is not able to update itself in RAM at the same time.

#### Function call

```
int32_t xChannelDownload( CIFXHANDLE               hChannel,
                          uint32_t                 ulMode,
                          char*                    pszFileName,
                          uint8_t*                 pabFileData,
                          uint32_t                 ulFileSize,
                          PFN_PROGRESS_CALLBACK    pfnCallback,
                          PFN_RECV_PKT_CALLBACK    pfnRecvPktCallback,
                          void*                    pvUser)
```

#### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel, the download is performed on. |
| ulMode | uint32_t | Download mode (see (see section *xSysdeviceUpload* on page 45) |
| pszFileName | char* | Short file name of the passed data on the device. |
| pabFileData | uint8_t* | File data to download. |
| ulFileSize | uint32_t | Length of the downloaded file |
| pfnCallback | PFN_PROGRESS_CALLBACK | Callback function to indicate the download progress. Passing NULL will suppress callbacks. |
| pfnRecvPktCallback | PFN_RECV_PKT_CALLBACK | Callback function to receive unhandled packets during this function. Passing NULL will suppress callbacks and discard all received packets that do not belong to the file download. |
| pvUser | void* | User parameter which is passed on every callback |

*Table 47: xChannelDownload arguments*

#### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.4 xChannelFindFirstFile

Start enumerating a directory on the channel. This call will deliver the first directory/file entry on the channel if available.

### Function call

```
int32_t xChannelFindFirstFile ( CIFXHANDLE              hChannel,
                                CIFX_DIRECTORYENTRY*    ptDirectoryInfo,
                                PFN_RECV_PKT_CALLBACK   pfnRecvPktCallback,
                                void*                   pvUser)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the communication channel. |
| ptDirectoryInfo | CIFX_DIRECTORYENTRY* | Returned first directory entry.<br>The szFilename entry can be used to start enumerating on a special file.<br>Must be a zero length string to enumerate the whole directory. |
| pfnRecvPktCallback | PFN_RECV_PKT_CALLBACK | Callback function to receive unhandled packets during this function. Passing NULL will suppress callbacks and discard all received packets that do not belong to the file find. |
| pvUser | void* | User parameter which is passed on every callback |

*Table 48: xChannelFindFirstFile arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.5 xChannelFindNextFile

Continue enumerating a directory on the channel. This function must be called with a previously returned directory entry structure from *xChannelFindFirstFile()*.

### Function call

```
int32_t xChannelFindNextFile (  CIFXHANDLE                 hChannel,
                                CIFX_DIRECTORYENTRY*       ptDirectoryInfo,
                                PFN_RECV_PKT_CALLBACK      pfnRecvPktCallback,
                                void*                      pvUser)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the communication channel. |
| ptDirectoryInfo | CIFX_DIRECTORYENTRY* | Returned directory entry. |
| pfnRecvPktCallback | PFN_RECV_PKT_CALLBACK | Callback function to receive unhandled packets during this function. Passing NULL will suppress callbacks and discard all received packets that do not belong to the file find. |
| pvUser | void* | User parameter which is passed on every callback |

*Table 49: xChannelFindNextFile arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.6 xChannelUpload

Upload a given file from the communication channel.

### Function call

```
int32_t xChannelUpload (  CIFXHANDLE                hChannel,
                          uint32_t                  ulMode,
                          char*                     pszFileNam e,
                          uint8_t*                  pabFileData,
                          uint32_t*                 pulFileSize,
                          PFN_PROGRESS_CALLBACK     pfnCallback,
                          PFN_RECV_PKT_CALLBACK     pfnRecvPktCallback,
                          void*                     pvUser)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the communication channel. |
| ulMode | uint32_t | Upload Mode (see section *xSysdeviceUpload* on page 45) |
| pszFileName | char* | Name of the file to upload (must conform to 8.3 filename rules) |
| pabFileData | uint8_t* | Buffer to place uploaded data in |
| pulFileSize | uint32_t * | [in] Size of the buffer,<br>[out] Number of uploaded bytes |
| pfnCallback | PFN_PROGRESS_CALLBACK | Callback function to indicate the download progress. Passing NULL will suppress callbacks. |
| pfnRecvPktCallback | PFN_RECV_PKT_CALLBACK | Callback function to receive unhandled packets during this function. Passing NULL will suppress callbacks and discard all received packets that do not belong to the file upload. |
| pvUser | void* | User parameter which is passed on every callback |

*Table 50: xChannelUpload arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.7 xChannelGetMBXState

Retrieve the current load of the given communication channel mailbox. This Function can be used to read the actual state of the channels send and receive mailbox without accessing the mailbox itself.

**Note:** Mailboxes are used to pass asynchronous data back and forth between the hardware and the host system. The amount of concurrent active asynchronous commands is limited by the hardware.

### Function call

```
int32_t xChannelGetMBXState(    CIFXHANDLE      hChannel,
                                uint32_t*       pulRecvPktCount,
                                uint32_t*       pulSendPktCount)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| pulRecvPktCount | uint32_t* | Number of packets waiting to be received by Host |
| pulSendPktCount | uint32_t* | Number of packets the Host is able to send at once. |

*Table 51: xChannelGetMBXState arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.8 xChannelPutPacket

Insert an asynchronous data packet into the given communication channel send mailbox to send it to the hardware.

### Function call

```
int32_t xChannelPutPacket( CIFXHANDLE      hChannel,
                           CIFX_PACKET*    ptSendPacket,
                           uint32_t        ulTimeout)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ptSendPacket | CIFX_PACKET* | Packet to be send. Total data length is acquired through the ulLen element inside the structure. |
| ulTimeout | uint32_t | Time in ms to wait for the mailbox to get free.<br>0 means, do not wait |

*Table 52: xChannelPutPacket arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.9 xChannelGetPacket

Retrieve an already waiting, asynchronous data packet from the given communication channel receive mailbox.

### Function call

```
int32_t xChannelGetPacket( CIFXHANDLE     hChannel,
                           uint32_t       ulBufferSize,
                           CIFX_PACKET*   ptRecvPacket,
                           uint32_t       ulTimeout)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulBufferSize | uint32_t | Size of the passed receive packet buffer |
| ptRecvPacket | CIFX_PACKET* | Buffer to returned packet |
| ulTimeout | uint32_t | Time in ms to wait for a receive message.<br>0 means, do not wait |

*Table 53: xChannelGetPacket arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.10 xChannelGetSendPacket

Retrieve the actual data packet send by the host, from the communication channel send mailbox. This function is none destructive. It does not guarantee any data consistency, because data are read without any synchronization.

The function is mainly used for debugging aids.

### Function call

```
int32_t xChannelGetSendPacket(  CIFXHANDLE      hChannel,
                                uint32_t        ulBufferSize,
                                CIFX_PACKET*    ptRecvPacket)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulBufferSize | uint32_t | Size of the passed packet buffer |
| ptRecvPacket | CIFX_PACKET* | Buffer to returned send packet |

*Table 54: xChannelGetSendPacket arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### 4.8.11 xChannelReset

Reset the given communication channel. The reset function offers a following two modes:

- CIFX_ CHANNELINIT Re-initialization of a communication channel
- CIFX_ SYSTEMSTART Restart the whole card

#### Function call

```
int32_t xChannelReset(     CIFXHANDLE      hChannel,
                           uint32_t        ulResetMode,
                           uint32_t        ulTimeout)
```

#### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulResetMode | uint32_t | Type of reset to be performed |
| ulTimeout | uint32_t | Time in ms to wait for the channel to be ready again |

*Table 55: xChannelReset arguments*

#### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.12 xChannelInfo

Retrieve the global communication channel information.

### Function call

```
int32_t xChannelInfo(      CIFXHANDLE      hChannel,
                           uint32_t        ulSize,
                           void*           pvChannelInfo)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulSize | uint32_t | Length of the passed buffer. |
| pvChannelInfo | void* | Pointer to a CHANNEL_INFORMATION structure, for returned data |

*Table 56: xChannelInfo arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.13 xChannelIOInfo

Retrieve I/O information about the communication channel.

### Function call

```
int32_t xChannelIOInfo(    CIFXHANDLE      hChannel,
                           uint32_t        ulCmd,
                           uint32_t        ulAreaNumber,
                           uint32_t        ulSize,
                           void*           pvData)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulCmd | uint32_t | 1 = **CIFX_IO_INPUT_AREA**<br>2 = **CIFX_IO_OUTPUT_AREA** |
| ulAreaNumber | uint32_t | Area number to query information for |
| ulSize | uint32_t | Length of the passed buffer. |
| pvData | void* | Pointer to a CHANNEL_IO_INFORMATION structure, for returned data |

*Table 57: xChannelIOInfo arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.14 xChannelWatchdog

Enable, trigger or disable the host watchdog. The watchdog function is used by a communication channel to supervise the processing of the user application. If the watchdog is configured it will be activated with the first call of the function *xChannelWatchdog()* passing the command CIFX_WATCHDOG_START. Once activated, the application must trigger it cyclically, during the configured watchdog time. The watchdog supervision is deactivated by passing CIFX_WATCHDOG_STOP in the call of *xChannelWatchdog()*.

### Function call

```
int32_t xChannelWatchdog( CIFXHANDLE      hChannel,
                          uint32_t        ulCmd,
                          uint32_t*       pulTrigger)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulCmd | uint32_t | Watchdog Command<br>Start and trigger the watchdog monitoring<br>**1 = CIFX_WATCHDOG_START**<br>Stop the watchdog monitoring<br>**0 = CIFX_WATCHDOG_STOP** |
| pulTrigger | uint32_t* | Last trigger value from the hardware |

*Table 58: xChannelWatchdog arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.15 xChannelConfigLock

Lock the configuration of the channel against modification. If the configuration is locked, the fieldbus stack does not allow doing a configuration update.

### Function call

```
int32_t xChannelConfigLock(     CIFXHANDLE      hChannel,
                                uint32_t        ulCmd,
                                uint32_t*       pulState,
                                uint32_t        ulTimeout)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulCmd | unit32_t | Configuration Lock Command<br>Unlock configuration<br>**0 = CIFX_CONFIGURATION_UNLOCK**<br>Lock configuration<br>**1 = CIFX_CONFIGURATION_LOCK**<br>Read the locking state<br>**2 = CIFX_CONFIGURATION_GETLOCKSTATE** |
| pulState | uint32_t* | returned state, if the CIFX_CONFIGURATION_GETLOCKSTATE command is used |
| ulTimeout | uint32_t | Timeout in ms to wait for configuration lock becoming active |

*Table 59: xChannelConfigLock arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.16 xChannelHostState

Toggle the *'Application Ready State Flag'* in the communication channel host handshake flags. This function is used to signal a communication stack the presents of a user application.

How the fieldbus stack uses the information is stack depending. Usually the stack will use the information to verify if the I/O data in the I/O image are valid.

### Function call

```
int32_t xChannelHostState( CIFXHANDLE      hChannel,
                           uint32_t        ulCmd,
                           uint32_t*       pulState,
                           uint32_t        ulTimeout)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulCmd | uint32_t | Host State Command<br>Clears the application ready flag<br>**0 = CIFX_HOST_STATE_NOT_READY**<br>Sets the application ready flag<br>**1 = CIFX_HOST_STATE_READY**<br>Read the current state of the flag<br>**2 = CIFX_HOST_STATE_READ** |
| pulState | uint32_t* | Returns the actual state of the application ready flag if CIFX_HOST_STATE_READ command is used |
| ulTimeout | uint32_t | Timeout in milliseconds.<br>If not 0, the function will wait the given time until the state is changed |

*Table 60: xChannelHostState arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.17 xChannelBusState

Toggle the 'Bus State Flag' in the communication channel handshake flags. Using this flag, the host application allows or disallows the firmware to open network connections. If set (CIFX_BUS_STATE_ON), the netX firmware tries to open network connections; if cleared (CIFX_BUS_STATE_OFF), no connections are allowed and open connections are closed (See reference [1] for further information).

In generally a fieldbus stack allows the configuration of the field bus start-up behavior. This can be either 'automatic startup' or 'controlled startup'. If the stack is configured in 'controlled startup' (i.e. the 'Bus State Flag' is cleared) it will not activate the bus communication until it receives a CIFX_BUS_STATE_ON state in its handshake flags.

**Note:** Setting the ‘Bus State flag’ to CIFX_BUS_STATE_ON successfully does not necessarily mean that the fieldbus stack has established a connection to the fieldbus system. The routine will signal an absent connection by returning the error code CIFX_DEV_NO_COM_FLAG (even though toggle of the ‘Bus state flag’ has succeeded).

### Function call

```
int32_t xChannelBusState( CIFXHANDLE      hChannel,
                          uint32_t        ulCmd,
                          uint32_t*       pulState,
                          uint32_t        ulTimeout)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulCmd | uint32_t | Bus State Commands:<br>Clears the BUS state flag<br>**0 = CIFX_BUS_STATE_OFF**<br>Sets the bus state flag<br>**1 = CIFX_BUS_STATE_ON**<br>Read the actual state of the bus state flag<br>**2 = CIFX_BUS_STATE_GETSTATE** |
| pulState | uint32_t* | Actual state returned |
| ulTimeout | uint32_t | Timeout in milliseconds.<br>If not 0, the function will wait until the communication has reached the chosen state. |

*Table 61: xChannelBusState arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

CIFX_DEV_NO_COM_FLAG if the function succeeds but fieldbus stack does not communicate.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.18 xChannelControlBlock

Reading / writing the communication channel control block.

**Note:** Due to endianess conversion, offset parameter on big-endian targets must be set to '0'.

### Function call

```
int32_t xChannelControlBlock (  CIFXHANDLE      hChannel,
                                uint32_t        ulCmd,
                                uint32_t        ulOffset,
                                uint32_t        ulDataLen,
                                void*           pvData);
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulCmd | uint32_t | Control block commands:<br>Read the block area<br>**1 = CIFX_CMD_READ_DATA**<br>Write the block area<br>**2 = CIFX_CMD_WRITE_DATA** |
| ulOffset | uint32_t | Start offset in the block area<br>Big-endian targets: **0** |
| ulDataLen | uint32_t | Number of bytes to read |
| pvData | void* | User buffer |

*Table 62: xChannelControlBlock arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.19 xChannelCommonStatusBlock

Read the channels common status block.

**Note:** Writing of the common status block by an application is not allowed.

**Note:** Due to endianess conversion, offset parameter on big-endian targets must be set to '0'.

### Function call

```
int32_t xChannelCommonStatusBlock (  CIFXHANDLE      hChannel,
                                     uint32_t        ulCmd,
                                     uint32_t        ulOffset,
                                     uint32_t        ulDataLen,
                                     void*           pvData);
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulCmd | uint32_t | Status block commands:<br>Read the block area<br>**1 = CIFX_CMD_READ_DATA** |
| ulOffset | uint32_t | Start offset in the block area<br>Big-endian targets: **0** |
| ulDataLen | uint32_t | Number of bytes to read |
| pvData | void* | User buffer |

*Table 63: xChannelCommonStatusBlock arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.20 xChannelExtendedStatusBlock

Read the communication channels extended status block.

**Note:** Writing of the extended status block by an application is not allowed

### Function call

```
int32_t xChannelExtendedStatusBlock (     CIFXHANDLE      hChannel,
                                           uint32_t        ulCmd,
                                           uint32_t        ulOffset,
                                           uint32_t        ulDataLen,
                                           void*           pvData);
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulCmd | uint32_t | Extended status block commands:<br>Read the block area<br>**1 = CIFX_CMD_READ_DATA** |
| ulOffset | uint32_t | Start offset in the block area |
| ulDataLen | uint32_t | Number of bytes to read |
| pvData | void* | User buffer |

*Table 64: xChannelExtendedStatusBlock arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### 4.8.21 xChannelUserBlock

not implemented yet!

## 4.8.22 xChannelIORead

The function reads the input process data image of a communication channel and afterwards it instructs the fieldbus protocol to update input data image with actual (latest) fieldbus data.

If the function returns error CIFX_NO_COM_FLAG, the protocol stack does not have an active network connection and the input data are not up to date. The protocol stack must be started and configured (see errors CIFX_DEV_NOT_READY / CIFX_DEV_NOT_RUNNING).

**Note**: On the basis of the implementation, xChannelIORead() delivers the input data from the "last" call of xChannelIORead() and not the "latest" input data from the fieldbus system. This has the advantage that the function has not to wait for the data update of the fieldbus protocol which could need a significant time which are several hundred microseconds up to milliseconds depending to the configuration and the number of devices connected to the fieldbus system. But also the limitation that the age of the input data are depending of the cycle time used to call xChannelIORead().

### Function call

```
int32_t xChannelIORead(    CIFXHANDLE      hChannel,
                           uint32_t        ulAreaNumber,
                           uint32_t        ulOffset,
                           uint32_t        ulDataLen,
                           void*           pvData,
                           uint32_t        ulTimeout)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulAreaNumber | uint32_t | Number of the I/O Input area to get data from |
| ulOffset | uint32_t | Offset inside area to start reading data from |
| ulDataLen | uint32_t | Length of the data being retrieved |
| pvData | void* | Pointer to the return data buffer |
| ulTimeout | uint32_t | Timeout in ms to wait for I/O handshake completion of the channel (if configured) |

*Table 65: xChannelIORead arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### 4.8.23 xChannelIOWrite

The function writes the output process data image of a communication channel and instructs the fieldbus protocol to send the data to the fieldbus system.

The function can also be used to initialize the output data image the first time, before the fieldbus system starts the network data transfer. The fieldbus protocol will always be informed to update the internal output data image by the written data, even if the functions detects that no bus communication is running (error return CIFX_NO_COM_FLAG).

The only requirement is a started and configured fieldbus protocol (see errors CIFX_DEV_NOT_READY / CIFX_DEV_NOT_RUNNING).

**Note**: The function xChannelIOWrite() does not wait until the data are taken by the hardware or physically transferred by the fieldbus system, because this depends at least on the fieldbus connection and the cycle time of the fieldbus system which are usually unknown.

#### Function call

```
int32_t xChannelIOWrite(  CIFXHANDLE      hChannel,
                          uint32_t        ulAreaNumber,
                          uint32_t        ulOffset,
                          uint32_t        ulDataLen,
                          void*           pvData,
                          uint32_t        ulTimeout)
```

#### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulAreaNumber | uint32_t | Number of the I/O Output area to send data to |
| ulOffset | uint32_t | Offset inside area to start writing data to |
| ulDataLen | uint32_t | Length of the data being send |
| pvData | void* | Pointer to the send data buffer |
| ulTimeout | uint32_t | Timeout in ms to wait for I/O handshake completion of the channel (if configured) |

*Table 66: xChannelIOWrite arguments*

#### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.24 xChannelIOReadSendData

Read back the actual content of the output process data image from a communication channel.

### Function call

```
int32_t xChannelIOReadSendData( CIFXHANDLE      hChannel,
                                uint32_t        ulAreaNumber,
                                uint32_t        ulOffset,
                                uint32_t        ulDataLen,
                                void*           pvData)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulAreaNumber | uint32_t | Number of the I/O Output area to get data from |
| ulOffset | uint32_t | Offset inside area to start reading data from |
| ulDataLen | uint32_t | Length of the data being received |
| pvData | void* | Pointer to the returned data buffer |

*Table 67: xChannelIOReadSendData arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### 4.8.25 PLC I/O image functions

**Note:** Do not use xChannelIORead()/xChannelIOWrite() functions while using the PLC functions. If PLC functions are used, the application is responsible to synchronize I/O data access by using the xChannelPLCIsReady...()/xChannelPLCActivate...() functions. There is no internal synchronization mechanism available to synchronize access states between xChannelIORead()/xChannelIOWrite() and the PLC functions. Mixing the functions will result in unpredictable I/O states.

Some of the PLC programs (Programmable Logic Controller also known as SoftPLCs) are using an own process data image layout. Such programs need to copy the process data, from the local buffers, necessary for the standard *xChannelIORead()/xChannelIOWrite() functions,* into their own data image layout. In such a case, process data are always copied two times. First time between the cards I/O process data image and the local function buffers offered by the application and the second time between the local buffers and the PLC specific process images.

PLC functions are design to save the copy between the cards I/O process data image and the local function buffers (done by *xChannelIORead()/xChannelIOWrite()*).

Therefore *xChannelIORead()* and *xChannelIOWrite()* are split into separate functions. One function to get data pointers to the input and output process data image which can be used by the application to directly access the cards I/O process data image. And two other functions to control and synchronize the access to the cards I/O process image data between the user application and the fieldbus protocol, running on the card.

- xChannelPLCMemoryPtr
  Getting the data pointers to the cards I/O process data image
- xChannelPLCActivateRead() / xChannelPLCActivateWrite()
  Activate the data exchange of the cards I/O process data image with the fieldbus protocol running on the card
- xChannelPLCIsReadReady() / xChannelPLCIsWriteReady()
  Check if the fieldbus protocol has finished the access to the crads I/ process data image and if the application is allowed to access the data

Important for the use of the functions is a prior call to the *xChannelPLCMemoryPtr()* function. This will deliver the necessary pointers to the I/O process data image.

**Note:** If the PLC functions are used, the application is responsible to synchronize the data access between the host and the communication channel.

Before closing an application, all memory pointers retrieved by calling xChannelPLCMemoryPtr(), (command **CIFX_MEM_PTR_OPEN**) should be returned to the system to avoid memory leaks. Pointers are returned by calling xChannelPLCMemoryPtr() using the **CIFX_MEM_PTR_CLOSE** command.

#### 4.8.25.1 xChannelPLCMemoryPtr

Retrieve a memory pointer to the I/O data area for a PLC (Programmable Logic Controller). This enables an application to write data directly to the dual port memory (I/O data image) without doing a combined handshake like in *xChannelIORead()* or *xChannelIOWrite().*

Before closing an application, all retrieved pointers should be released to avoid system memory leaks. Releasing a pointer is done by calling xChannelPLCMemoryPtr() using the **CIFX_MEM_PTR_CLOSE** command.

**Function call**

```
int32_t xChannelPLCMemoryPtr(   CIFXHANDLE      hChannel,
                                uint32_t        ulCmd,
                                void*           pvMemoryInfo)
```

**Arguments**

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulCmd | uint32_t | PLC Memory Pointer Commands:<br>Acquire a memory pointer<br>**1 = CIFX_MEM_PTR_OPEN**<br>Map a user specific memory area<br>2 = CIFX_MEM_PTR_USR -> not supported<br>Release a memory pointer<br>**3 = CIFX_MEM_PTR_CLOSE** |
| pvMemoryInfo | void* | Pointer to **PLC_MEMORY_INFORMATION** structure. This structure describes the requested area and also contains the returned memory pointer on success |

*Table 68: xChannelPLCMemoryPtr arguments*

**Return values**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### Description of the PLC_MEMORY_INFORMATION Structure

| Value | Data type | Description |
|---|---|---|
| pvMemoryID | void* | Identifier of the memory area |
| ppvMemoryPtr | void** | Memory pointer |
| ulAreaDefinition | uint32_t | Input / Output area |
| ulAreaNumber | uint32_t | Area number (0..1) |
| pulIOAreaStartOffset | uint32_t* | Buffer to store the I/O area start offset |
| pulAreaSize | uint32_t* | Buffer to store the size of the I/O area |

### PLC_MEMORY_INFORMATION Structure

```
/*****************************************************************************/
/*! PLC Memory Information structure                                         */
/*****************************************************************************/
typedef __CIFx_PACKED_PRE struct PLC_MEMORY_INFORMATIONtag
{
  void*            pvMemoryID;           /*!< Identification of the memory area      */
  void**           ppvMemoryPtr;         /*!< Memory pointer                         */
  uint32_t         ulAreaDefinition;     /*!< Input/output area                      */
  uint32_t         ulAreaNumber;         /*!< Area number                            */
  uint32_t*        pulIOAreaStartOffset; /*!< Start offset                           */
  uint32_t*        pulIOAreaSize;        /*!< Memory size                            */
} __CIFx_PACKED_POST PLC_MEMORY_INFORMATION;
```

### Example

```
//===========================================================================
// Test PLC Functions
//
//
//===========================================================================
void TestPLCFunctions( void)
{
  unsigned char abBuffer[1000]  = {0};
  uint32_t      ulState         = 0;

  printf("\n--- Test PLC functions ---\r\n");

  long lRet = CIFX_NO_ERROR;

  /* Open channel */
  CIFXHANDLE hDevice = NULL;
  lRet = xChannelOpen(NULL, "CIFx0", 0, &hDevice);
  if(lRet != CIFX_NO_ERROR)
  {
    ShowError(lRet);
  } else
  {
    /* Start PLC functions */
    unsigned char*  pabDPMMemory          = NULL;
    uint32_t        ulAreaStartOffset     = 0;
    uint32_t        ulAreaSize            = 0;
    long            lRet                  = CIFX_NO_ERROR;
    long            lRetIN                = CIFX_NO_ERROR;
    long            lRetOUT               = CIFX_NO_ERROR;

    /* Define the memory structures for Input data */
    PLC_MEMORY_INFORMATION tMemory = {0};
    tMemory.pvMemoryID            = NULL;                   // Identification of the memory area
    tMemory.ppvMemoryPtr          = (void**)&pabDPMMemory;  // Memory pointer
    tMemory.ulAreaDefinition      = CIFX_IO_INPUT_AREA;     // Input/output area
    tMemory.ulAreaNumber          = 0;                      // Area number
    tMemory.pulIOAreaStartOffset  = &ulAreaStartOffset;     // Start offset of the requested channel
    tMemory.pulIOAreaSize         = &ulAreaSize;            // Memory size of the requested channel


    /* Define the memory structures for Output data */
    unsigned char*  pabDPMMemory_OUT         = NULL;
    uint32_t        ulAreaStartOffset_OUT    = 0;
```

```
    uint32_t        ulAreaSize_OUT            = 0;

    PLC_MEMORY_INFORMATION tMemory_OUT = {0};
    tMemory_OUT.pvMemoryID            = NULL;                       // Identification of the memory 
area
    tMemory_OUT.ppvMemoryPtr          = (void**)&pabDPMMemory_OUT;  // Memory pointer
    tMemory_OUT.ulAreaDefinition      = CIFX_IO_OUTPUT_AREA;        // Input/output area
    tMemory_OUT.ulAreaNumber          = 0;                          // Area number
    tMemory_OUT.pulIOAreaStartOffset  = &ulAreaStartOffset_OUT;     // Start offset of the requested 
channel
    tMemory_OUT.pulIOAreaSize         = &ulAreaSize_OUT;            // Memory size of the requested 
channel

    /* Open a DPM memory pointer */
    if ( (CIFX_NO_ERROR != (lRetIN  = xChannelPLCMemoryPtr( hDevice, CIFX_MEM_PTR_OPEN, &tMemory)) ) 
||
         (CIFX_NO_ERROR != (lRetOUT = xChannelPLCMemoryPtr( hDevice, CIFX_MEM_PTR_OPEN, 
&tMemory_OUT)))  )
    {
      // Failed to get the memory mapping
      ShowError( lRetIN);
      ShowError( lRetOUT);
    } else
    {
      uint32_t ulWaitBusCount = 100;

      /* Signal application is ready */
      lRet = xChannelHostState( hDevice, CIFX_HOST_STATE_READY, &ulState, 100);
      if( CIFX_NO_ERROR != lRet)
      {
        ShowError(lRet);
      }

      /* Wait until BUS is up and running */
      printf("\r\nWait until BUS communication is available!\r\n");
      do
      {
        lRet = xChannelBusState( hDevice, CIFX_BUS_STATE_ON, &ulState, 100);
        if( CIFX_NO_ERROR != lRet)
        {
          if( CIFX_DEV_NO_COM_FLAG != lRet)
          {
            ShowError(lRet);
            break;
          }
        } else if( 1 == ulState)
        {
          /* Bus is ON */
          printf("\r\nBUS is ON!\r\n");
          break;
        }
      } while ( --ulWaitBusCount > 0);

      if( 0 == ulWaitBusCount)
      {
        ShowError(lRet);
      }

      /*----------------------*/
      /* Start cyclic data IO */
      /*----------------------*/
      if( CIFX_NO_ERROR == lRet)
      {
        printf("\n Press any key to stop \r\n");
        while (!_kbhit())
        {
          // We have a memory mapping, check if access to the DPM is allowed
          uint32_t ulReadState  = 0;
          uint32_t ulWriteState = 0;

          /*---------------------------------------*/
          /* Check if we can access the INPUT image */
          /*---------------------------------------*/

          lRet = xChannelPLCIsReadReady  ( hDevice, 0, &ulReadState);
          if( CIFX_NO_ERROR != lRet)
          {
            ShowError( lRet);
          } else if( 1 == ulReadState)
```

```
          {
            /* It is allowed to read the image */
            /* Read 100 Bytes */
            memcpy( abBuffer, pabDPMMemory, sizeof(abBuffer));

            /* Activate transfer */
            lRet = xChannelPLCActivateRead  ( hDevice, 0);
            if( CIFX_NO_ERROR != lRet)
              ShowError( lRet);
          }

          /*----------------------------------------*/
          /* Check if we can access the OUTPUT image */
          /*----------------------------------------*/
          lRet = xChannelPLCIsWriteReady ( hDevice, 0, &ulWriteState);
          if( CIFX_NO_ERROR != lRet)
          {
            ShowError( lRet);
          } else if( 1 == ulWriteState)
          {
            /* It is allowed to write the image */
            pabDPMMemory_OUT[0]++;
            pabDPMMemory_OUT[1] = abBuffer[1];

            lRet = xChannelPLCActivateWrite  ( hDevice, 0);
            if( CIFX_NO_ERROR != lRet)
              ShowError( lRet);
          }
        }

        /* clean keyboard buffer */
        _getch();
      }
    }

    lRet = xChannelBusState( hDevice, CIFX_BUS_STATE_OFF, &ulState, 100);
    if(CIFX_NO_ERROR != lRet)
    {
      ShowError(lRet);
    }

    lRet = xChannelHostState( hDevice, CIFX_HOST_STATE_NOT_READY, &ulState, 100);
    if(CIFX_NO_ERROR != lRet)
    {
      ShowError(lRet);
    }

    /* Return the DPM memory pointer */
    if ( NULL != pabDPMMemory)
    {
      lRet = xChannelPLCMemoryPtr( hDevice, CIFX_MEM_PTR_CLOSE, &tMemory);
      if(lRet != CIFX_NO_ERROR)
        /* Failed to return memory pointer */
        ShowError( lRet);
    }

    /* Return the DPM memory pointer */
    if ( NULL != pabDPMMemory_OUT)
    {
      lRet = xChannelPLCMemoryPtr( hDevice, CIFX_MEM_PTR_CLOSE, &tMemory_OUT);
      if(lRet != CIFX_NO_ERROR)
        /* Failed to return memory pointer */
        ShowError( lRet);
    }

    // Close channel
    if( hDevice != NULL) xChannelClose(hDevice);
  }

  printf("\n Test PLC functions done\r\n");
}
```

### 4.8.25.2 xChannelPLCActivateRead

Instruct the communication channel to refresh the input process data image. The end of the update cycle must be checked by the application using the function *xChannelPLCIsReadReady()*

**Note:** Do not call this function while the actual state is '*not finished*' (check with the corresponding *xChannelPLCIs......Ready()* function), otherwise the result is unpredictable.

#### Function call

```
int32_t xChannelPLCActivateRead(     CIFXHANDLE      hChannel,
                                     uint32_t        ulAreaNumber)
```

#### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulAreaNumber | uint32_t | Number of the I/O area to request a input data refresh |

*Table 69: xChannelPLCActivateRead arguments*

#### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### 4.8.25.3 xChannelPLCActivateWrite

Instruct the communication channel to refresh the output process data image with the data from the dual port memory. The end of the update cycle must be checked by the user application, using the function *xChannelPLCIsWriteReady()*.

**Note:** Do not call this function while the actual state is '*not finished*' (check with the corresponding *xChannelPLCIs......Ready()* function), otherwise the result is unpredictable.

#### Function call

```
int32_t xChannelPLCActivateWrite(    CIFXHANDLE      hChannel,
                                     uint32_t        ulAreaNumber)
```

#### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulAreaNumber | uint32_t | Number of the I/O area to request a output data refresh |

*Table 70: xChannelPLCActivateWrite arguments*

#### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

#### 4.8.25.4 xChannelPLCIsReadReady

Check if the last read request of the I/O data image is processed and finished by the hardware.

**Function call**

```
int32_t xChannelPLCIsReadReady( CIFXHANDLE      hChannel,
                                uint32_t        ulAreaNumber,
                                uint32_t*       pulReadState)
```

**Arguments**

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulAreaNumber | uint32_t | Number of the I/O area to check for read request completion |
| pulReadState | uint32_t * | Returned state of the handshake operation<br>**0 = pending**<br>**!=0 = finished** |

*Table 71: xChannelPLCIsReadReady arguments*

**Return values**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

#### 4.8.25.5 xChannelPLCIsWriteReady

Check if the last write request handshake is processed and finished by the hardware.

**Function call**

```
int32_t xChannelPLCIsWriteReady(     CIFXHANDLE      hChannel,
                                     uint32_t        ulAreaNumber,
                                     uint32_t*       pulWriteState)
```

**Arguments**

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulAreaNumber | uint32_t | Number of the I/O area to check for read request completion |
| pulWriteState | uint32_t* | Returned state of the handshake operation<br>**0 = pending**<br>**!=0 = finished** |

*Table 72: xChannelPLCIsWriteReady arguments*

**Return values**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.26 DMA functions

### 4.8.26.1 xChannelDMAState

Toggle the *'DMA Enable Flag'* in the communication channel handshake flags. This function can be used to change the I/O image transfer from DPM to bus-master-DMA mode. If PLC memory functions are used, the I/O image pointers need to be re-read after enabling/disabling DMA mode.

**Note:** DMA is only possible on PCI based hardware. On none PCI based hardware, this function is not available and will return with an error

#### Function call

```
int32_t xChannelDMAState(  CIFXHANDLE      hChannel,
                           uint32_t        ulCmd,
                           uint32_t*       pulState)
```

#### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulCmd | uint32_t | DMA State Commands:<br>Disable DMA mode<br>**0 = CIFX_DMA_STATE_OFF**<br>Enable DMA mode<br>**1 = CIFX_DMA_STATE_ON**<br>Get actual DMA state<br>**2 = CIFX_DMA_STATE_GETSTATE** |
| pulState | uint32_t* | Actual state returned |

*Table 73: xChannelDMAState arguments*

#### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

## 4.8.27 Notification functions

Notification functions can be used for devices running in interrupt mode. These functions are registering a callback for pre-defined events from the hardware.

The callback function is called if the corresponding event occurs on the device.

**Note:** Notification functions are only available for devices running in interrupt mode.

### Available notifications

| Notification | Definition | Description |
|---|---|---|
| Packet transfer | CIFX_NOTIFY_RX_MBX_FULL | Receive mailbox full<br>(packet available) |
| | CIFX_NOTIFY_TX_MBX_EMPTY | Send mailbox is empty<br>(packet can be send) |
| I/O data transfer | CIFX_NOTIFY_PD0_IN | Input area 0 has been processed<br>(see below) |
| | CIFX_NOTIFY_PD1_IN | Input area 1 has been processed<br>(see below) |
| | CIFX_NOTIFY_PD0_OUT | Output area 0 has been processed<br>(see below) |
| | CIFX_NOTIFY_PD1_OUT | Output area 1 has been processed<br>(see below) |
| Synchronization | CIFX_NOTIFY_SYNC | Fieldbus synchronous event occurred |
| Communication Flag State | CIFX_NOTIFY_COM_STATE | Communication state of the communication channel has changed |

*Table 74: Notification functions*

### 4.8.27.1 Packet transfer notifications

Packet transfer is used for asynchronous command/confirmation data (e. g. SDOs).

Packet data are handled via a mailbox system. In interrupt mode the actual state of the mailbox system (send/receive mailbox) can be signaled by notifications (see above).

### 4.8.27.2 I/O data transfer notifications

The result of the I/O handling and the corresponding notifications which can be signaled to the user application are depending on the configured I/O data exchange mode. This also effect the handling in the user application, when it is reasonable to call xChannelIORead() and xChannelIOWrite().

**Note:** "**I/O Data Transfer**" notifications depending on the so called "**I/O Exchange Mode**" configured on the device. These modes are defining how notifications are created by the device state changes. The callback functions are called if the driver detects a state change in the device handshake flags. How the application processes the notification is part of the application development and must correspond to the configured mode settings of the device. Handshake modes are described in [1].

Handshake modes are defining which part (device/host) is the active part.

**Following modes are known**

| I/O Exchange mode | Description |
|---|---|
| uncontrolled | No data access synchronization between host and device. Both systems are running independent of each other and data are exchanged without taking care about data consistency.<br>Notification: NONE |
| buffered<br>host controlled<br>**(default)** | The host activates the data transfer between device and host.<br>Actual I/O data from the fieldbus system and from the host are always stored in the local I/O buffers on the device.<br>If the host requests new input data (calling xChannelIORead()), the device will copy the currently available input data from the local buffer to the DPM (PDx_IN area) and signals "data updated" (1). The host can read the new input data with the next call to xChannelIORead().<br>If the host writes new output data to the DPM (PDx_OUT area) by calling xChannelIOWrite(). The device, will copy the data from the DPM to the local output buffer and signals "data updated" (2) if the copy is done.<br>(1) Notification for the input data: CIFX_NOTIFY_PDx_IN<br>(2) Notification for the output data: CIFX_NOTIFY_PDx_OUT |
| buffered<br>device controlled | **ATTENTION:**<br>**By default NOT supported from Hilscher Stacks**<br>The device will start to copy the actual input data from the fieldbus system to the DPM (PDx_IN area) and signals "input data updated" (1). The user has to call xChannelIORead() to read the input data from the DPM. All further input data received by the fieldbus are stored in the device local input buffer until the host reads the data again.<br>The device requests new output data from the host (2) and until the host has written new data, output data are send from the local device buffer to the fieldbus system. If the host writes new output data (calling xChannelIOWrite()) , the device copies the data to the local output buffer and requests (2) new output data as soon as the copy of the data is done.<br>(1) Notification for the input data: CIFX_NOTIFY_PDx_IN<br>(2) Notification for the output data: CIFX_NOTIFY_PDx_OUT |

*Table 75: I/O Exchange modes*

### Data Exchange Mode - Buffered Host Controlled I/O

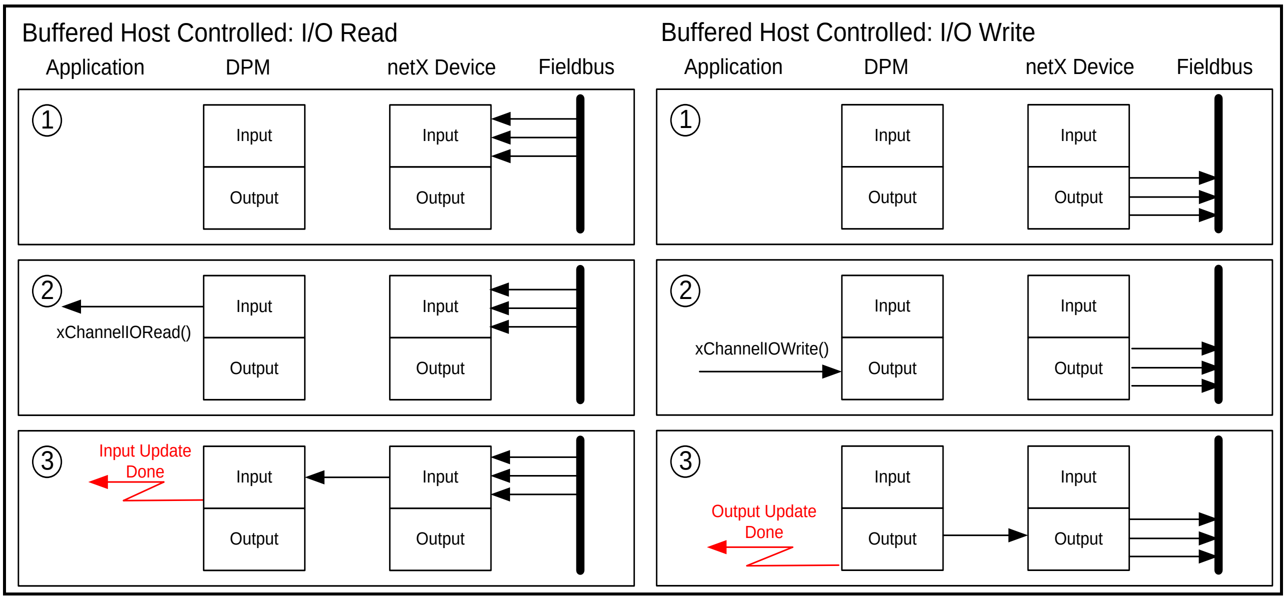



*Figure 4: Data Exchange Mode: Buffered Host Controlled I/O*

| **I/O Read** | | | **I/O Write** | |
|---|---|---|---|---|
| **Step** | **Description** | | **Step** | **Description** |
| 1 | Fieldbus protocol reads input data from the fieldbus system and stores the data in the internal "**Input Buffer**" | | 1 | Fieldbus protocol sends the output data stored in the internal "**Output Buffer**" to the fieldbus system |
| 2 | Application uses xChannelIORead() which reads the actual data from the DPM (Dual-Ported Memory) PD-IN area and signals the card to update the PD-IN area. | | 2 | Application calls xChannelIOWrite() which write the actual user data to the DPM (Dual-Ported Memory) PD-OUT area and signals the card to update the internal "**Output Buffer**". |
| 3 | The stack copies the actual data form the internal "**Input Buffer**" (holding the latest input data) to the DPM PD-IN area.<br>After the data copy, the protocol stack signals "**Input Update Done**" which schedules a CIFX_NOTIFY_PDx_IN notification. | | 3 | The stack copies the data from the DPM PB-OUT area to the internal "**Output Buffer**".<br>After the data copy, the protocol stack signals "**Output Update Done**" which schedules a CIFX_NOTIFY_PDx_OUT notification. |

*Table 76: Data Exchange Mode: Buffered Host Controlled I/O (steps)*

**Note:** In these modes, the notifications just inform the application when the input data are copied from the device local input buffer to the DPM and when the output data are copied from the DPM to the device local output buffer. There is no synchronization with any fieldbus data cycle.

## Data Exchange Mode - Buffered Device Controlled I/O

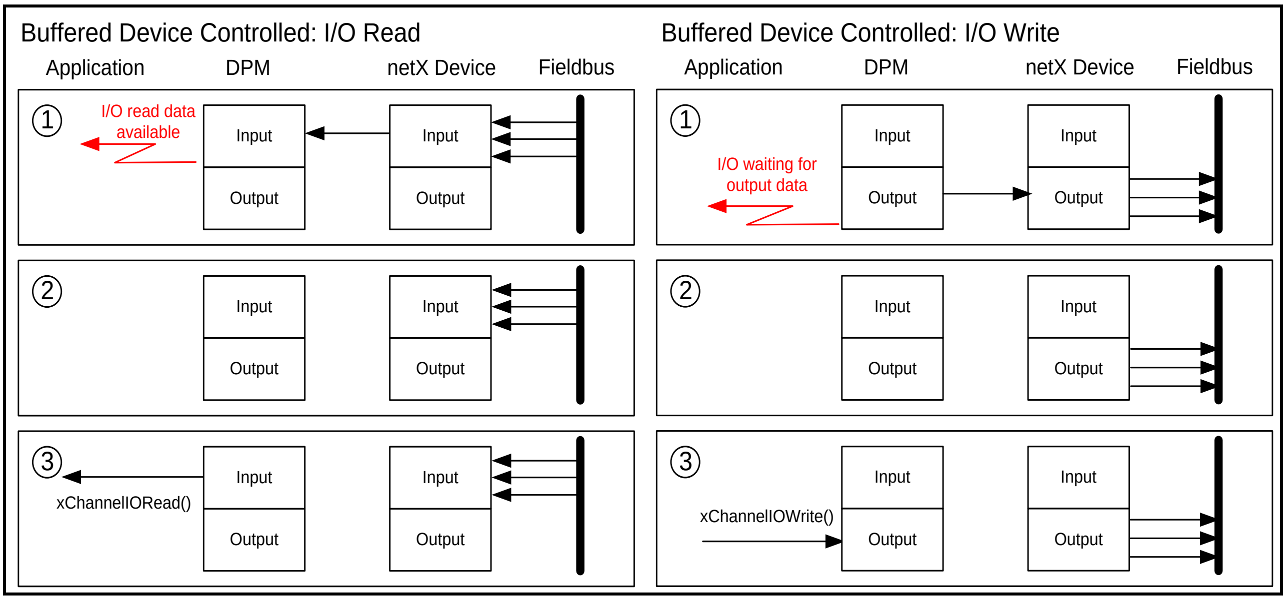



*Figure 5: Data Exchange Mode: Buffered Device Controlled I/O*

| **I/O Read** | | | **I/O Write** | |
|---|---|---|---|---|
| **Step** | **Description** | | **Step** | **Description** |
| 1 | Fieldbus protocol reads input data from the fieldbus system and stores the data in the internal "**Input Buffer**", copies the data to the DPM (PDx_IN) area.<br>After writing the input data to the DPM (PDx_IN), the device signals **"I/O read data available**" scheduling a CIFX_NOTIFY_PDx_IN notification. | | 1 | The device takes the output date from the DPM (PDx_OUT area) and copies it to the device local "**Output Buffer**".<br>After the copy the device requests new output data by signaling a CIFX_NOTIFY_PDx_OUT notification. |
| 2 | Any further data from a bus cycle will be stored in the device local input buffer. | | 2 | The device sends the stored output with any further bus cycle |
| 3 | If the host reads the input data (by calling xChannelIORead()), the device is signaled "**Input data done**" than the device is able to update the input data again. | | 3 | If the host writes new output data to the DPM (PDx_OUT) by calling xChannelIOWrite(). The device is signaled "**New output data available**". |

*Table 77: Data Exchange Mode: Buffered Device Controlled I/O (steps)*

**Note:** The application determines when read input or write output data. The notification informs the application when read or write is possible. There is no synchronization with any fieldbus data cycle.

### Determining the Configured "I/O Exchange Mode"

The configured I/O data exchange (host controlled/device controlled) can be read from the communication channels "Common Status Block" (bPDInHskMode / (bPDOutHskMode). The block can be read and evaluated by the user application using the *xChannelCommonStatusBlock()* function.

The "Common Status Block" is described in the "netX Dual-Port Memory Interface DPM Manual".

Following data exchanges mode definitions are available:

```
/* Block definition: I/O Mode */
#define RCX_IO_MODE_DEFAULT        0x0000    /*!< I/O mode default, for compability reasons this
value is identical to 0x4 (buffered host controlled) */
#define RCX_IO_MODE_BUFF_DEV_CTRL  0x0002    /*!< I/O mode buffered device controlled */
#define RCX_IO_MODE_UNCONTROLLED   0x0003    /*!< I/O mode bus synchronous device controlled */
#define RCX_IO_MODE_BUFF_HST_CTRL  0x0004    /*!< I/O mode buffered host controlled */
```

**Note:** Possible data exchanges modes are fieldbus protocol specific and described in the corresponding fieldbus "Protocol API" manual.

### 4.8.27.3 Bus synchronization notifications

The notification functions offering a bus synchronization event if supported by the fieldbus protocol.

**Note:** "**Synchronization**" notifications" depending on the so called "**Synchronization Mode**" configured on the device.

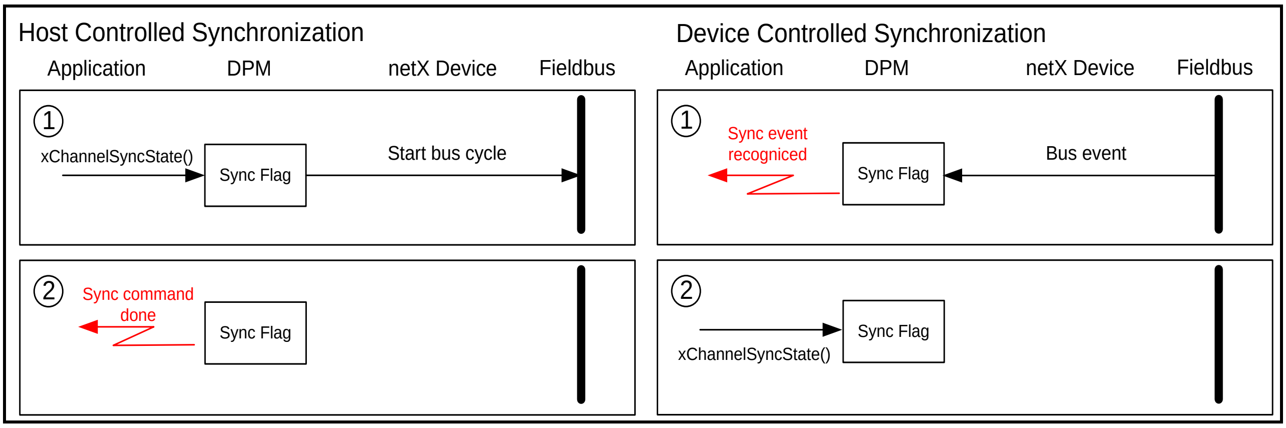



*Figure 6: Bus synchronization notifications*

| Host Controlled Synchronization | | | Device Controlled Synchronization | |
|---|---|---|---|---|
| **Step** | **Description** | | **Step** | **Description** |
| 1 | The application starts to send a synchronization command xChannelSyncState(**CIFX_SYNC_SIGNAL_CMD**) to the device.<br>Depending on the configuration the synchronization command (e.g. start bus cycle) is executed by the device. | | 1 | If the fieldbus protocol recognizes the configured sync event, is signals a "Sync event recognized" and scheduled a CIFX_NOTIFY_SYNC notification. |
| 2 | The device signals "Sync command done", by scheduling a CIFX_NOTIFY_SYNC notification if the command is processed. | | 2 | Application has to call xChannelSyncState(**CIFX_SYNC_SIGNAL_ACK**) before a new bus event is signaled again. |

*Table 78: Bus synchronization notifications (steps)*

#### Determining the Configured "Synchronization Mode"

The configured synchronization mode (host controlled/device controlled) can be read from the communication channels "Common Status Block" (bSyncHskMode). The block can be read and evaluated by the user application using the *xChannelCommonStatusBlock()* function.

The "Common Status Block" is described in the "netX Dual-Port Memory Interface DPM Manual".

Following synchronization mode definitions are available:

```
/* Block definition: Synchronization Mode */
#define RCX_SYNC_MODE_OFF                                    0x00
#define RCX_SYNC_MODE_DEV_CTRL                               0x01
#define RCX_SYNC_MODE_HST_CTRL                               0x02
```

**Note:** Possible synchronization modes are fieldbus protocol specific and described in the corresponding fieldbus "Protocol API" manual

### 4.8.27.4 PFN_NOTIFY_CALLBACK - Callback function definition

**Note:** The registered callback function will be invoked as soon as the callback is registered and the corresponding event is valid. This could also happen while the user application is still in the *xChannelRegisterNotification()* function call.

#### Function call

```
void NotificationCallback( uint32_t   ulNotification,
                           uint32_t   ulDataLen,
                           void*      pvData,
                           void*      pvUser);
```

#### Arguments

| Argument | Data type | Description |
|---|---|---|
| ulNotification | uint32_t | Occurred event |
| ulDataLen | uint32_t | Length of additional data |
| pvData | void* | Additional Data (depends on ulNotification) |
| pvUser | void* | User parameter from registration |

*Table 79: PFN_NOTIFY_CALLBACK arguments*

#### Possible notification events

| ulNotification | Passed Data | Description |
|---|---|---|
| CIFX_NOTIFY_RX_MBX_FULL | Pointer to CIFX_NOTIFY_RX_MBX_FULL_DATA_T structure containing the total number of packets waiting to be read from the device. | Signaled when receive mailbox becomes full and a data packet is available to read. |
| CIFX_NOTIFY_TX_MBX_EMPTY | Pointer to CIFX_NOTIFY_TX_MBX_EMPTY_DATA_T structure containing the maximum amount of packets which can be send to the device. | Send mailbox becomes empty and a new packet can be send to the device. |
| CIFX_NOTIFY_PD0_IN | none | Input area 0 has been processed |
| CIFX_NOTIFY_PD1_IN | none | Input area 1 has been processed |
| CIFX_NOTIFY_PD0_OUT | none | Output area 0 has been processed |
| CIFX_NOTIFY_PD1_OUT | none | Output area 1 has been processed |
| CIFX_NOTIFY_SYNC | none | Bus synchronization notification, signals the SYNC event on the fieldbus/device occurred. |
| CIFX_NOTIFY_COM_STATE | Pointer to CIFX_NOTIFY_COM_STATE_T structure containing the actual state of the COM-flag | Communication flag notification. Signals state changes of the COM-flag (set or cleared). |

*Table 80: Notification events*

### 4.8.27.5 xChannelRegisterNotification

Register an event callback for channel events.

Depending on the event type additional information is passed in the callback. If a callback is already registered for the given event, the function will return an error.

It is not possible to register multiple applications for the same notification.

**Note:** The registered callback function will be invoked as soon as the callback is registered and the corresponding event is valid. This could also happen while the user application is still in the *xChannelRegisterNotification()* function call.

#### Function call

```
int32_t xChannelRegisterNotification(     CIFXHANDLE          hChannel,
                                          uint32_t            ulNotification,
                                          PFN_NOTIFY_CALLBACK pfnCallback,
                                          void*               pvUser);
```

#### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulNotification | uint32_t | Possible Notification:<br>**1 = CIFX_NOTIFY_RX_MBX_FULL**<br>**2 = CIFX_NOTIFY_TX_MBX_EMPTY**<br>**3 = CIFX_NOTIFY_PD0_IN**<br>**4 = CIFX_NOTIFY_PD1_IN**<br>**5 = CIFX_NOTIFY_PD0_OUT**<br>**6 = CIFX_NOTIFY_PD1_OUT**<br>**7 = CIFX_NOTIFY_SYNC**<br>**8 = CIFX_NOTIFY_COM_STATE** |
| pfnCallback | PFN_NOTIFY_CALLBACK | Function to be called if event occurs |
| pvUser | void* | Parameter passed to callback |

*Table 81: xChannelRegisterNotification arguments*

#### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### 4.8.27.6 xChannelUnregisterNotification

Un-registers a previously registered notification event callback function for channel events.

#### Function call

```
int32_t xChannelUnregisterNotification(   CIFXHANDLE      hChannel,
                                          uint32_t        ulNotification);
```

#### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulNotification | uint32_t | Possible Notification:<br>**1 = CIFX_NOTIFY_RX_MBX_FULL**<br>**2 = CIFX_NOTIFY_TX_MBX_EMPTY**<br>**3 = CIFX_NOTIFY_PD0_IN**<br>**4 = CIFX_NOTIFY_PD1_IN**<br>**5 = CIFX_NOTIFY_PD0_OUT**<br>**6 = CIFX_NOTIFY_PD1_OUT**<br>**7 = CIFX_NOTIFY_SYNC**<br>**8 = CIFX_NOTIFY_COM_STATE** |

*Table 82: xChannelUnregisterNotification arguments*

#### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

### 4.8.28 Fieldbus synchronization handling

Certain fieldbus protocol stacks are offering so call synchronization functionalities to synchronize devices connected to a fieldbus system.

Such synchronization functions are handled independent from of the cyclic I/O data transfer.

**General Definition:**

In general, synchronization handling distinguishes between device synchronization and host synchronization operation. The difference between the two modes is the component (host / device) which activates the synchronization and the response to the synchronization signal (event).

- Host Controlled Synchronization
  In this mode, the host signals a synchronization to the hardware and the hardware has to respond to this signal
- Device Controlled Synchronization
  In this case, the device starts to signal a synchronization event and the host has to acknowledge the reception of the synch signal.

Synchronization must be handled by the user application and can be done in polling mode (not preferred) and interrupt mode of the hardware. In interrupt mode the drivers notification function is used to handle synchronization event via a user callback function.

**Synchronization Handling in Polling Mode**

In polling mode the xChannelSyncState() function is used to activate (**CIFX_SYNC_SIGNAL_CMD**) or to acknowledge (**CIFX_SYNC_ACKNOWLEDGE_CMD**) a synchronization signal, depending on the configured fieldbus synchronization mode (host controlled / device controlled).

xChannelSyncState() can also be used to check (ulTimeout == 0) or to wait (ulTimeout != 0) for a device synchronization signal. Or until a new host synchronization command can be initiated.

**Synchronization Handling in Interrupt Mode**

In interrupt mode, the drivers register notification function is used to handle synchronization events. A user application is able to register a callback function for synchronization events (CIFX_SYNC_EVENT). The registered callback function will be executed if either the device is signaling a synchronization event or if the device acknowledges a synchronization command initiated by the host application.

- Device Synchronization Mode
  The host has to register for a synchronization event and if the event occurs (callback function is invoked) the host has to acknowledge the event using the
  *xChannelSyncState(...***CIFX_SYNC_ACKNOWLEDGE_CMD...***).*
- Host Synchronization Mode
  The host calls *xChannelSyncState(...***CIFX_SYNC_SIGNAL_CMD**...*)* to signal a synchronization. The registered callback function will be invoked if the device acknowledges the command.

**Verifying Synchronization Misses:**

*xChannelSyncState()* offers a pointer to an error counter buffer (pulErrorCount). This counter can be used by the user application to determine the lost of a synchronization signals.

A changing error counter value between two subsequent *xChannelSyncState()* calls indicates a lost signal. This means, in "Host Controlled Mode", the device was not quick enough to process the previous command and in "Device Controlled Mode", the host has not acknowledged the synchronization signal until the next synchronization signal was initiated.

**Determining the Configured Synchronization Mode**

The configured synchronization mode (host controlled / device controlled) can be read from the communication channels "Common Status Block" (bSyncHskMode). The block can be read and evaluated by the user application using the *xChannelCommonStatusBlock()* function.

The "Common Status Block" is described in the "netX Dual-Port Memory Interface DPM Manual".

Currently the following synchronization modes are defined.

```
/* Block definition: Synchronization Mode */
#define RCX_SYNC_MODE_OFF                                   0x00
#define RCX_SYNC_MODE_DEV_CTRL                              0x01
#define RCX_SYNC_MODE_HST_CTRL                              0x02
```

Also the synchronization error counter (bErrorSyncCnt) and the synchronization source (bSyncSource) can be evaluated from the "Common Status Block".

**Note:** Fieldbus synchronization must be supported by the used fieldbus protocol stack. Please consult the corresponding fieldbus "Protocol API" manual to make sure synchronization is supported.

**Note:** Synchronization operation assumes a corresponding fieldbus configuration.

**Note:** Fieldbus synchronization is a time critical process and should be processed as fast as possible. On Windows operating systems, responds times to synchronization events are not guaranteed and can lead in serious jitter. Usually synchronization will be handled in interrupt mode.

The *xChannelSyncState()* function can also be used in polling mode using a timeout and the CIFX_SYNC_WAIT_CMD command, but this will not change the Windows operating system respond timing issues.

### Function call

```
int32_t xChannelSyncState(CIFXHANDLE      hChannel,
                          uint32_t        ulCmd,
                          uint32_t        ulTimeout
                          uint32_t*       pulErrorCount)
```

### Arguments

| Argument | Data type | Description |
|---|---|---|
| hChannel | CIFXHANDLE | Handle of the channel. |
| ulCmd | uint32_t | Synchronization Commands:<br>Signal sync to device<br>**1 = CIFX_SYNC_SIGNAL_CMD**<br>Acknowledge a sync that has been set by the device<br>**2 = CIFX_SYNC_ACKNOWLEDGE_CMD**<br>Wait for sync being signaled by device (Device Controlled), or until host can signal new Sync State (Host Controlled)<br>**3 = CIFX_SYNC_WAIT_CMD** |
| ulTimeout | uint32_t | Timeout in ms to wait until bits can be signaled or have been signaled by the device |
| pulErrorCount | uint32_t* | Returned Actual Sync Error counter |

*Table 83: xChannelSyncState arguments*

### Return values

CIFX_NO_ERROR if the function succeeds.

If the function fails, a non-zero error code is returned. For a list, see chapter *Error codes* on page 110. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

# 5 Simple C-application example

The simple C application demonstrates the minimum functions which must be called to enable an application to work with a CIFX/COMX/netX based hardware.

The example is named CIFXDEMO and the source, including a Microsoft Visual C++ 6.0 project, can be found on the Hilscher system CDs.

## 5.1 The Main() function

```
/*****************************************************************************/
/*! The main function
*   \return 0 on success                                                     */
/*****************************************************************************/
int main(int argc, char* argv[])
{
  CIFXHANDLE hDriver = NULL;
  int32_t    lRet    = CIFX_NO_ERROR;

  UNREFERENCED_PARAMETER(argc);
  UNREFERENCED_PARAMETER(argv);

  /* Open the cifX driver */
  lRet = xDriverOpen(&hDriver);

  if(CIFX_NO_ERROR != lRet)
  {
    printf("Error opening driver. lRet=0x%08X\r\n", lRet);
  } else
  {
    /* Example how to find a cifX/comX board */
    EnumBoardDemo(hDriver);

    /* Example how to communicate with the SYSTEM device of a board */
    SysdeviceDemo(hDriver, "cifX0");

    /* Example how to communicate with a communication channel on a board */
    ChannelDemo(hDriver, "cifX0", 0);

    /* Close the cifX driver */
    xDriverClose(hDriver);
  }
  return 0;
}
```

## 5.2 System device example

```
/*****************************************************************************/
*! Function to demonstrate system device functionality (Packet Transfer)
*   \return CIFX_NO_ERROR on success                                        */
/*****************************************************************************/
int32_t SysdeviceDemo(CIFXHANDLE hDriver, char* szBoard)
{
  int32_t    lRet = CIFX_NO_ERROR;
  CIFXHANDLE hSys = NULL;

  printf("---------- System Device handling demo ----------\r\n");

  /* Driver/Toolkit successfully opened */
  lRet = xSysdeviceOpen(hDriver, szBoard, &hSys);

  if(CIFX_NO_ERROR != lRet)
  {
    printf("Error opening SystemDevice!\r\n");
  } else
  {
    SYSTEM_CHANNEL_SYSTEM_INFO_BLOCK tSysInfo       = {0};
    uint32_t                         ulSendPktCount = 0;
    uint32_t                         ulRecvPktCount = 0;
    CIFX_PACKET                      tSendPkt       = {0};
    CIFX_PACKET                      tRecvPkt       = {0};

    /* System channel successfully opened, try to read the System Info Block */
    if( CIFX_NO_ERROR != (lRet = xSysdeviceInfo(hSys,
                                                 CIFX_INFO_CMD_SYSTEM_INFO_BLOCK,
                                                 sizeof(tSysInfo),
                                                 &tSysInfo)))
    {
      printf("Error querying system information block\r\n");
    } else
    {
      printf("System Channel Info Block:\r\n");
      printf("DPM Size         : %u\r\n", tSysInfo.ulDpmTotalSize);
      printf("Device Number    : %u\r\n", tSysInfo.ulDeviceNumber);
      printf("Serial Number    : %u\r\n", tSysInfo.ulSerialNumber);
      printf("Manufacturer     : %u\r\n", tSysInfo.usManufacturer);
      printf("Production Date  : %u\r\n", tSysInfo.usProductionDate);
      printf("Device Class     : %u\r\n", tSysInfo.usDeviceClass);
      printf("HW Revision      : %u\r\n", tSysInfo.bHwRevision);
      printf("HW Compatibility : %u\r\n", tSysInfo.bHwCompatibility);
    }
```

```
    /* Do a simple Packet exchange via system channel */
    xSysdeviceGetMBXState(hSys, &ulRecvPktCount, &ulSendPktCount);
    printf("System Mailbox State: MaxSend = %u, Pending Receive = %u\r\n",
           ulSendPktCount, ulRecvPktCount);

    if(CIFX_NO_ERROR != (lRet = xSysdevicePutPacket(hSys,
                                                    &tSendPkt,
                                                    PACKET_WAIT_TIMEOUT)))
    {
      printf("Error sending packet to device!\r\n");
    } else
    {
      printf("Send Packet:\r\n");
      DumpPacket(&tSendPkt);

      xSysdeviceGetMBXState(hSys, &ulRecvPktCount, &ulSendPktCount);
      printf("System Mailbox State: MaxSend = %u, Pending Receive = %u\r\n",
              ulSendPktCount, ulRecvPktCount);

      if(CIFX_NO_ERROR != (lRet = xSysdeviceGetPacket(hSys,
                                                      sizeof(tRecvPkt),
                                                      &tRecvPkt,
                                                      PACKET_WAIT_TIMEOUT)) )
      {
        printf("Error getting packet from device!\r\n");
      } else
      {
        printf("Received Packet:\r\n");
        DumpPacket(&tRecvPkt);

        xSysdeviceGetMBXState(hSys, &ulRecvPktCount, &ulSendPktCount);
        printf("System Mailbox State: MaxSend = %u, Pending Receive = %u\r\n",
                ulSendPktCount, ulRecvPktCount);
      }
    }

    /* Close the system device */
    xSysdeviceClose(hSys);
  }

  printf(" State = 0x%08X\r\n", lRet);
  printf("----------------------------------------------------\r\n");

  return lRet;
}
```

## 5.3 Communication channel example

```
/*****************************************************************************/
/*! Function to demonstrate communication channel functionality
*   Packet Transfer and I/O Data exchange
*   \return CIFX_NO_ERROR on success                                        */
/*****************************************************************************/
int32_t ChannelDemo(CIFXHANDLE hDriver, char* szBoard, uint32_t ulChannel)
{
  CIFXHANDLE hChannel = NULL;
  int32_t    lRet     = CIFX_NO_ERROR;

  printf("---------- Communication Channel demo ----------\r\n");

  lRet = xChannelOpen(hDriver, szBoard, ulChannel, &hChannel);

  if(CIFX_NO_ERROR != lRet)
  {
    printf("Error opening Channel!");
  } else
  {
    CHANNEL_INFORMATION tChannelInfo  = {0};
    CIFX_PACKET         tSendPkt      = {0};
    CIFX_PACKET         tRecvPkt      = {0};
    /* Read and write I/O data (32Bytes). Output data will be incremented each
       cyle */
    uint8_t              abSendData[32] = {0};
    uint8_t              abRecvData[32] = {0};
    uint32_t             ulCycle        = 0;
    uint32_t             ulState        = 0;

    /* Channel successfully opened, so query basic information */
    if( CIFX_NO_ERROR != (lRet = xChannelInfo(hChannel,
                                              sizeof(CHANNEL_INFORMATION),
                                              &tChannelInfo)))
    {
      printf("Error querying system information block\r\n");
    } else
    {
      printf("Communication Channel Info:\r\n");
      printf("Device Number    : %u\r\n", tChannelInfo.ulDeviceNumber);
      printf("Serial Number    : %u\r\n", tChannelInfo.ulSerialNumber);
      printf("Firmware         : %s\r\n", tChannelInfo.abFWName);
      printf("FW Version       : %u.%u.%u build %u\r\n",
              tChannelInfo.usFWMajor,
              tChannelInfo.usFWMinor,
              tChannelInfo.usFWRevision,
              tChannelInfo.usFWBuild);
      printf("FW Date          : %02u/%02u/%04u\r\n",
              tChannelInfo.bFWMonth,
              tChannelInfo.bFWDay,
              tChannelInfo.usFWYear);
      printf("Mailbox Size     : %u\r\n", tChannelInfo.ulMailboxSize);
    }
```

```
    /* Do a basic Packet Transfer */
    if(CIFX_NO_ERROR != (lRet = xChannelPutPacket( hChannel,
                                                   &tSendPkt,
                                                   PACKET_WAIT_TIMEOUT)))
    {
      printf("Error sending packet to device!\r\n");
    } else
    {
      printf("Send Packet:\r\n");
      DumpPacket(&tSendPkt);

      if(CIFX_NO_ERROR != (lRet = xChannelGetPacket(hChannel,
                                                    sizeof(tRecvPkt),
                                                    &tRecvPkt,
                                                    PACKET_WAIT_TIMEOUT)) )
      {
        printf("Error getting packet from device!\r\n");
      } else
      {
        printf("Received Packet:\r\n");
        DumpPacket(&tRecvPkt);
      }
    }
```

```
    /* Do a basic IO data transfer */
    /* Set Host Ready to signal the filed bus an application is ready */
    lRet = xChannelHostState(hChannel,
                             CIFX_HOST_STATE_READY,
                             &ulState,
                             HOSTSTATE_TIMEOUT);

    if(CIFX_NO_ERROR != lRet)
    {
      printf("Error setting host ready!\r\n");
    } else
    {
      /* Switch on the bus if it is not automatically running (see configuration
         options) */
      lRet = xChannelBusState( hChannel, CIFX_BUS_STATE_ON, &ulState, 0L);
      if(CIFX_NO_ERROR != lRet)
      {
        printf("Unable to start the filed bus!\r\n");
      } else
      {

        /* Do I/O Data exchange until a key is hit */
        while(!kbhit())
        {
          if(CIFX_NO_ERROR != (lRet = xChannelIORead(hChannel,
                                                     0, 0, sizeof(abRecvData),
                                                     abRecvData,
                                                     IO_WAIT_TIMEOUT)))
          {
            printf("Error reading IO Data area!\r\n");
            break;
          } else
          {
            printf("IORead Data:");
            DumpData(abRecvData, sizeof(abRecvData));

            if(CIFX_NO_ERROR != (lRet = xChannelIOWrite(hChannel,
                                                        0, 0, sizeof(abRecvData),
                                                        abRecvData,
                                                        IO_WAIT_TIMEOUT)))
            {
              printf("Error writing to IO Data area!\r\n");
              break;
            } else
            {
              printf("IOWrite Data:");
              DumpData(abSendData, sizeof(abSendData));

              /* Create new output data */
              memset(abSendData, ulCycle + 1, sizeof(abSendData));
            }
          }
        }
      }
    }
```

```
    /* Switch off the bus */
    xChannelBusState( hChannel, CIFX_BUS_STATE_OFF, &ulState, 0L);

    /* Set Host not ready to stop bus communication */
    xChannelHostState(hChannel, CIFX_HOST_STATE_NOT_READY,
                                &ulState,
                                HOSTSTATE_TIMEOUT);

    /* Close the communication channel */
    xChannelClose(hChannel);
  }

  if(CIFX_NO_ERROR != lRet)
  {
    char szBuffer[256] = {0};
    xDriverGetErrorDescription(lRet, szBuffer, sizeof(szBuffer));
    printf(" State = 0x%08X <%s>\r\n", lRet, szBuffer);
  } else
  {
    printf(" State = 0x%08X\r\n", lRet);
  }
  printf("----------------------------------------------------\r\n");

  return lRet;
}
```

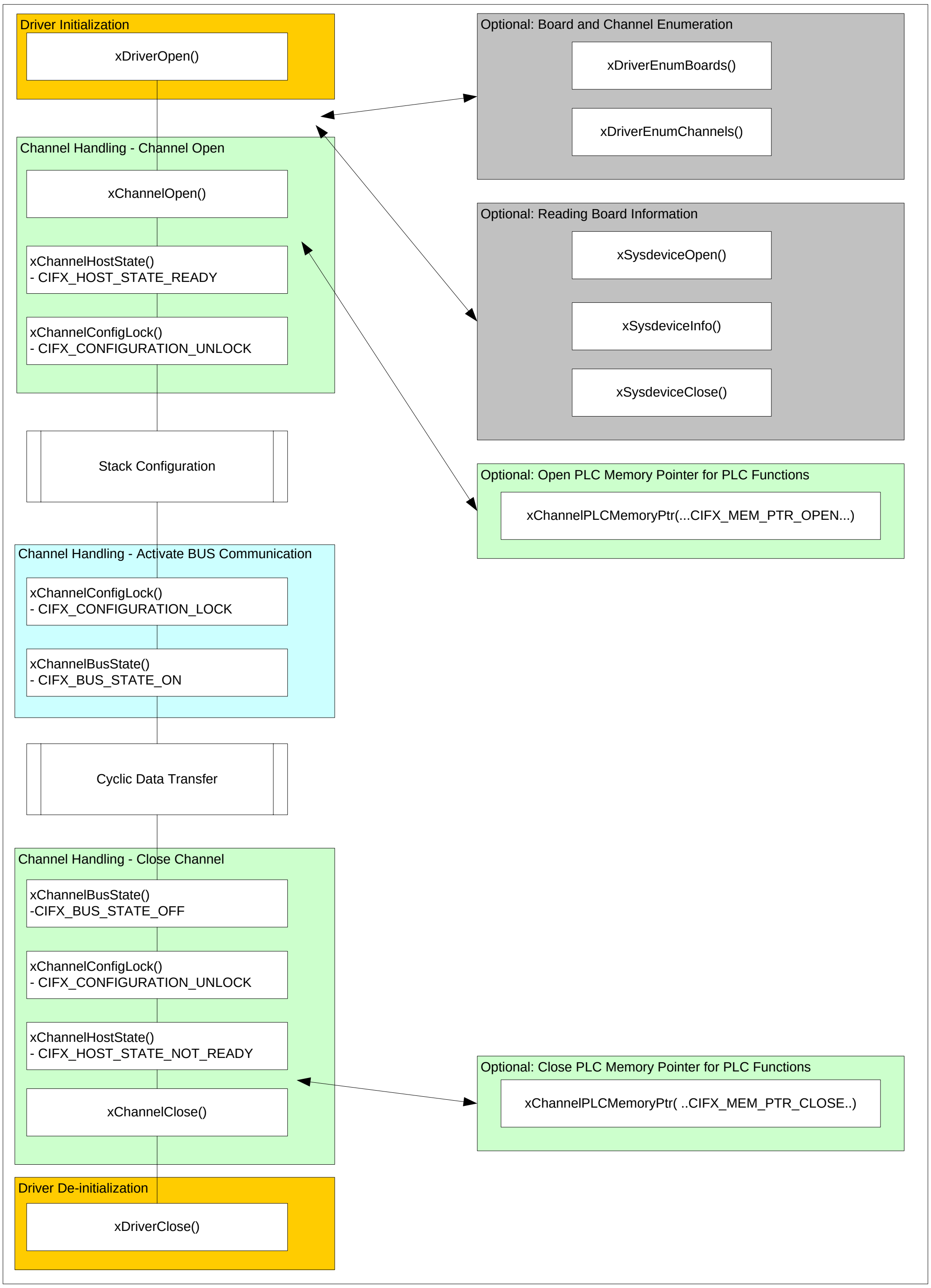
## 5.4 Board and channel enumeration

```
/*****************************************************************************/
/*! Function to demonstrate the board/channel enumeration
*   \return CIFX_NO_ERROR on success                                        */
/*****************************************************************************/
void EnumBoardDemo(CIFXHANDLE hDriver)
{
  uint32_t          ulBoard    = 0;
  BOARD_INFORMATION tBoardInfo = {0};

  printf("---------- Board/Channel enumeration demo ----------\r\n");

  /* Iterate over all boards */
  while(CIFX_NO_ERROR == xDriverEnumBoards(hDriver, ulBoard, sizeof(tBoardInfo),
                                           &tBoardInfo))
  {
    uint32_t            ulChannel    = 0;
    CHANNEL_INFORMATION tChannelInfo = {0};

    printf("Found Board %.10s\r\n", tBoardInfo.abBoardName);
    if(strlen( (char*)tBoardInfo.abBoardAlias) != 0)
      printf(" Alias        : %.10s\r\n", tBoardInfo.abBoardAlias);
    printf(" DeviceNumber : %u\r\n", tBoardInfo.tSystemInfo.ulDeviceNumber);
    printf(" SerialNumber : %u\r\n", tBoardInfo.tSystemInfo.ulSerialNumber);
    printf(" Board ID     : %u\r\n", tBoardInfo.ulBoardID);
    printf(" System Error : 0x%08X\r\n", tBoardInfo.ulSystemError);
    printf(" Channels     : %u\r\n", tBoardInfo.ulChannelCnt);
    printf(" DPM Size     : %u\r\n", tBoardInfo.ulDpmTotalSize);

    /* iterate over all channels on the current board */
    while(CIFX_NO_ERROR == xDriverEnumChannels(hDriver, ulBoard, ulChannel,
                                               sizeof(tChannelInfo), &tChannelInfo))
    {
      printf(" - Channel %u:\r\n", ulChannel);
      printf("    Firmware : %s\r\n", tChannelInfo.abFWName);
      printf("    Version  : %u.%u.%u build %u\r\n",
             tChannelInfo.usFWMajor,
             tChannelInfo.usFWMinor,
             tChannelInfo.usFWBuild,
             tChannelInfo.usFWRevision);
      printf("    Date     : %02u/%02u/%04u\r\n",
             tChannelInfo.bFWMonth,
             tChannelInfo.bFWDay,
             tChannelInfo.usFWYear);
      ++ulChannel;
    }
    ++ulBoard;
  }
  printf("----------------------------------------------------\r\n");
}
```

# 6 General protocol stack handling

This chapter describes the general usage of the *CIFX API* in conjunction with a fieldbus protocol stack.

## 6.1 Overview



**Driver Initialization**

- xDriverOpen() **Open the Driver**

**Reading Driver Information (Optional)**

- xDriverEnumBoards() **Enumerate all available Boards**
- xDriverEnumChannels() **Enumerate channels on a given board**

**Reading Board Information (Optional)**

- xSysdeviceOpen() **Open the system device of a board**
- xSysdeviceInfo() **Read board information board via system channel**
- xSysdeviceClose() **Close the system channel**

**Channel Handling - Open Channel**

- xChannelOpen() **Open a communication channel**
- **Optional:** Read the channel I/O memory pointers if the PLC functions *xChannelPLC*... are used for I/O data transfer

  xChannelPLCMemoryPtr(...CIFX_PLC_MEM_PTR_OPEN...)
- xChannelHostState(...CIFX_HOST_STATE_READY...) **Signal Application is online**

  Wait until channel is READY if the timeout <> 0

  Standard Timeout = 1000 ms
- xChannelConfigLock(CIFX_CONFIGURATION_UNLOCK) **Unlock the configuration**

**==> Stack Configuration**

**Channel Handling - Activate BUS Communication**

- xChannelConfigLock(...CIFX_CONFIGURATION_LOCK...) **Locking of the configuration**
- xChannelBusState(...CIFX_BUS_STATE_ON...) **Switch BUS to ON**

  Timeout <> 0, waits until BUS is ON

  Standard Timeout: 5000 ms

**==> Cyclic Data Transfer**

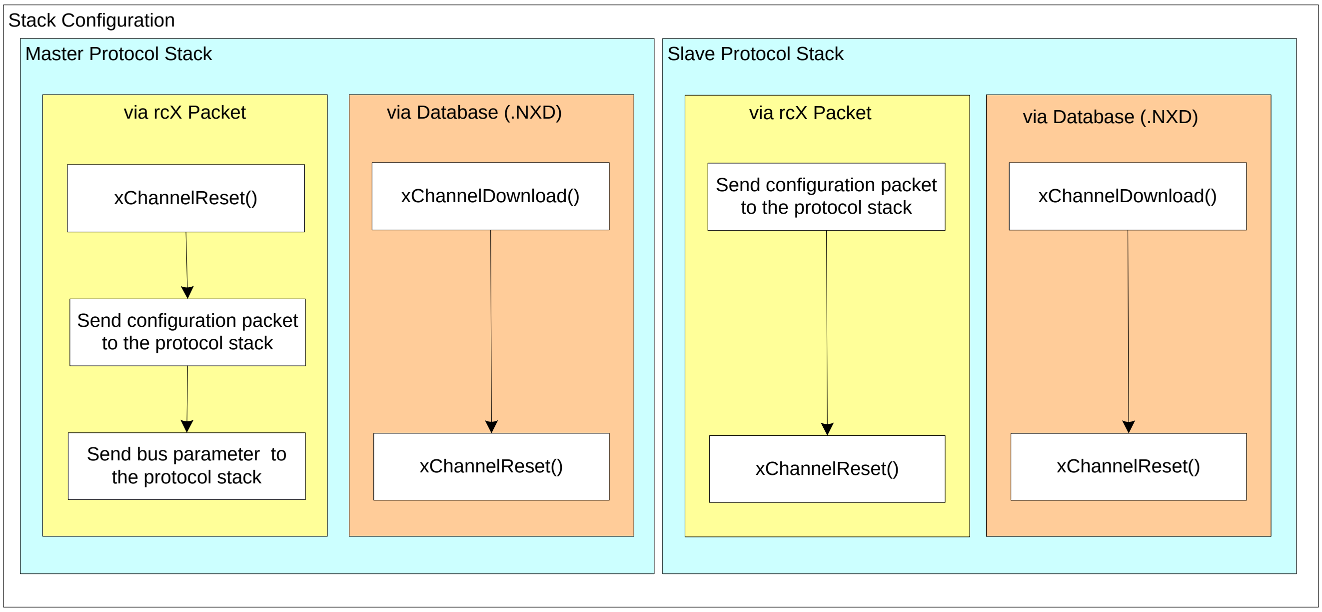
**Channel Handling - Close Channel**

- xChannelBusState(...CIFX_BUS_STATE_OFF...) **Switch off BUS Communication**
  Timeout <> 0, the function waits until the BUS is OFF
  Standard Timeout: 5000ms
- xChannelConfigLock(...CIFX_CONFIGURATION_UNLOCK...) **Unlock the Configuration**
  Unlock the configuration for further changes
- xChannelHostState(...CIFX_HOST_STATE_NOT_READY...) **User Application Closed**
  Signal the protocol stack, no application is online
- xChannelClose() **Close Channel**

**Driver Deinitialization**

- xDriverClose() **Close the cifX Driver**

# 6.2 Protocol stack configuration




### Master Stack Configuration - via rcX Packet

- xChannelReset(...CIFX_CHANNELINIT...) **Deactivate actual configuration**
  Maximum Timeout: 10000ms
- Send configuration
  This is described in the protocol API manual
- Send Bus Parameter
  This is described in the protocol API manual
  Configuration is activated automatically after writing the BUS parameters

### Master Stack Configuration - via Database

- xChannelDownload(...DOWNLOAD_CONFIGURATION...) **Download a database**
- xChannelReset(...CIFX_CHANNELINIT...) **Activate actual configuration**
  Maximum Timeout: 10000ms

### Slave Stack Configuration: via rcX Packets

- Send configuration data
- xChannelReset(...CIFX_CHANNELINIT...) **Activate the configuration**
  Maximum Timeout: 10000ms
- **Optional:** Set watchdog time (RCX_SET_WATCHDOG_TIME_REQ) via xChannelPutPacket() / xChannelGetPacket() )
  Standard Put/GetPacket() Timeout: 1000ms

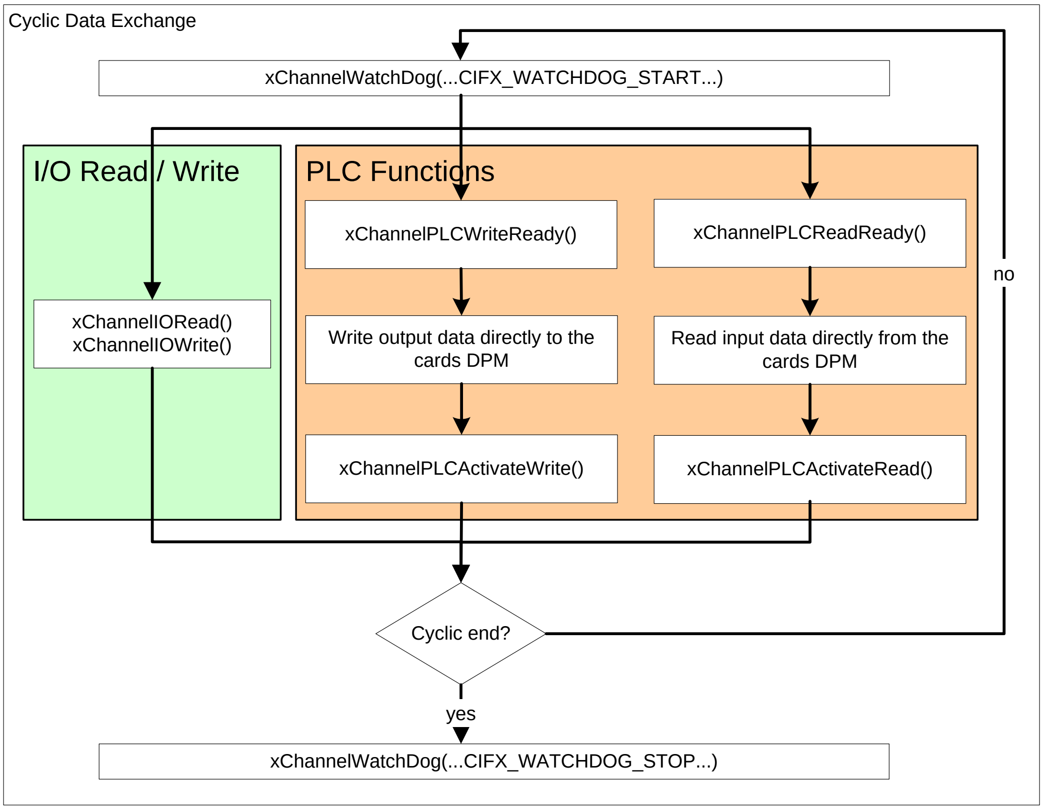
 

**Slave Stack Configuration - via Database**

- xChannelDownload(...DOWNLOAD_CONFIGURATION...) **Download a database**
- xChannelReset(...CIFX_CHANNELINIT...) **Activate actual configuration**

  Maximum Timeout: 10000ms

## 6.3 Cyclic data exchange




### Activate / Trigger Watchdog

- xChannelWatchdog (...CIFX_WATCHDOG_START...) **Activate/Trigger Watchdog**
- **Note:** xChannelWatchdog() must be called, with the parameter CIFX_WATCHDOG_START, within the configured watchdog time.

### Using I/O Read/Write Functions

- xChannelIORead() / xChannelIOWrite() **Read / Write I/O Data**

### Using PLC Functions

- xChannelPLCIsWriteReady() / xChannelPLCIsReadReady() **Use PLC Functions**

  Check if DPM access is allowed
- .... Read / Write data from/to the DPM I/O image areas
- xChannelPLCActivateRead() / xChannelPLCActivateWrite() **Activate I/O Data Transfer**

  Activate data transfer (DPM is switched to hardware)

### Deactivate Watchdog

- xChannelWatchdog (...CIFX_WATCHDOG_STOP...) **Deactivate Watchdog**

# 7 Error codes

| Error code | Symbol / Description |
|---|---|
| 0x00000000 | CIFX_NO_ERROR<br>No error |

### General error codes (0x800Axxxx)

| Error code | Symbol / Description |
|---|---|
| 0x800A0001 | CIFX_INVALID_POINTER<br>Invalid pointer (e.g. NULL was passed to the function) |
| 0x800A0002 | CIFX_INVALID_BOARD<br>No board with the given name / index available |
| 0x800A0003 | CIFX_INVALID_CHANNEL<br>No channel with the given index available |
| 0x800A0004 | CIFX_INVALID_HANDLE<br>An invalid handle was passed to the function |
| 0x800A0005 | CIFX_INVALID_PARAMETER<br>Invalid parameter passed to the function |
| 0x800A0006 | CIFX_INVALID_COMMAND<br>Command parameter is invalid |
| 0x800A0007 | CIFX_INVALID_BUFFERSIZE<br>The supplied buffer does not match the expected size |
| 0x800A0008 | CIFX_INVALID_ACCESS_SIZE<br>Invalid access size (e.g. I/O area size is exceeded by given offset and length) |
| 0x800A0009 | CIFX_FUNCTION_FAILED<br>Generic function failure |
| 0x800A000A | CIFX_FILE_OPEN_FAILED<br>A file cannot not be opened |
| 0x800A000B | CIFX_FILE_SIZE_ZERO<br>File size is zero |
| 0x800A000C | CIFX_FILE_LOAD_INSUFF_MEM<br>Insufficient memory to load file a file to RAM |
| 0x800A000D | CIFX_FILE_CHECKSUM_ERROR<br>File checksum comparison failed |
| 0x800A000E | CIFX_FILE_READ_ERROR<br>Error while reading file |
| 0x800A000F | CIFX_FILE_TYPE_INVALID<br>The given file is invalid for the operation |
| 0x800A0010 | CIFX_FILE_NAME_INVALID<br>Invalid filename given |
| 0x800A0011 | CIFX_FUNCTION_NOT_AVAILABLE<br>Function is not available on the driver |
| 0x800A0012 | CIFX_BUFFER_TOO_SHORT<br>The passed buffer is too short to receive all of the requested data |
| 0x800A0013 | CIFX_MEMORY_MAPPING_FAILED<br>Error mapping the dual port memory for later memory access |

| Error code | Symbol / Description |
|---|---|
| 0x800A0014 | CIFX_NO_MORE_ENTRIES<br>No more entries available. Returned by enumeration functions (e.g. xDriverEnumBoards(), directories etc.) |
| 0x800A0015 | CIFX_CALLBACK_MODE_UNKNOWN<br>Unknown callback handling mode |
| 0x800A0016 | CIFX_CALLBACK_CREATE_EVENT_FAILED<br>Creation of callback events failed |
| 0x800A0017 | CIFX_CALLBACK_CREATE_RECV_BUFFER<br>Creation of callback receive buffer failed |
| 0x800A0018 | CIFX_CALLBACK_ALREADY_USED<br>Another application has already registered a callback for the given event |
| 0x800A0019 | CIFX_CALLBACK_NOT_REGISTERED<br>A callback was not registered before |
| 0x800A001A | CIFX_INTERRUPT_DISABLED<br>Device interrupt is disabled. The executed function expects an enabled hardware interrupt (depending on the driver this must be done either by the device configuration or driver setup program). |

*Table 84: General Error Codes (0x800Axxxx)*

## Driver-related error codes (0x800Bxxxx)

| Error code | Symbol / Description |
|---|---|
| 0x800B0001 | CIFX_DRV_NOT_INITIALIZED<br>Driver was not correctly initialized during startup or driver is already closed |
| 0x800B0002 | CIFX_DRV_INIT_STATE_ERROR<br>Initialization state error. Hardware does not show correct or expected states and information in the DPM after a reset or boot start |
| 0x800B0003 | CIFX_DRV_READ_STATE_ERROR<br>Driver read state error |
| 0x800B0004 | CIFX_DRV_CMD_ACTIVE<br>The called function is in use by another program instance or application |
| 0x800B0005 | CIFX_DRV_DOWNLOAD_FAILED<br>General error during download (e.g. boot loader could not be downloaded or started) |
| 0x800B0006 | CIFX_DRV_WRONG_DRIVER_VERSION<br>Wrong driver version |
| 0x800B0030 | CIFX_DRV_DRIVER_NOT_LOADED<br>CIFX driver is not loaded / running. Failed to open or start the driver, returned by xDriverOpen() |
| 0x800B0031 | CIFX_DRV_INIT_ERROR<br>Failed to initialize the driver |
| 0x800B0032 | CIFX_DRV_CHANNEL_NOT_INITIALIZED<br>Channel not initialized (e.g. xChannelOpen() not called) |
| 0x800B0033 | CIFX_DRV_IO_CONTROL_FAILED<br>Function call into the driver failed<br>(e.g. used by the Windows API DLL to signal problems in IO-Control driver calls) |
| 0x800B0034 | CIFX_DRV_NOT_OPENED<br>Driver was not opened by calling xDriverOpen() |
| 0x800B0040 | CIFX_DRV_DOWNLOAD_STORAGE_UNKNOWN<br>Unknown download storage type (RAM/FLASH based) found |
| 0x800B0041 | CIFX_DRV_DOWNLOAD_FW_WRONG_CHANNEL<br>Channel number for a firmware download not supported |
| 0x800B0042 | CIFX_DRV_DOWNLOAD_MODULE_NO_BASEOS<br>Modules are not allowed without a Base OS firmware |

*Table 85: Driver-related error codes (0x800Bxxxx)*

## Device / Communication-related error codes (0x800Cxxxx)

| Error code | Symbol / Description |
|---|---|
| 0x800C0010 | CIFX_DEV_DPM_ACCESS_ERROR<br>Dual port memory not accessible (e.g. board not found, wrong dual port memory content) |
| 0x800C0011 | CIFX_DEV_NOT_READY<br>Device is not ready (NSF_READY or NCF_READY flag is not set)<br>The system device or communication channel is not working |
| 0x800C0012 | CIFX_DEV_NOT_RUNNING<br>Device is not running (NCF_RUNNING flag is not set). The communication channel is not configured |
| 0x800C0013 | CIFX_DEV_WATCHDOG_FAILED<br>Watchdog test failed |
| 0x800C0015 | CIFX_DEV_SYSERR<br>Error in handshake flags |
| 0x800C0016 | CIFX_DEV_MAILBOX_FULL<br>Send mailbox is full. The PutPacket() function was not able to write a packet to the device mailbox. Either the mailbox state does not show empty or no more resources on the device available.<br>(NSF_SEND_MBX_ACK / HSF_SEND_MBX_CMD or NCF_SEND_MBX_ACK / HCF_SEND_MBX_CMD flags in wrong state or mailbox counter usPackagesAccepted = 0) |
| 0x800C0017 | CIFX_DEV_PUT_TIMEOUT<br>Send packet timeout. The PutPacket() function was not able to write a packet to the device mailbox and the wait time in PutPacket() has expired. Either the mailbox state does not show empty or no more resources on the device available.<br>(NSF_SEND_MBX_ACK / HSF_SEND_MBX_CMD or NCF_SEND_MBX_ACK / HCF_SEND_MBX_CMD flags in wrong state or mailbox counter usPackagesAccepted = 0) |
| 0x800C0018 | CIFX_DEV_GET_TIMEOUT<br>Receive packet timeout. GetPacket() function was not able to read a packet from the device and the wait time in GetPacket() has expired. Either the mailbox state does not show a packet available or the device has not sent a packet.<br>(NSF_RECV_MBX_CMD / HSF_RECV_MBX_ACK or NCF_RECV_MBX_CMD / HCF_RECV_MBX_ACK flags in wrong state or mailbox counter usWaitingPackages = 0) |
| 0x800C0019 | CIFX_DEV_GET_NO_PACKET<br>No packet available. The GetPacket() function was called with timeout = 0 and the function was not able to read a packet from the device. Either the mailbox state does not show a packet available or the device has not sent a packet.<br>(NSF_RECV_MBX_CMD / HSF_RECV_MBX_ACK or NCF_RECV_MBX_CMD / HCF_RECV_MBX_ACK flags in wrong state or mailbox counter usWaitingPackages = 0) |
| 0x800C001A | CIFX_DEV_MAILBOX_TOO_SHORT<br>Mailbox is to short for the given packet. The packet send by PutPacket() does not fit into the mailbox. |
| 0x800C0020 | CIFX_DEV_RESET_TIMEOUT<br>Reset command timeout. The device was not reaching READY state, in the given reset timeout, after the application has initiated a reset (RCX_COMM_COS_READY flag not set). |
| 0x800C0021 | CIFX_DEV_NO_COM_FLAG<br>Communication flag not set. The fieldbus protocol stack has no communication to the fieldbus devices. Either the cable is disconnected or no other device is connected to the wire (NCF_COMMUNICATING flag not set). |

| Error code | Symbol / Description |
| --- | --- |
| 0x800C0022 | CIFX_DEV_EXCHANGE_FAILED<br>I/O data exchange failed. Function xChannelIORead() or xChannelIOWrite() fails, because the device does not allow to access the I/O data image.<br>(NCF_PDIN / NCF_PDOUT flags are not in the state allowing access to the I/O process data image) |
| 0x800C0023 | CIFX_DEV_EXCHANGE_TIMEOUT<br>I/O data exchange timeout. The given timeout in xChannelIORead() / xChannelIOWrite() expires while the function is waiting to get access to the process data image.<br>(NCF_PDIN / NCF_PDOUT flags are not in the state allowing access to the I/O process data image) |
| 0x800C0024 | CIFX_DEV_COM_MODE_UNKNOWN<br>Unknown I/O data exchange mode (mode is not within 0..5) |
| 0x800C0025 | CIFX_DEV_FUNCTION_FAILED<br>Device function failed |
| 0x800C0026 | CIFX_DEV_DPMSIZE_MISMATCH<br>DPM size differs from configuration, The firmware signals a communication channel size which does not fit into the maximum DPM size defined by the hardware or defined by the user. |
| 0x800C0027 | CIFX_DEV_STATE_MODE_UNKNOWN<br>Unknown state mode |
| 0x800C0028 | CIFX_DEV_HW_PORT_IS_USED<br>Device is accessed either by another application or another instance.<br>- Driver / device can't be unloaded, open connection to the system device or a communication channels still active<br>- xChannelOpen() can't be executed because it is currently used by another application |
| 0x800C0029 | CIFX_DEV_CONFIG_LOCK_TIMEOUT<br>Failed lock the communication channels configuration within the given time. xChannelConfigLock() wait time expired (RCX_COMM_COS_CONFIG_LOCKED flag not set). |
| 0x800C002A | CIFX_DEV_CONFIG_UNLOCK_TIMEOUT<br>Failed to unlock the communication channel configuration within the given time. xChannelConfigLock() wait time expired (RCX_COMM_COS_CONFIG_LOCKED flag not cleared) |
| 0x800C002B | CIFX_DEV_HOST_STATE_SET_TIMEOUT<br>Wait time expires during xChannelHostState() without reaching CIFX_HOST_STATE_READY.<br>(The function was not able to set the RCX_APP_COS_APP_READY flag or the device has not acknowledged the new status in time) |
| 0x800C002C | CIFX_DEV_HOST_STATE_CLEAR_TIMEOUT<br>Wait time expires during xChannelHostState() without reaching CIFX_HOST_STATE_NOT_READY<br>(The function was not able to clear the RCX_APP_COS_APP_READY flag or the device has not acknowledged the new status in time) |
| 0x800C002D | CIFX_DEV_INITIALIZATION_TIMEOUT<br>Timeout during device / channel initialization |
| 0x800C002E | CIFX_DEV_BUS_STATE_ON_TIMEOUT<br>Wait time expires during xChannelBusState() without reaching CIFX_BUS_STATE_ON (RCX_COMM_COS_BUS_ON flag not set)<br>Using a timeout, the function will activate fieldbus communication and waits until communication to another fieldbus device is available (NCF_COMMUNICATION flag is set) |

| Error code | Symbol / Description |
|---|---|
| 0x800C002F | CIFX_DEV_BUS_STATE_OFF_TIMEOUT<br>Wait time expires during xChannelBusState() without reaching CIFX_BUS_STATE_OFF.<br>(The function was not able to clear the RCX_APP_COS_BUS_ON flag or the device has not acknowledged the new status in time and still signals bus communication is active by RCX_COM_COS_BUS_ON). |
| 0x800C0040 | CIFX_DEV_MODULE_ALREADY_RUNNING<br>Firmware module (NXO) download and start failed because a module is already running |
| 0x800C0041 | CIFX_DEV_MODULE_ALREADY_EXISTS<br>Firmware module (NXO) download was skipped because the module already exists |
| 0x800C0050 | CIFX_DEV_DMA_INSUFF_BUFFER_COUNT<br>Number of configured DMA buffers insufficient (at least 8 buffers are expected)<br>Or xChannelDMAState() is used without previously configured DMA buffers. |
| 0x800C0051 | CIFX_DEV_DMA_BUFFER_TOO_SMALL<br>DMA buffers size too small (min. size 256 Byte) |
| 0x800C0052 | CIFX_DEV_DMA_BUFFER_TOO_BIG<br>DMA buffers size too big (max. size 63,75 KByte) |
| 0x800C0053 | CIFX_DEV_DMA_BUFFER_NOT_ALIGNED<br>DMA buffer alignment failed (must be 256 Byte) |
| 0x800C0054 | CIFX_DEV_DMA_HANSHAKEMODE_NOT_SUPPORTED<br>I/O process data exchange mode "uncontrolled" not allowed when DMA transfer is activated |
| 0x800C0055 | CIFX_DEV_DMA_IO_AREA_NOT_SUPPORTED<br>I/O process data area index in DMA mode not supported (only area 0 possible) |
| 0x800C0056 | CIFX_DEV_DMA_STATE_ON_TIMEOUT<br>Failed to set DMA transfer to "ON" within the given wait time in xChannelDMAState().<br>(The device has not acknowledged the new status or not set the RCX_COM_COS_DMA flag) |
| 0x800C0057 | CIFX_DEV_DMA_STATE_OFF_TIMEOUT<br>Failed to set DMA transfer to "OFF" within the given wait time in xChannelDMAState().<br>(The device has not acknowledged the new status or not cleared the RCX_COM_COS_DMA flag) |
| 0x800C0058 | CIFX_DEV_SYNC_STATE_INVALID_MODE<br>Device is in invalid mode for the command initiated by xChannelSyncState().<br>The mode must be either "SYNC Host Controlled" (RCX_SYNC_MODE_HST_CTRL) or "SYNC Device Controlled" (RCX_SYNC_MODE_DEV_CTRL) |
| 0x800C0059 | CIFX_DEV_SYNC_STATE_TIMEOUT<br>Wait time expired during xChannelSyncState(..,CIFX_SYNC_WAIT_CMD, ).<br>Device does not signal the expected synchronization handshake flag state |

*Table 86: Device / Communication-related error codes (0x800Cxxxx)*

# 8 Appendix

## 8.1 List of tables



## 8.2 List of figures



## 8.3 Legal notes



### 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.

## 8.4 Contacts

### Headquarters

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

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

### Subsidiaries

**China**
Hilscher 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