# Table of Contents

1 Introduction .......................................................................................................................... 4  
1.1 About this Document ........................................................................................................ 4  
1.2 List of Revisions ................................................................................................................ 4  
1.3 Terms, Abbreviations and Definitions ............................................................................. 5  
1.4 References to Documents ................................................................................................ 5  

2 Application Note .................................................................................................................. 6  
2.1 Component Overview for Host Applications .................................................................. 6  
2.2 Component Overview for netX Applications ................................................................... 7  
2.3 Availability of API Functions ........................................................................................... 8  

3 API Basics ............................................................................................................................ 11  
3.1 DPM Layout, Devices and Channels .............................................................................. 12  
3.2 Basic Fieldbus Data Handling ....................................................................................... 13  
3.3 Fieldbus Specific Information and Functions ................................................................. 13  

4 CIFX API (Application Programming Interface) ................................................................. 14  
4.1 API Header Files .............................................................................................................. 14  
4.2 Driver Functions .............................................................................................................. 15  
4.3 System Device Functions ............................................................................................... 16  
4.4 Communication Channel Functions .............................................................................. 16  
4.5 Structure Definitions ...................................................................................................... 18  
4.5.1 Driver Information ...................................................................................................... 18  
4.5.2 Board Information ...................................................................................................... 18  
4.5.3 System Channel Information ..................................................................................... 19  
4.5.4 Communication Channel Information ...................................................................... 21  
4.6 Driver Related Functions ............................................................................................... 22  
4.6.1 xDriverOpen .............................................................................................................. 22  
4.6.2 xDriverClose ............................................................................................................. 23  
4.6.3 xDriverGetInformation ............................................................................................ 24  
4.6.4 xDriverGetErrorDescription ................................................................................... 25  
4.6.5 xDriverEnumBoards .................................................................................................. 26  
4.6.6 xDriverEnumChannels .............................................................................................. 27  
4.6.7 xDriverRestartDevice .............................................................................................. 28  
4.6.8 xDriverMemoryPointer ............................................................................................ 29  
4.7 System Device Specific Functions ................................................................................... 31  
4.7.1 xSysdeviceOpen ....................................................................................................... 31  
4.7.2 xSysdeviceClose ........................................................................................................ 32  
4.7.3 xSysdeviceInfo ......................................................................................................... 33  
4.7.4 xSysdeviceReset ....................................................................................................... 34  
4.7.5 xSysdeviceBootstrap ................................................................................................. 35  
4.7.6 xSysdeviceGetMBXState .......................................................................................... 36  
4.7.7 xSysdevicePutPacket ............................................................................................... 37  
4.7.8 xSysdeviceGetPacket .............................................................................................. 38  
4.7.9 xSysdeviceDownload ............................................................................................... 39  
4.7.10 xSysdeviceFindFirstFile ......................................................................................... 40  
4.7.11 xSysdeviceFindNextFile ......................................................................................... 41  
4.7.12 xSysdeviceUpload ................................................................................................. 42  
4.7.13 xSysdeviceExtendedMemory .................................................................................. 43  
4.8 Channel Specific Functions ............................................................................................ 47  
4.8.1 xChannelOpen ......................................................................................................... 47  
4.8.2 xChannelClose .......................................................................................................... 48  
4.8.3 xChannelDownload .................................................................................................... 49  
4.8.4 xChannelFindFirstFile ............................................................................................. 50  
4.8.5 xChannelFindNextFile ............................................................................................. 51  
4.8.6 xChannelUpload ....................................................................................................... 52  
4.8.7 xChannelGetMBXState ............................................................................................. 53  
4.8.8 xChannelPutPacket ................................................................................................. 54  
4.8.9 xChannelGetPacket ................................................................................................. 55  
4.8.10 xChannelGetSendPacket ....................................................................................... 56  
4.8.11 xChannelReset ........................................................................................................ 57  
4.8.12 xChannelInfo .......................................................................................................... 58  
4.8.13 xChannel0Info ........................................................................................................ 59
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

<table>
<thead>
<tr>
<th>Rev</th>
<th>Date</th>
<th>Name</th>
<th>Chapter</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2012-12-11</td>
<td>RM</td>
<td>all</td>
<td>Extracted from the CIFX Device Driver - Windows DRV 21 EN.pdf</td>
</tr>
<tr>
<td>2</td>
<td>2013-02-18</td>
<td>RM</td>
<td>4.8.28</td>
<td>Section Fieldbus Synchronization Handling added. Tables in chapter Error Codes revised.</td>
</tr>
</tbody>
</table>

Table 1: List of Revisions
1.3 Terms, Abbreviations and Definitions

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>netX</td>
<td>Hilscher highly integrated network controller</td>
</tr>
<tr>
<td>rcX</td>
<td>Hilscher Real Time Communication System for netX</td>
</tr>
<tr>
<td>cifX</td>
<td>Communication Interface based on netX</td>
</tr>
<tr>
<td>comX</td>
<td>Communication Module based on netX</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>DPM</td>
<td>Dual-Port Memory Physical memory area, connected to a host processor.</td>
</tr>
<tr>
<td></td>
<td>Standard interface to Hilscher communication boards like CIFX/COMX or netX evaluation boards (Attention: DPM may also be used as a shortcut for PROFIBUS-DP Master field bus protocol).</td>
</tr>
<tr>
<td>SHM</td>
<td>Shared Memory System memory area shared between different processes inside a software application</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Calls</td>
</tr>
<tr>
<td>DPM Manual</td>
<td>Description of the standard Hilscher DPM layout and functionality</td>
</tr>
<tr>
<td>netX Transport</td>
<td>Diagnostics and remote access functions to netX based remote devices via serial interfaces</td>
</tr>
<tr>
<td>CIFX Toolkit</td>
<td>C source-code based implementation of the standard Hilscher DPM access functions</td>
</tr>
<tr>
<td>SDO</td>
<td>Service Data Object</td>
</tr>
<tr>
<td>PDO</td>
<td>Process Data Object</td>
</tr>
</tbody>
</table>

Table 2: Terms, Abbreviations and Definitions

1.4 References to Documents

This document refers to the following documents:


Definition and description of the serial remote access functions to Hilscher netX based devices

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/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).

![CIFX API Components](image)

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](image-url)
# 2.3 Availability of API Functions

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Short Description</th>
<th>DPM</th>
<th>DPM via SPI</th>
<th>USB / serial / Ethernet</th>
<th>netX</th>
</tr>
</thead>
<tbody>
<tr>
<td>xDriverOpen</td>
<td>Opens the driver</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xDriverClose</td>
<td>Closes the driver</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xDriverGetInformation</td>
<td>Retrieves driver information</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xDriverGetErrorDescription</td>
<td>Retrieves an error code description</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xDriverEnumBoards</td>
<td>Enumerate available boards/devices</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xDriverEnumChannels</td>
<td>Enumerate available channels on a specific board</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xDriverRestartDevice</td>
<td>Restart a device</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>xDriverMemoryPointer</td>
<td>Get/Release a pointer to the dual port memory.</td>
<td>X</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

*Only be used for debugging, purpose*

<table>
<thead>
<tr>
<th>Function</th>
<th>Short Description</th>
<th>DPM</th>
<th>DPM via SPI</th>
<th>USB / serial / Ethernet</th>
<th>netX</th>
</tr>
</thead>
<tbody>
<tr>
<td>xSysdeviceOpen</td>
<td>Opens a connection a system device</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xSysdeviceClose</td>
<td>Closes a connection to a system device</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xSysdeviceInfo</td>
<td>Get System device information</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xSysdeviceReset</td>
<td>Perform a device reset</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xSysdeviceBootstart</td>
<td>Perform a device boot start</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>xSysdeviceGetMBXState</td>
<td>Retrieves the system mailbox state</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xSysdeviceGetPacket</td>
<td>Read a pending packet</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xSysdevicePutPacket</td>
<td>Send a packet</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xSysdeviceDownload</td>
<td>Downloads a file/configuration/firmware</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xSysdeviceFindFirstFile</td>
<td>Find the first file in the given directory</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xSysdeviceFindNextFile</td>
<td>Find the next file entry in the given directory</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xSysdeviceUpload</td>
<td>Uploads a file/configuration/firmware</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>xSysdeviceExtendedMemory</td>
<td>Get a pointer an extended memory area</td>
<td>X</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

Table 4: List of API Functions – Driver Functions

Table 5: List of API Functions – System Device Functions
<table>
<thead>
<tr>
<th>Function</th>
<th>Short Description</th>
<th>DPM</th>
<th>DPM via SPI</th>
<th>USB / serial / Ethernet</th>
<th>netX</th>
<th>RPC</th>
<th>Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>xChannelOpen</td>
<td>Opens a communication channel</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelClose</td>
<td>Closes a communication channel</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>Asynchronous services (Packets)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xChannelGetMBXState</td>
<td>Retrieve the channels mailbox state</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelGetPacket</td>
<td>Read packet from the channel mailbox</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelPutPacket</td>
<td>Send a packet to the channel mailbox</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelGetSendPacket</td>
<td>Read back the packet sent</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>Device Administrational/Informational functions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xChannelDownload</td>
<td>Download a file/configuration to the channel</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelReset</td>
<td>Reset the channel</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X(4)</td>
<td></td>
</tr>
<tr>
<td>xChannelInfo</td>
<td>Retrieve channel specific information</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelWatchdog</td>
<td>Activate/Deactivate/Trigger Watchdog</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelHostState</td>
<td>Set the Application state flag (signal application is running or not)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelBusState</td>
<td>Set the bus state flag (start or stop fieldbus communication)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelControlBlock</td>
<td>Access the Channels control block</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelCommonStatusBlock</td>
<td>Access to the common status block</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelExtendedStatusBlock</td>
<td>Access to the extended status block</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelUserBlock</td>
<td>Access user block (not implemented yet!)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>Cyclic Data services (I/O’s)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xChannelIORead</td>
<td>Instructs the device to place the latest data into the DPM and passes them to the user</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelIOWrite</td>
<td>Copies the data to the DPM and waits for the firmware to retrieve them</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelIOReadSendData</td>
<td>Reads back the last send data</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td><strong>Cyclic Data services (I/O’s, PLC optimized)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xChannelPLCMemoryPtr</td>
<td>Get a pointer to the IO Block</td>
<td>X</td>
<td>O(2)</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelPLCActivateRead</td>
<td>Instruct the firmware to place the latest input data into the dual port (no wait for completion)</td>
<td>X</td>
<td>O(2)</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelPLCActivateWrite</td>
<td>Instruct the firmware to retrieve the latest output data from the dual port (no wait for completion)</td>
<td>X</td>
<td>O(2)</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelPLCIsReadReady</td>
<td>Checks if the last Read Activation has finished</td>
<td>X</td>
<td>O(2)</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>xChannelPLCIsWriteReady</td>
<td>Checks if the last Write Activation has finished</td>
<td>X</td>
<td>O(2)</td>
<td>O</td>
<td>O</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
### Table 6: List of API Functions – Communication Channel Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Short Description</th>
<th>DPM</th>
<th>DPM via SPI</th>
<th>USB / serial / Ethernet</th>
<th>netX</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DMA services</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xChannelDMAState</td>
<td>Activate/Deactivate DMA mode</td>
<td>X(1)</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td><strong>Bus synchronous operation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xChannelSyncState</td>
<td>Wait for synchronization event or trigger/acknowledge sync</td>
<td>X</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td><strong>Notification services (only available in Interrupt mode)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xChannelRegisterNotification</td>
<td>Register a notification callback</td>
<td>X</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>xChannelUnregisterNotification</td>
<td>Un-register a notification callback</td>
<td>X</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

(1) PCI / PCIe hardware only  
(2) Special implementation necessary  
(3) A system reset will reset the whole device. Boot start is not implemented.  
(4) A system reset will reset the whole device
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 DPM Layout:**

- **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:  \textit{cifXUser.h}  
Error definitions:  \textit{cifXError.h}  

4.2  Driver Functions

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

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

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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xSysdeviceOpen</td>
<td>Opens a connection to a board's system device</td>
</tr>
<tr>
<td>xSysdeviceClose</td>
<td>Closes a connection to a system device</td>
</tr>
<tr>
<td>xSysdeviceInfo</td>
<td>Get System device specific information (e.g. mailbox size)</td>
</tr>
<tr>
<td>xSysdeviceReset</td>
<td>Perform a device reset</td>
</tr>
<tr>
<td>xSysdeviceBootstart</td>
<td>Perform a device boot start. This will activate the 2nd Stage bootloader. An available firmware will not be started.</td>
</tr>
<tr>
<td>xSysdeviceGetMBXState</td>
<td>Retrieves the system mailbox state</td>
</tr>
<tr>
<td>xSysdeviceGetPacket</td>
<td>Retrieves a pending packet from the system mailbox</td>
</tr>
<tr>
<td>xSysdevicePutPacket</td>
<td>Send a packet to the system mailbox</td>
</tr>
<tr>
<td>xSysdeviceDownload</td>
<td>Downloads a file/configuration/firmware to the device</td>
</tr>
<tr>
<td>xSysdeviceFindFirstFile</td>
<td>Find the first file entry in the given directory</td>
</tr>
<tr>
<td>xSysdeviceFindNextFile</td>
<td>Find the next file entry in the given directory</td>
</tr>
<tr>
<td>xSysdeviceUpload</td>
<td>Uploads a file/configuration/firmware from the device</td>
</tr>
<tr>
<td>xSysdeviceExtendedMemory</td>
<td>Get a pointer to an available extended memory area</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xChannelOpen</td>
<td>Opens a connection to a communication channel</td>
</tr>
<tr>
<td>xChannelClose</td>
<td>Closes a connection</td>
</tr>
<tr>
<td></td>
<td><strong>Asynchronous services (Packets)</strong></td>
</tr>
<tr>
<td>xChannelGetMBXState</td>
<td>Retrieve the channels mailbox state</td>
</tr>
<tr>
<td>xChannelGetPacket</td>
<td>Retrieve a pending packet from the channel mailbox</td>
</tr>
<tr>
<td>xChannelPutPacket</td>
<td>Send a packet to the channel mailbox</td>
</tr>
<tr>
<td>xChannelGetSendPacket</td>
<td>Read back the last sent packet</td>
</tr>
<tr>
<td></td>
<td><strong>Device Administrational/Informational functions</strong></td>
</tr>
<tr>
<td>xChannelDownload</td>
<td>Download a file/configuration to the channel</td>
</tr>
<tr>
<td>xChannelReset</td>
<td>Reset the channel</td>
</tr>
<tr>
<td>xChannelInfo</td>
<td>Retrieve channel specific information</td>
</tr>
<tr>
<td>xChannelWatchdog</td>
<td>Activate/Deactivate/Trigger the channel Watchdog</td>
</tr>
<tr>
<td>xChannelHostState</td>
<td>Set the application state flag in the application COS flags, to signal the hardware if an application is running or not</td>
</tr>
<tr>
<td>xChannelBusState</td>
<td>Set the bus state flag in the application COS state flags, to start or stop fieldbus communication.</td>
</tr>
<tr>
<td>xChannelControlBlock</td>
<td>Access the channel control block</td>
</tr>
<tr>
<td>xChannelCommonStatusBlock</td>
<td>Access to the common status block</td>
</tr>
<tr>
<td>xChannelExtendedStatusBlock</td>
<td>Access to the extended status block</td>
</tr>
<tr>
<td>xChannelUserBlock</td>
<td>Access user block (not implemented yet!)</td>
</tr>
<tr>
<td></td>
<td><strong>Cyclic Data services (I/O’s)</strong></td>
</tr>
<tr>
<td>xChannelIORead</td>
<td>Instructs the device to place the latest data into the DPM and passes them to the user</td>
</tr>
<tr>
<td>xChannelIOWrite</td>
<td>Copies the data to the DPM and waits for the firmware to retrieve them</td>
</tr>
<tr>
<td>xChannelIOReadSendData</td>
<td>Reads back the last send data</td>
</tr>
<tr>
<td></td>
<td><strong>Cyclic Data services (I/O’s, PLC optimized)</strong></td>
</tr>
<tr>
<td>xChannelPLCMemoryPtr</td>
<td>Get a pointer to the I/O memory block</td>
</tr>
<tr>
<td>xChannelPLCActivateRead</td>
<td>Instruct the firmware to place the latest input data into the I/O memory block (no wait for completion)</td>
</tr>
<tr>
<td>xChannelPLCActivateWrite</td>
<td>Instruct the firmware to retrieve the latest output data from the I/O memory block (no wait for completion)</td>
</tr>
<tr>
<td>xChannelPLClReadReady</td>
<td>Checks if the last read activation has finished</td>
</tr>
<tr>
<td>xChannelPLClWriteReady</td>
<td>Checks if the last write activation has finished</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td><strong>DMA services</strong></td>
<td></td>
</tr>
<tr>
<td>xChannelDMAState</td>
<td>Activate/Deactivate DMA mode</td>
</tr>
<tr>
<td><strong>Bus synchronous operation</strong></td>
<td></td>
</tr>
<tr>
<td>xChannelSyncState</td>
<td>Wait for synchronization events or trigger / acknowledge a sync event</td>
</tr>
<tr>
<td><strong>Notification services (only available in Interrupt mode)</strong></td>
<td></td>
</tr>
<tr>
<td>xChannelRegisterNotification</td>
<td>Register a notification callback</td>
</tr>
<tr>
<td>xChannelUnregisterNotification</td>
<td>Un-register a notification callback</td>
</tr>
</tbody>
</table>

*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.

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abDriverVersion</td>
<td>uint8_t[32]</td>
<td>Human readable driver name and version</td>
</tr>
<tr>
<td>ulBoardCnt</td>
<td>uint32_t</td>
<td>Number of handled boards</td>
</tr>
</tbody>
</table>

Table 10: Driver Information Structure

4.5.2 Board Information

The board information structure is used, when enumerating boards.

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

Table 11: Board Information Structure
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

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

Table 12: System Channel Information

Command: CIFX_INFO_CMD_SYSTEM_INFO_BLOCK

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

Table 13: System Channel Info Block
Command: CFX_INFO_CMD_SYSTEM_CHANNEL_BLOCK

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abInfoBlock</td>
<td>uint8_t[8][16]</td>
<td>Channel information in the system channel</td>
</tr>
</tbody>
</table>

| Table 14: System Channel - Channel Info Block |

Area definitions in cifXUser.h:

- CFX_MAX_NUMBER_OF_CHANNEL_DEFINTION = 8
- CFX_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: CFX_INFO_CMD_SYSTEM_CONTROL_BLOCK

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ulSystemCommandCOS</td>
<td>uint32_t</td>
<td>System channel host COS flags</td>
</tr>
<tr>
<td>ulReserved</td>
<td>uint32_t</td>
<td>unused/reserved</td>
</tr>
</tbody>
</table>

| Table 15: System Channel Control Block |

Command: CFX_INFO_CMD_SYSTEM_STATUS_BLOCK

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ulSystemCOS</td>
<td>uint32_t</td>
<td>System channel device COS flags</td>
</tr>
<tr>
<td>ulSystemStatus</td>
<td>uint32_t</td>
<td>Actual system state</td>
</tr>
<tr>
<td>ulSystemError</td>
<td>uint32_t</td>
<td>Actual system error</td>
</tr>
<tr>
<td>ulReserved1</td>
<td>uint32_t</td>
<td>unused/reserved</td>
</tr>
<tr>
<td>ulTimeSinceStart</td>
<td>uint32_t</td>
<td>Time since system start in seconds</td>
</tr>
<tr>
<td>usCpuLoad</td>
<td>uint16_t</td>
<td>CPU load in 0,01% units (10000 =&gt; 100%)</td>
</tr>
<tr>
<td>usReserved</td>
<td>uint16_t</td>
<td>Reserved for later use</td>
</tr>
<tr>
<td>ulHWFeatures</td>
<td>Uint32_t</td>
<td>Information about hardware features (e.g. MRAM / RTC)</td>
</tr>
<tr>
<td>abReserved</td>
<td>uint8_t[36]</td>
<td>unused/reserved</td>
</tr>
</tbody>
</table>

| Table 16: System Channel Status Block |
4.5.4 Communication Channel Information

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

<table>
<thead>
<tr>
<th>Element</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abBoardName</td>
<td>uint8_t[16]</td>
<td>This is the name of the board which can be used for opening a channel or the system device on it.</td>
</tr>
<tr>
<td>abBoardAlias</td>
<td>uint8_t[16]</td>
<td>This is an alternate, user-definable name for the device</td>
</tr>
<tr>
<td>ulDeviceNumber</td>
<td>uint32_t</td>
<td>Device number (as found on the matrix label)</td>
</tr>
<tr>
<td>ulSerialNumber</td>
<td>uint32_t</td>
<td>Serial number (as found on the matrix label)</td>
</tr>
<tr>
<td>usFWMajor</td>
<td>uint16_t</td>
<td>Major version number of firmware</td>
</tr>
<tr>
<td>usFWMinor</td>
<td>uint16_t</td>
<td>Minor version number of firmware</td>
</tr>
<tr>
<td>usFWBuild</td>
<td>uint16_t</td>
<td>Build number of firmware</td>
</tr>
<tr>
<td>usFWRevision</td>
<td>uint16_t</td>
<td>Revision version number of firmware</td>
</tr>
<tr>
<td>bFWNameLength</td>
<td>uint8_t</td>
<td>Length of firmware name</td>
</tr>
<tr>
<td>abFWName</td>
<td>uint8_t[63]</td>
<td>Firmware name</td>
</tr>
<tr>
<td>usFWYear</td>
<td>uint16_t</td>
<td>Build year of firmware</td>
</tr>
<tr>
<td>bFWMonth</td>
<td>uint8_t</td>
<td>Build month of firmware (1..12)</td>
</tr>
<tr>
<td>bFDaday</td>
<td>uint8_t</td>
<td>Build day of firmware (1..31)</td>
</tr>
<tr>
<td>ulChannelError</td>
<td>uint32_t</td>
<td>Communication channel error from the &quot;Common Status Block&quot;</td>
</tr>
<tr>
<td>ulOpenCnt</td>
<td>uint32_t</td>
<td>Number of calls to \texttt{xChannelOpen} for this channel</td>
</tr>
<tr>
<td>ulPutPacketCnt</td>
<td>uint32_t</td>
<td>Number of successful transmitted packets</td>
</tr>
<tr>
<td>ulGetPacketCnt</td>
<td>uint32_t</td>
<td>Number of successfully received packets</td>
</tr>
<tr>
<td>ulMailboxSize</td>
<td>uint32_t</td>
<td>Mailbox size in Bytes</td>
</tr>
<tr>
<td>ulI0lnAreaCnt</td>
<td>uint32_t</td>
<td>Number of I/O Input areas</td>
</tr>
<tr>
<td>ulI0OutAreaCnt</td>
<td>uint32_t</td>
<td>Number of I/O output areas</td>
</tr>
<tr>
<td>ulHskSize</td>
<td>uint32_t</td>
<td>RCX_HANDSHAKE_SIZE_8BIT (0x01) or RCX_HANDSHAKE_SIZE_16BIT (0x02)</td>
</tr>
<tr>
<td>ulNetxFlags</td>
<td>uint32_t</td>
<td>Actual netX communication flags \texttt{(usNetxCommFlag)}</td>
</tr>
<tr>
<td>ulHostFlags</td>
<td>uint32_t</td>
<td>Actual host communication flags \texttt{(usHostCommFlags)}</td>
</tr>
<tr>
<td>ulHostCOSFlags</td>
<td>uint32_t</td>
<td>Actual application COS flags \texttt{(ulApplicationCOS of Control Block)}</td>
</tr>
<tr>
<td>ulDeviceCOSFlags</td>
<td>uint32_t</td>
<td>Actual communication COS flags \texttt{(ulCommunicationCOS of Common Status Block)}</td>
</tr>
</tbody>
</table>

*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:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>phDriver</td>
<td>CIFXHANDLE*</td>
<td>returned handle to the driver</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDriver</td>
<td>CIFXHANDLE</td>
<td>Handle returned by xDriverOpen</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDriver</td>
<td>CIFXHANDLE</td>
<td>Handle returned by xDriverOpen</td>
</tr>
<tr>
<td>ulSize</td>
<td>uint32_t</td>
<td>Size of the passed structure</td>
</tr>
<tr>
<td>pvDriverInfo</td>
<td>void*</td>
<td>Pointer to a DRIVER_INFORMATION structure, to place returned values in.</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lError</td>
<td>int32_t</td>
<td>Error value returned by any driver function</td>
</tr>
<tr>
<td>szBuffer</td>
<td>String</td>
<td>Pointer to a ASCII string buffer, to place returned text in</td>
</tr>
<tr>
<td>ulBufferLen</td>
<td>uint32_t</td>
<td>length of the string buffer for returned data</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

Example:

```c
// 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:**

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

**Arguments:**

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

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 101 is returned. You can use the function `xDriverGetErrorDescription()` to get a description of this error.

**Example:**

```c
int32_t lBoardRet;
int32_t ulBoardIdx = 0;
BOARD_INFORMATION tBoardInfo = (0);
do {
    lBoardRet = xDriverEnumBoards(NULL, ulBoardIdx++, sizeof(tBoardInfo), &tBoardInfo);
    if(lBoardRet == CIFX_NO_ERROR)
        { /* do something */ }
} while(lBoardRet == CIFX_NO_ERROR);
```
### 4.6.6 xDriverEnumChannels

Enumerate all available channels on a board/card.

**Function call:**

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

**Arguments:**

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

**Return Values:**

- CIFX_NO_ERROR if the function succeeds.
- If the function fails, a nonzero error code from chapter *Error Codes* from page 101 is returned. You can use the function `xDriverGetErrorDescription()` to get a description of this error.

**Example:**

```c
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 bootloader, 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:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDriver</td>
<td>CIFXHANDLE</td>
<td>Handle to the driver (returned by xDriverOpen)</td>
</tr>
<tr>
<td>szBuffer</td>
<td>String</td>
<td>Identifier for the Board. (e.g. &quot;cifX&lt;Board Number&gt;&quot;)</td>
</tr>
<tr>
<td>pvData</td>
<td>void*</td>
<td>For further extensions can be NULL</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDriver</td>
<td>CIFXHANDLE</td>
<td>Handle to the driver (returned by xDriverOpen)</td>
</tr>
<tr>
<td>ulBoard</td>
<td>uint32_t</td>
<td>Board number to return pointer for.</td>
</tr>
<tr>
<td>ulCmd</td>
<td>uint32_t</td>
<td>Maps the dual port memory for direct access from an application</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = CIFX_MEM_PTR_OPEN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Map a user specific memory area</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = CIFX_MEM_PTR_USR -&gt; not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Release the dual port pointer (same memory structure MUST be passed)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 = CIFX_MEM_PTR_CLOSE</td>
</tr>
<tr>
<td>pvMemoryInfo</td>
<td>void*</td>
<td>Pointer to returned MEMORY_INFORMATION structure</td>
</tr>
</tbody>
</table>

Note: The Parameter ulChannel must be inserted!

Return Values:

CIFX_NO_ERROR if the function succeeds.
If the function fails, a nonzero error code from chapter *Error Codes* from page 101 is returned. You can use the function `xDriverGetErrorDescription()` to get a description of this error.

Description of the MEMORY_INFORMATION Structure:

<table>
<thead>
<tr>
<th>Value</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pvMemoryID</td>
<td>void*</td>
<td>Identifier of the memory area</td>
</tr>
<tr>
<td>ppvMemoryPtr</td>
<td>void**</td>
<td>Memory pointer</td>
</tr>
<tr>
<td>pulMemorySize</td>
<td>DWORD*</td>
<td>Complete size of the mapped memory</td>
</tr>
<tr>
<td>ulChannel</td>
<td>DWORD*</td>
<td>Requested channel number</td>
</tr>
<tr>
<td>pulChannelStartOffset</td>
<td>DWORD*</td>
<td>Start offset of the requested channel</td>
</tr>
<tr>
<td>pulChannelSize</td>
<td>DWORD*</td>
<td>Memory size of the requested channel</td>
</tr>
</tbody>
</table>
MEMORY_INFORMATION Structure:

```c
typedef __CIFx_PACKED_PRE struct MEMORY_INFORMATION tag
{
    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:

```c
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
        // 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:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hSysdevice</td>
<td>CIFXHANDLE</td>
<td>Handle of the driver</td>
</tr>
<tr>
<td>szBoard</td>
<td>String</td>
<td>Identifier for the Board. Can by cifX&lt;Board Number&gt; or the associated alias.</td>
</tr>
<tr>
<td>phSysdevice</td>
<td>CIFXHANDLE*</td>
<td>Returned handle to the system device, to be used on all other sysdevice functions</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:**

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

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hSysdevice</td>
<td>CIFXHANDLE</td>
<td>Handle of the system device that is to be closed.</td>
</tr>
</tbody>
</table>

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 101 is returned. 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:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hSysdevice</td>
<td>CIFXHANDLE</td>
<td>Handle of the system device</td>
</tr>
<tr>
<td>ulCmd</td>
<td>uint32_t</td>
<td>Available Commands:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = CIFX_INFO_CMD_SYSTEM_INFORMATION</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = CIFX_INFO_CMD_SYSTEM_INFO_BLOCK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 = CIFX_INFO_CMD_SYSTEM_CHANNEL_BLOCK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 = CIFX_INFO_CMD_SYSTEM_CONTROL_BLOCK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 = CIFX_INFO_CMD_SYSTEM_STATUS_BLOCK</td>
</tr>
<tr>
<td>ulSize</td>
<td>uint32_t</td>
<td>Size of the passed system info buffer</td>
</tr>
<tr>
<td>pvSystemInfo</td>
<td>void*</td>
<td>Pointer to SYSTEM_CHANNEL_INFORMATION structure for returned data</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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 could be a software restart or a complete hardware reset.

Usually a software reset is performed.

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

**Function call:**

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

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hSysdevice</td>
<td>CIFXHANDLE</td>
<td>Handle of the system device</td>
</tr>
<tr>
<td>ulTimeout</td>
<td>uint32_t</td>
<td>Timeout in ms to wait for reset to complete</td>
</tr>
</tbody>
</table>

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 101 is returned. You can use the function `xDriverGetErrorDescription()` to get a description of this error.
4.7.5  xSysdeviceBootstart

Perform a boot start on the hardware. This is necessary if the 2nd Stage Bootloader should 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 2nd Stage Bootloader is stored in the FLASH of the hardware.

**Function call:**

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

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hSysdevice</td>
<td>CIFXHANDLE</td>
<td>Handle of the system device</td>
</tr>
<tr>
<td>ulTimeout</td>
<td>uint32_t</td>
<td>Timeout in ms to wait for reset to complete</td>
</tr>
</tbody>
</table>

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 101 is returned. You can use the function `xDriverGetErrorDescription()` to get a description of this error.
4.7.6  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:**

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

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hSysdevice</td>
<td>CIFXHANDLE</td>
<td>Handle of the system device.</td>
</tr>
<tr>
<td>pulRecvPktCount</td>
<td>uint32_t*</td>
<td>Number of packets waiting to be received by Host</td>
</tr>
<tr>
<td>pulSendPktCount</td>
<td>uint32_t*</td>
<td>Number of packets the Host is able to send at once.</td>
</tr>
</tbody>
</table>

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 101 is returned. You can use the function `xDriverGetErrorDescription()` to get a description of this error.
### 4.7.7 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:**

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

**Arguments:**

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

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 101 is returned. You can use the function `xDriverGetErrorDescription()` to get a description of this error.
4.7.8 xSysdeviceGetPacket

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

Function call:

```c
int32_t xSysdeviceGetPacket(CIFXHANDLE hSysdevice, uint32_t ulBufferSize, CIFX_PACKET* ptRecvPacket, uint32_t ulTimeout);
```

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hSysdevice</td>
<td>CIFXHANDLE</td>
<td>Handle of the system device.</td>
</tr>
<tr>
<td>ulBufferSize</td>
<td>uint32_t</td>
<td>Size of the passed receive packet buffer</td>
</tr>
<tr>
<td>ptRecvPacket</td>
<td>CIFX_PACKET*</td>
<td>Buffer to returned packet</td>
</tr>
<tr>
<td>ulTimeout</td>
<td>uint32_t</td>
<td>Time in ms to wait for a receive message. 0 means, do not wait</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. You can use the function xDriverGetErrorDescription() to get a description of this error.
4.7.9  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.

Function call:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel, the download is performed on.</td>
</tr>
<tr>
<td>ulChannel</td>
<td>uint32_t</td>
<td>Number of the channel, to receive the file</td>
</tr>
<tr>
<td>ulMode</td>
<td>uint32_t</td>
<td>Download mode (See DOWNLOAD_MODE_XXX defines)</td>
</tr>
<tr>
<td>szFileName</td>
<td>String</td>
<td>Short file name of the passed data on the device.</td>
</tr>
<tr>
<td>pabFileData</td>
<td>uint8_t*</td>
<td>File data to download.</td>
</tr>
<tr>
<td>ulFileSize</td>
<td>uint32_t</td>
<td>Length of the file in bytes</td>
</tr>
<tr>
<td>pfnCallback</td>
<td>PFN_PROGRESS_CALLBACK</td>
<td>Callback function to indicate the download progress. Passing NULL will suppress callbacks.</td>
</tr>
<tr>
<td>pfnRecvPktCallback</td>
<td>PFN_RECV_PKT_CALLBACK</td>
<td>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.</td>
</tr>
<tr>
<td>pvUser</td>
<td>void*</td>
<td>User parameter which is passed on every callback</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. You can use the function `xDriverGetErrorDescription()` to get a description of this error.
4.7.10  xSysdeviceFindFirstFile

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

Function call:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hSysdevice</td>
<td>CIFXHANDLE</td>
<td>Handle of the system device.</td>
</tr>
<tr>
<td>ulChannel</td>
<td>uint32_t</td>
<td>Channel number</td>
</tr>
<tr>
<td>ptDirectoryInfo</td>
<td>CIFX_DIRECTORYENTRY*</td>
<td>Returned first directory entry. The szFilename entry can be used to start enumerating on a special file. Must be a zero length string to enumerate the whole directory.</td>
</tr>
<tr>
<td>pfnRecvPktCallback</td>
<td>PFN_RECV_PKT_CALLBACK</td>
<td>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.</td>
</tr>
<tr>
<td>pvUser</td>
<td>void*</td>
<td>User parameter which is passed on every callback</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. You can use the function `xDriverGetErrorDescription()` to get a description of this error.
4.7.11  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:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hSysdevice</td>
<td>CIFXHANDLE</td>
<td>Handle of the system device.</td>
</tr>
<tr>
<td>ulChannel</td>
<td>uint32_t</td>
<td>Channel number</td>
</tr>
<tr>
<td>ptDirectoryInfo</td>
<td>CIFX_DIRECTORYENTRY*</td>
<td>Returned directory entry.</td>
</tr>
<tr>
<td>pfnRecvPktCallback</td>
<td>PFN_RECV_PKT_CALLBACK</td>
<td>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.</td>
</tr>
<tr>
<td>pvUser</td>
<td>void*</td>
<td>User parameter which is passed on every callback</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. You can use the function xDriverGetErrorDescription() to get a description of this error.
4.7.12  xSysdeviceUpload

Upload a given file from the device.

Function call:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hSysdevice</td>
<td>CIFXHANDLE</td>
<td>Handle of the system device.</td>
</tr>
<tr>
<td>ulChannel</td>
<td>uint32_t</td>
<td>Channel number of the file</td>
</tr>
<tr>
<td>ulMode</td>
<td>uint32_t</td>
<td>Upload Mode (see DOWNLOAD_MODE_XXX)</td>
</tr>
<tr>
<td>szFilename</td>
<td>char*</td>
<td>Name of the file to upload (must conform to 8.3 filename rules)</td>
</tr>
<tr>
<td>pabFileData</td>
<td>uint8_t*</td>
<td>Buffer to place uploaded data in</td>
</tr>
<tr>
<td>pulFileSize</td>
<td>uint32_t*</td>
<td>[in] Size of the buffer, [out] Number of uploaded bytes</td>
</tr>
<tr>
<td>pfnCallback</td>
<td>PFN_PROGRESS_CALLBACK</td>
<td>Callback function to indicate the download progress.  Passing NULL will suppress callbacks.</td>
</tr>
<tr>
<td>pfnRecvPktCallback</td>
<td>PFN_RECV_PKT_CALLBACK</td>
<td>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.</td>
</tr>
<tr>
<td>pvUser</td>
<td>void*</td>
<td>User parameter which is passed on every callback</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. You can use the function xDriverGetErrorDescription() to get a description of this error.
4.7.13  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 bootloader 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:**

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

**Arguments:**

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

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. You can use the function xDriverGetErrorDescription() to get a description of this error.

**Description of the CIFX_EXTENDED_MEMORY_INFORMATION Structure:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pvMemoryID</td>
<td>void*</td>
<td>Identifier of the memory area</td>
</tr>
<tr>
<td>pvMemoryPtr</td>
<td>void*</td>
<td>Memory pointer to the extended memory area</td>
</tr>
<tr>
<td>ulMemorySize</td>
<td>uint32_t</td>
<td>Size of the extended memory area</td>
</tr>
<tr>
<td>ulMemoryType</td>
<td>uint32_t</td>
<td>Type of the extended memory area (e.g. MRAM)</td>
</tr>
</tbody>
</table>
ulMemoryInformation:

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

**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):

```c
#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:

```c
typedef __CIFx_PACKED_PRE struct CIFX_EXTENDED_MEMORY_INFORMATION_tag {
    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:

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

    printf("--- 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 extended 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:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hDriver</td>
<td>CIFXHANDLE</td>
<td>Handle to the driver (returned by xDriverOpen)</td>
</tr>
<tr>
<td>szBoard</td>
<td>String</td>
<td>Identifier for the Board. Can by cifX&lt;BoardNumber&gt; or the associated alias.</td>
</tr>
<tr>
<td>ulChannel</td>
<td>uint32_t</td>
<td>Channel number to open</td>
</tr>
<tr>
<td>phChannel</td>
<td>CIFXHANDLE*</td>
<td>Returned handle to the channel, to be used on all other channel functions</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel that is to be closed.</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. You can use the function `xDriverGetErrorDescription()` to get a description of this error.
4.8.3  xChannelDownload

Download a file to a communication channel.

Function call:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel, the download is performed on.</td>
</tr>
<tr>
<td>ulMode</td>
<td>uint32_t</td>
<td>Download mode (See DOWNLOAD_MODE_XXX defines)</td>
</tr>
<tr>
<td>szFileName</td>
<td>String</td>
<td>Short file name of the passed data on the device.</td>
</tr>
<tr>
<td>pabFileData</td>
<td>uint8_t*</td>
<td>File data to download.</td>
</tr>
<tr>
<td>ulFileSize</td>
<td>uint32_t</td>
<td>Length of the downloaded file</td>
</tr>
<tr>
<td>pfnCallback</td>
<td>PFN_PROGRESS_CALLBACK</td>
<td>Callback function to indicate the download progress.</td>
</tr>
<tr>
<td>pfnRecvPktCallback</td>
<td>PFN_RECV_PKT_CALLBACK</td>
<td>Callback function to receive unhandled packets during this function.</td>
</tr>
<tr>
<td>pvUser</td>
<td>void*</td>
<td>User parameter which is passed on every callback</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:**

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

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the communication channel.</td>
</tr>
<tr>
<td>ptDirectoryInfo</td>
<td>CIFX_DIRECTORYENTRY*</td>
<td>Returned first directory entry. The szFilename entry can be used to start enumerating on a special file. Must be a zero length string to enumerate the whole directory.</td>
</tr>
<tr>
<td>pfnRecvPktCallback</td>
<td>PFN_RECV_PKT_CALLBACK</td>
<td>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.</td>
</tr>
<tr>
<td>pvUser</td>
<td>void*</td>
<td>User parameter which is passed on every callback</td>
</tr>
</tbody>
</table>

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 101 is returned. You can use the function `xDriverGetErrorDescription()` to get a description of this error.
4.8.5 \textit{xChannelFindNextFile}

Continue enumerating a directory on the channel. This function must be called with a previously returned directory entry structure from \textit{xChannelFindFirstFile}.

\textbf{Function call:}

\begin{verbatim}
int32_t xChannelFindNextFile ( CIFXHANDLE hChannel,  
CIFX_DIRECTORYENTRY* ptDirectoryInfo,  
PFN_RECV_PKT_CALLBACK pfnRecvPktCallback,  
void* pvUser)
\end{verbatim}

\textbf{Arguments:}

\begin{tabular}{|l|l|p{14cm}|}
\hline
Argument & Data type & Description  \\
\hline
hSysdevice & CIFXHANDLE & Handle of the communication channel.  \\
\hline
ptDirectoryInfo & CIFX_DIRECTORYENTRY* & Returned directory entry.  \\
\hline
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.  \\
\hline
pvUser & void* & User parameter which is passed on every callback  \\
\hline
\end{tabular}

\textbf{Return Values:}

\textbf{CIFX\_NO\_ERROR} if the function succeeds.

If the function fails, a nonzero error code from chapter \textit{Error Codes} from page 101 is returned. You can use the function \textit{xDriverGetErrorDescription()} to get a description of this error.
4.8.6 xChannelUpload

Upload a given file from the communication channel.

Function call:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the communication channel.</td>
</tr>
<tr>
<td>ulMode</td>
<td>uint32_t</td>
<td>Upload Mode (see DOWNLOAD_MODE_XXX)</td>
</tr>
<tr>
<td>szFilename</td>
<td>char*</td>
<td>Name of the file to upload (must conform to 8.3 filename rules)</td>
</tr>
<tr>
<td>pabFileData</td>
<td>uint8_t*</td>
<td>Buffer to place uploaded data in</td>
</tr>
<tr>
<td>pulFileSize</td>
<td>uint32_t*</td>
<td>[in] Size of the buffer, [out] Number of uploaded bytes</td>
</tr>
<tr>
<td>pfnCallback</td>
<td>PFN_PROGRESS_CALLBACK</td>
<td>Callback function to indicate the download progress. Passing NULL will suppress callbacks.</td>
</tr>
<tr>
<td>pfnRecvPktCallback</td>
<td>PFN_RECV_PKT_CALLBACK</td>
<td>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.</td>
</tr>
<tr>
<td>pvUser</td>
<td>void*</td>
<td>User parameter which is passed on every callback</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel.</td>
</tr>
<tr>
<td>pulRecvPktCount</td>
<td>uint32_t*</td>
<td>Number of packets waiting to be received by Host</td>
</tr>
<tr>
<td>pulSendPktCount</td>
<td>uint32_t*</td>
<td>Number of packets the Host is able to send at once.</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel.</td>
</tr>
<tr>
<td>ptSendPacket</td>
<td>CIFX_PACKET*</td>
<td>Packet to be send. Total data length is acquired through the uLen element inside the structure.</td>
</tr>
<tr>
<td>ulTimeout</td>
<td>uint32_t</td>
<td>Time in ms to wait for the mailbox to get free.</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel.</td>
</tr>
<tr>
<td>ulBufferSize</td>
<td>uint32_t</td>
<td>Size of the passed receive packet buffer</td>
</tr>
<tr>
<td>ptRecvPacket</td>
<td>CIFX_PACKET*</td>
<td>Buffer to returned packet</td>
</tr>
<tr>
<td>ulTimeout</td>
<td>uint32_t</td>
<td>Time in ms to wait for a receive message. 0 means, do not wait</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel.</td>
</tr>
<tr>
<td>ulBufferSize</td>
<td>uint32_t</td>
<td>Size of the passed packet buffer</td>
</tr>
<tr>
<td>ptRecvPacket</td>
<td>CIFX_PACKET*</td>
<td>Buffer to returned send packet</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:**

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

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel.</td>
</tr>
<tr>
<td>ulResetMode</td>
<td>uint32_t</td>
<td>Type of reset to be performed</td>
</tr>
<tr>
<td>ulTimeout</td>
<td>uint32_t</td>
<td>Time in ms to wait for the channel to be ready again</td>
</tr>
</tbody>
</table>

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 101 is returned. 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:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel.</td>
</tr>
<tr>
<td>ulSize</td>
<td>uint32_t</td>
<td>Length of the passed buffer.</td>
</tr>
<tr>
<td>pvChannelInfo</td>
<td>void*</td>
<td>Pointer to a CHANNEL_INFORMATION structure, for returned data</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.
If the function fails, a nonzero error code from chapter *Error Codes* from page 101 is returned. 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:**

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

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel.</td>
</tr>
</tbody>
</table>
| ulCmd       | DWORD       | Watchdog Command
|             |             | Start and trigger the watchdog monitoring                                  |
|             |             | 1 = CIFX_WATCHDOG_START                                                     |
|             |             | Stop the watchdog monitoring                                                |
|             |             | 0 = CIFX_WATCHDOG_STOP                                                      |
| pulTrigger  | uint32_t*   | Last trigger value from dual port                                           |

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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 presence 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:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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. 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' it will not activate the bus communication until it receives a BUS_ON state in its handshake flags.

**Function call:**

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

**Arguments:**

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

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. You can use the function `xDriverGetErrorDescription()` to get a description of this error.
4.8.18  xChannelControlBlock

Reading / writing the communication channel control block.

Function call:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel.</td>
</tr>
<tr>
<td>ulCmd</td>
<td>uint32_t</td>
<td>Control block commands:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read the block area</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = CIFX_CMD_READ_DATA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write the block area</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = CIFX_CMD_WRITE_DATA</td>
</tr>
<tr>
<td>ulOffset</td>
<td>uint32_t*</td>
<td>Start offset in the block area</td>
</tr>
<tr>
<td>ulDataLen</td>
<td>uint32_t</td>
<td>Number of bytes to read</td>
</tr>
<tr>
<td>pvData</td>
<td>void*</td>
<td>User buffer</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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

**Function call:**

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

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel.</td>
</tr>
<tr>
<td>ulCmd</td>
<td>uint32_t</td>
<td>Status block commands:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read the block area</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = CIFX_CMD_READ_DATA</td>
</tr>
<tr>
<td>ulOffset</td>
<td>uint32_t*</td>
<td>Start offset in the block area</td>
</tr>
<tr>
<td>ulDataLen</td>
<td>uint32_t</td>
<td>Number of bytes to read</td>
</tr>
<tr>
<td>pvData</td>
<td>void*</td>
<td>User buffer</td>
</tr>
</tbody>
</table>

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 101 is returned. 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:**

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

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel.</td>
</tr>
<tr>
<td>ulCmd</td>
<td>uint32_t</td>
<td>Extended status block commands:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read the block area</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = CIFX_CMD_READ_DATA</td>
</tr>
<tr>
<td>ulOffset</td>
<td>uint32_t*</td>
<td>Start offset in the block area</td>
</tr>
<tr>
<td>ulDataLen</td>
<td>uint32_t</td>
<td>Number of bytes to read</td>
</tr>
<tr>
<td>pvData</td>
<td>void*</td>
<td>User buffer</td>
</tr>
</tbody>
</table>

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 101 is returned. You can use the function `xDriverGetErrorDescription()` to get a description of this error.
4.8.21 xChannelUserBlock

not implemented yet!
4.8.22 xChannelIORead

Instructs the channel to refresh the input data with actual fieldbus data and reads the Input process data image of the channel.

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. You can use the function xDriverGetErrorDescription() to get a description of this error.
4.8.23  xChannelIOWrite

Writes the Output process data image of the channel and commands the card to send it to the field bus.

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. You can use the function `xDriverGetErrorDescription()` to get a description of this error.
4.8.24  xChannelIOReadSendData

Reads the last output data image from the host.

Function call:

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel.</td>
</tr>
<tr>
<td>ulAreaNumber</td>
<td>uint32_t</td>
<td>Number of the I/O Output area to get data from</td>
</tr>
<tr>
<td>ulOffset</td>
<td>uint32_t</td>
<td>Offset inside area to start reading data from</td>
</tr>
<tr>
<td>ulDataLen</td>
<td>uint32_t</td>
<td>Length of the data being received</td>
</tr>
<tr>
<td>pvData</td>
<td>void*</td>
<td>Pointer to the returned data buffer</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. You can use the function xDriverGetErrorDescription() to get a description of this error.
4.8.25 PLC I/O Image Functions

Some of the PLC programs (Programmable Logic Controller also known as SoftPLCs) are using an own process data image. In such cases the standard \texttt{xChannelIORead/xChannelIOWrite()} functions would process an additional copy function from/to the PLC buffer into the cards I/O image DPM location.

The following PLC functions are design to save this extra copy handling by splitting the combined \texttt{xChannelIORead()/xChannelIOWrite()} functions into three separate functions to be able to control the access to the cards I/O data image.

Important for the use of the functions is a prior call to the \texttt{xChannelPLCMemoryPtr()} function. This will deliver the necessary pointers to the requested I/O data image.

\begin{center}
\textbf{Note:} If the PLC functions are used, the application is responsible to synchronize the data access between the host and the communication channel.
\end{center}

4.8.25.1 \texttt{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 \texttt{xChannelIORead()} or \texttt{xChannelIOWrite()}.

\begin{center}
\textbf{Function call:}
\end{center}

\begin{center}
\begin{verbatim}
int32_t xChannelPLCMemoryPtr( CIFXHANDLE hChannel,
                              uint32_t ulCmd,
                              void* pvMemoryInfo)
\end{verbatim}
\end{center}

\begin{center}
\textbf{Arguments:}
\end{center}

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

\begin{center}
\textbf{Return Values:}
\end{center}

\texttt{CIFX_NO_ERROR} if the function succeeds.

If the function fails, a nonzero error code from chapter \textit{Error Codes} from page 101 is returned. You can use the function \texttt{xDriverGetErrorDescription()} to get a description of this error.
Description of the PLC_MEMORY_INFORMATION Structure:

<table>
<thead>
<tr>
<th>Value</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pvMemoryID</td>
<td>void*</td>
<td>Identifier of the memory area</td>
</tr>
<tr>
<td>ppvMemoryPtr</td>
<td>void**</td>
<td>Memory pointer</td>
</tr>
<tr>
<td>ulAreaDefinition</td>
<td>uint32_t</td>
<td>Input / Output area</td>
</tr>
<tr>
<td>ulAreaNumber</td>
<td>uint32_t</td>
<td>Area number (0..1)</td>
</tr>
<tr>
<td>pulIOAreaStartOffset</td>
<td>uint32_t*</td>
<td>Buffer to store the I/O area start offset</td>
</tr>
<tr>
<td>pulAreaSize</td>
<td>uint32_t*</td>
<td>Buffer to store the size of the I/O area</td>
</tr>
</tbody>
</table>

PLC_MEMORY_INFORMATION Structure:

```c
#ifndef PLC_MEMORY_INFORMATION
#define PLC_MEMORY_INFORMATION  
/* PLC Memory Information structure */
typedef __CIFx_PACKED_PRE struct PLC_MEMORY_INFORMATION
{
  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;
#endif
```

Example:

```c
void TestPLCFunctions( void)
{
  unsigned char abBuffer[1000] = {0};
  uint32_t ulState = 0;
  printf("--- Test PLC functions ---
  ");
  long lRet = CIFX_NO_ERROR;
  /* Open channel */
  HANDLE hDevice = NULL;
  lRet = xChannelOpen(NULL, "CIFx0", 0, &hDevice);
  if(lRet != CIFX_NO_ERROR)
    ShowError(lRet);
  else
    /* Start PLC functions */
    unsigned char* pabDMMemory = 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**)&pabDMMemory; // 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

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

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

/* 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 */
if( CIFX_NO_ERROR != CIFX_HOST_STATE_READY, &ulState, 100); if( CIFX_NO_ERROR != lRet)
{
    ShowError(lRet);
 }

/* Wait until BUS is up and running */
printf("\nWait until BUS communication is available!\n\n");
do
{
    if( CIFX_NO_ERROR != CIFX_HOST_STATE_READY, &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("\nBUS is ON!\n");
    break;
    }
    while ( --ulWaitBusCount > 0);

    /*----------------------*/
    /* Start cyclic data IO */
    /*----------------------*/
    if( CIFX_NO_ERROR == lRet)
    {
        printf("\n Press any key to stop \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 */
            /*------------------------*/
            if( CIFX_NO_ERROR == lRet)
            {
                printf("\n Press any key to stop \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 */
                    /*------------------------*/
                    if( CIFX_NO_ERROR == lRet)
                    {
ShowError( lRet);
} else if( 1 == ulReadState)
{
    /* It is allowed to read the image */
    /* Read 100 Bytes */
    memcpy( abBuffer, pabDMEMemory, 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 */
    pabDMEMemory_OUT[0]++;
    pabDMEMemory_OUT[1] = abBuffer[1];

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

/* clean keyboard buffer */
_getch();

/* Return the DPM memory pointer */
if ( NULL != pabDMEMemory)
{
    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 != pabDMEMemory_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:** If this function is called multiple times while the actual state is ‘not ready’ (use the corresponding `xChannelPLCIs......Ready()` function), the result is unpredictable.

**Function call:**

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

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel.</td>
</tr>
<tr>
<td>ulAreaNumber</td>
<td>uint32_t</td>
<td>Number of the I/O area to request a input data refresh</td>
</tr>
</tbody>
</table>

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 101 is returned. 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:** If this function is called multiple times while the actual state is 'not ready' (use the corresponding xChannelPLCIs......Ready() function), the result is unpredictable.

**Function call:**

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

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel.</td>
</tr>
<tr>
<td>ulAreaNumber</td>
<td>uint32_t</td>
<td>Number of the I/O area to request a output data refresh</td>
</tr>
</tbody>
</table>

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel.</td>
</tr>
<tr>
<td>ulCmd</td>
<td>uint32_t</td>
<td>DMA State Commands:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disable DMA mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = CIFX_DMA_STATE_OFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Enable DMA mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = CIFX_DMA_STATE_ON</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Get actual DMA state</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = CIFX_DMA_STATE_GETSTATE</td>
</tr>
<tr>
<td>pulState</td>
<td>uint32_t*</td>
<td>Actual state returned</td>
</tr>
</tbody>
</table>

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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

**Note:** Handling of CIFX_NOTIFY_PD0_IN / CIFX_NOTIFY_PD0_OUT and CIFX_NOTIFY_PD1_IN / CIFX_NOTIFY_PD1_OUT depends on the so called “I/O Exchange Mode” configured on the device.

Handling of the CIFX_NOTIFY_SYNC events depends on the so called “Synchronization Modes” configured on the device.

The modes are defining how the device handshake flags are driven (by application or by the device).

The callback functions are called if the driver detects a change in the corresponding handshake flags.

How the application processes the events is part of the application development and must correspond to the configured mode settings of the device.

Handshake modes are described in [1].
Determining the Configured I/O Handshake Mode:
The configured I/O handshake mode (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". Currently the following synchronization modes are defined.

```c
/* Block definition: I/O Mode */
#define RCX_IO_MODE_DEFAULT        0x0000    /*!< I/O mode default, for compatibility 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 */
```

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.

```c
/* 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: Handshake and synchronization modes must be supported by the firmware. Consult the corresponding fieldbus "Protocol API" manual to verify which modes are supported.
4.8.27.1 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.

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

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ulNotification</td>
<td>uint32_t</td>
<td>Occurred event</td>
</tr>
<tr>
<td>ulDataLen</td>
<td>uint32_t</td>
<td>Length of additional data</td>
</tr>
<tr>
<td>pvData</td>
<td>void*</td>
<td>Additional Data (depends on ulNotification)</td>
</tr>
<tr>
<td>pvUser</td>
<td>void*</td>
<td>User parameter from registration</td>
</tr>
</tbody>
</table>

Possible Notification Events:

<table>
<thead>
<tr>
<th>ulNotification</th>
<th>Passed Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFX_NOTIFY_RX_MBX_FULL</td>
<td>Pointer to CIFX_NOTIFY_RX_MBX_FULL_DATA_T structure which contains the total number of available packets</td>
<td>Packet is available in receive mailbox</td>
</tr>
<tr>
<td>CIFX_NOTIFY_TX_MBX_EMPTY</td>
<td>Pointer to CIFX_NOTIFY_TX_MBX_EMPTY_DATA_T structure which contains the maximum amount of packets that can be send by host</td>
<td>Send mailbox is empty</td>
</tr>
<tr>
<td>CIFX_NOTIFY_PD0_IN</td>
<td>none</td>
<td>Input area 0 has been processed</td>
</tr>
<tr>
<td>CIFX_NOTIFY_PD1_IN</td>
<td>none</td>
<td>Input area 1 has been processed</td>
</tr>
<tr>
<td>CIFX_NOTIFY_PD0_OUT</td>
<td>none</td>
<td>Output area 0 has been processed</td>
</tr>
<tr>
<td>CIFX_NOTIFY_PD1_OUT</td>
<td>none</td>
<td>Output area 1 has been processed</td>
</tr>
<tr>
<td>CIFX_NOTIFY_SYNC</td>
<td>none</td>
<td>Bus-Synchronous notification</td>
</tr>
</tbody>
</table>
4.8.27.2 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:**

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

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel.</td>
</tr>
<tr>
<td>ulNotification</td>
<td>uint32_t</td>
<td>Possible Notification:</td>
</tr>
<tr>
<td>pfnCallback</td>
<td>PFN_NOTIFY_CALLBACK</td>
<td>Function to be called if event occurs</td>
</tr>
<tr>
<td>pvUser</td>
<td>void*</td>
<td>Parameter passed to callback</td>
</tr>
</tbody>
</table>

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. You can use the function xDriverGetErrorDescription() to get a description of this error.
### 4.8.27.3 xChannelUnregisterNotification

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

**Function call:**

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

**Arguments:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hChannel</td>
<td>CIFXHANDLE</td>
<td>Handle of the channel.</td>
</tr>
<tr>
<td>ulNotification</td>
<td>uint32_t</td>
<td>Possible Notification:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = CIFX_NOTIFY_RX_MBX_FULL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = CIFX_NOTIFY_TX_MBX_EMPTY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 = CIFX_NOTIFY_PD0_IN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 = CIFX_NOTIFY_PD1_IN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 = CIFX_NOTIFY_PD0_OUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 = CIFX_NOTIFY_PD1_OUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 = CIFX_NOTIFY_SYNC</td>
</tr>
</tbody>
</table>

**Return Values:**

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 101 is returned. 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.

```c
/* 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) 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:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter Error Codes from page 101 is returned. 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

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    HANDLE 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

```c
/*!
 * Function to demonstrate system device functionality (Packet Transfer)
 * \return CIFX_NO_ERROR on success
 */
int32_t SysdeviceDemo(HANDLE hDriver, char* szBoard)
{
    int32_t lRet = CIFX_NO_ERROR;
    HANDLE 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\n", 
   ulSendPktCount, ulRecvPktCount);

if(CIFX_NO_ERROR != (lRet = xSysdevicePutPacket(hSys, 
   &tSendPkt, 
   PACKET_WAIT_TIMEOUT)))
{
   printf("Error sending packet to device!\n");
} else
{
   printf("Send Packet:\n");
   DumpPacket(&tSendPkt);
   xSysdeviceGetMBXState(hSys, &ulRecvPktCount, &ulSendPktCount);
   printf("System Mailbox State: MaxSend = %u, Pending Receive = %u\n", 
   ulSendPktCount, ulRecvPktCount);

   if(CIFX_NO_ERROR != (lRet = xSysdeviceGetPacket(hSys, 
      sizeof(tRecvPkt), 
      &tRecvPkt, 
      PACKET_WAIT_TIMEOUT)) )
   {
      printf("Error getting packet from device!\n");
   } else
   {
      printf("Received Packet:\n");
      DumpPacket(&tRecvPkt);
      xSysdeviceGetMBXState(hSys, &ulRecvPktCount, &ulSendPktCount);
      printf("System Mailbox State: MaxSend = %u, Pending Receive = %u\n", 
      ulSendPktCount, ulRecvPktCount);
   }
}
/* Close the system device */
xSysdeviceClose(hSys);

printf(" State = 0x%08X\n", lRet);
printf("----------------------------------------\n");
return lRet;
5.3 Communication Channel Example

```c
/*! Function to demonstrate communication channel functionality 
* Packet Transfer and I/O Data exchange 
* \return CIFX_NO_ERROR on success */
/*****************************************************************************/
int32_t ChannelDemo(HANDLE hDriver, char* szBoard, uint32_t ulChannel)
{
    HANDLE 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!\n");
    } 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 cycle */
        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 field 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 field 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>%s</s><\r\n", lRet, szBuffer);
} else
{
    printf(" State = 0x%08X<\r\n", lRet);
}
printf("----------------------------------------------------\r\n");
return lRet;
5.4 Board and Channel Enumeration

```c
/* **************************************************************************/ /*! Function to demonstrate the board/channel enumeration */ /* \return CIFX_NO_ERROR on success */inkle**************************************************************************/ void EnumBoardDemo(HANDLE 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 = 1000ms
- 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: 5000ms
General Protocol Stack Handling

=> 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 De-initialization:

- xDriverClose()  Close the cifX Driver
6.2 Protocol Stack Configuration

**Master Stack Configuration - via rcX Packet:**
- `xCIFX_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:**
- `xCIFX_CHANNELINIT()` **Activate actual configuration**
  - Maximum Timeout: 10000ms
- `DOWNLOAD_CONFIGURATION...` **Download a database**

**Slave Stack Configuration: via rcX Packets:**
- Send configuration data
- `xCIFX_CHANNELINIT()` **Activate the configuration**
  - Maximum Timeout: 10000ms
- **Optional:** Set watchdog time (RCX_SET_WATCHDOG_TIME_REQ) via `xCIFX_CHANNELINIT()`
  - Standard Put/GetPacket() Timeout: 1000ms

**Slave Stack Configuration - via Database:**
- `DOWNLOAD_CONFIGURATION...` **Download a database**
- `xCIFX_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:
- `xChannelPLCWriteReady()` / `xChannelPLCReadReady()` **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**
# Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Symbol / Description</th>
</tr>
</thead>
</table>
| 0x00000000 | CIFX_NO_ERROR  
No error  |
<p>| 0x800Axxxx | General Error Codes                                      |</p>
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Symbol / Description</th>
</tr>
</thead>
</table>
| 0x800A0001 | CIFX_INVALID_POINTER  
Invalid pointer (e.g. NULL was passed to the function) |
| 0x800A0002 | CIFX_INVALID_BOARD  
No board with the given name / index available |
| 0x800A0003 | CIFX_INVALID_CHANNEL  
No channel with the given index available |
| 0x800A0004 | CIFX_INVALID_HANDLE  
An invalid handle was passed to the function |
| 0x800A0005 | CIFX_INVALID_PARAMETER  
Invalid parameter passed to the function |
| 0x800A0006 | CIFX_INVALID_COMMAND  
Command parameter is invalid |
| 0x800A0007 | CIFX_INVALID_BUFFERSIZE  
The supplied buffer does not match the expected size |
| 0x800A0008 | CIFX_INVALID_ACCESS_SIZE  
Invalid access size (e.g. I/O area size is exceeded by given offset and length) |
| 0x800A0009 | CIFX_FUNCTION_FAILED  
Generic function failure |
| 0x800A000A | CIFX_FILE_OPEN_FAILED  
A file could not be opened |
| 0x800A000B | CIFX_FILE_SIZE_ZERO  
File size is zero |
| 0x800A000C | CIFX_FILE_LOAD_INSUFF_MEM  
Insufficient memory to load file a file to RAM |
| 0x800A000D | CIFX_FILE_READ_ERROR  
Error reading file data |
| 0x800A000F | CIFX_FILE_TYPE_INVALID  
The given file is invalid for the operation |
| 0x800A0010 | CIFX_FILE_NAME_INVALID  
Invalid filename given |
| 0x800A0011 | CIFX_FUNCTION_NOT_AVAILABLE  
Function is not available on the driver |
| 0x800A0012 | CIFX_BUFFER_TOO_SHORT  
The passed buffer is too short to receive all of the requested data |
### General Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Symbol / Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x800A0013</td>
<td>CIFX_MEMORY_MAPPING_FAILED</td>
</tr>
<tr>
<td>0x800A0014</td>
<td>CIFX_NO_MORE_ENTRIES</td>
</tr>
<tr>
<td>0x800A0015</td>
<td>CIFX_CALLBACK_MODE_UNKNOWN</td>
</tr>
<tr>
<td>0x800A0016</td>
<td>CIFX_CALLBACK_CREATE_EVENT_FAILED</td>
</tr>
<tr>
<td>0x800A0017</td>
<td>CIFX_CALLBACK_CREATE_RECV_BUFFER</td>
</tr>
<tr>
<td>0x800A0018</td>
<td>CIFX_CALLBACK_ALREADY_USED</td>
</tr>
<tr>
<td>0x800A0019</td>
<td>CIFX_CALLBACK_NOT_REGISTERED</td>
</tr>
<tr>
<td>0x800A001A</td>
<td>CIFX_INTERRUPT_DISABLED</td>
</tr>
</tbody>
</table>

*Table 18: General Error Codes*

### Driver Related Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Symbol / Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x800B001</td>
<td>CIFX_DRV_NOT_INITIALIZED</td>
</tr>
<tr>
<td>0x800B002</td>
<td>CIFX_DRV_INIT_STATE_ERROR</td>
</tr>
<tr>
<td>0x800B003</td>
<td>CIFX_DRV_READ_STATE_ERROR</td>
</tr>
<tr>
<td>0x800B004</td>
<td>CIFX_DRV_CMD_ACTIVE</td>
</tr>
<tr>
<td>0x800B005</td>
<td>CIFX_DRV_DOWNLOAD_FAILED</td>
</tr>
<tr>
<td>0x800B006</td>
<td>CIFX_DRV_WRONG_DRIVER_VERSION</td>
</tr>
<tr>
<td>0x800B007</td>
<td>CIFX_DRV_DRIVER_NOT_LOADED</td>
</tr>
<tr>
<td>0x800B008</td>
<td>CIFX_DRV_CHANNEL_NOT_INITIALIZED</td>
</tr>
</tbody>
</table>

*Table 19: Driver Related Error Codes*
### Error Codes 103/107

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Symbol / Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFX_DRV_IO_CONTROL_FAILED</td>
<td>Function call into the driver failed (e.g. used by the Windows API DLL to signal problems in IO-Control driver calls)</td>
</tr>
<tr>
<td>CIFX_DRV_NOT_OPENED</td>
<td>Driver was not opened by calling xDriverOpen()</td>
</tr>
</tbody>
</table>

**Table 19: Driver Related Error Codes**

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Symbol / Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFX_DEV_DPM_ACCESS_ERROR</td>
<td>Dual port memory not accessible (e.g. board not found, wrong dual port memory content)</td>
</tr>
<tr>
<td>CIFX_DEV_NOT_READY</td>
<td>Device is not ready (NSF_READY or NCF_READY flag is not set) The system device or communication channel is not working</td>
</tr>
<tr>
<td>CIFX_DEV_NOT_RUNNING</td>
<td>Device is not running (NCF_RUNNING flag is not set). The communication channel is not configured</td>
</tr>
<tr>
<td>CIFX_DEV_WATCHDOG_FAILED</td>
<td>Watchdog test failed</td>
</tr>
<tr>
<td>CIFX_DEV_SYSERR</td>
<td>Error in handshake flags</td>
</tr>
<tr>
<td>CIFX_DEV_MAILBOX_FULL</td>
<td>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. (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)</td>
</tr>
<tr>
<td>CIFX_DEV_PUT_TIMEOUT</td>
<td>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. (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)</td>
</tr>
<tr>
<td>CIFX_DEV_GET_TIMEOUT</td>
<td>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. (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)</td>
</tr>
<tr>
<td>CIFX_DEV_GET_NO_PACKET</td>
<td>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. (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)</td>
</tr>
<tr>
<td>Error Code</td>
<td>Symbol / Description</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x800C001A</td>
<td>CIFX_DEV_MAILBOX_TOO_SHORT</td>
</tr>
<tr>
<td></td>
<td>Mailbox is too short for the given packet. The packet send by PutPacket() does not fit into the mailbox.</td>
</tr>
<tr>
<td>0x800C0020</td>
<td>CIFX_DEV_RESET_TIMEOUT</td>
</tr>
<tr>
<td></td>
<td>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).</td>
</tr>
<tr>
<td>0x800C0021</td>
<td>CIFX_DEV_NO_COM_FLAG</td>
</tr>
<tr>
<td></td>
<td>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).</td>
</tr>
<tr>
<td>0x800C0022</td>
<td>CIFX_DEV_EXCHANGE_FAILED</td>
</tr>
<tr>
<td></td>
<td>I/O data exchange failed. Function xChannelIORead() or xChannelIOWrite() fails, because the device does not allow to access the I/O data image. (NCF_PDIN / NCF_PDOUT flags are not in the state allowing access to the I/O process data image)</td>
</tr>
<tr>
<td>0x800C0023</td>
<td>CIFX_DEV_EXCHANGE_TIMEOUT</td>
</tr>
<tr>
<td></td>
<td>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. (NCF_PDIN / NCF_PDOUT flags are not in the state allowing access to the I/O process data image)</td>
</tr>
<tr>
<td>0x800C0024</td>
<td>CIFX_DEV_COM_MODE_UNKNOWN</td>
</tr>
<tr>
<td></td>
<td>Unknown I/O data exchange mode (mode is not within 0..5)</td>
</tr>
<tr>
<td>0x800C0025</td>
<td>CIFX_DEV_FUNCTION_FAILED</td>
</tr>
<tr>
<td></td>
<td>Device function failed</td>
</tr>
<tr>
<td>0x800C0026</td>
<td>CIFX_DEV_DPM_SIZE_MISMATCH</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>0x800C0027</td>
<td>CIFX_DEV_STATE_MODE_UNKNOWN</td>
</tr>
<tr>
<td></td>
<td>Unknown state mode</td>
</tr>
<tr>
<td>0x800C0028</td>
<td>CIFX_DEV_HW_PORT_IS_USED</td>
</tr>
<tr>
<td></td>
<td>Device is accessed either by another application or another instance.</td>
</tr>
<tr>
<td></td>
<td>- Driver / device can't be unloaded, open connection to the system device or a communication channels still active</td>
</tr>
<tr>
<td></td>
<td>- xChannelOpen() can't be executed because it is currently used by another application</td>
</tr>
<tr>
<td>0x800C0029</td>
<td>CIFX_DEV_CONFIG_LOCK_TIMEOUT</td>
</tr>
<tr>
<td></td>
<td>Failed lock the communication channels configuration within the given time. xChannelConfigLock() wait time expired (RCX_COMM_COS_CONFIG_LOCKED flag not set).</td>
</tr>
<tr>
<td>0x800C002A</td>
<td>CIFX_DEV_CONFIG_UNLOCK_TIMEOUT</td>
</tr>
<tr>
<td></td>
<td>Failed to unlock the communication channel configuration within the given time. xChannelConfigLock() wait time expired (RCX_COMM_COS_CONFIG_LOCKED flag not cleared)</td>
</tr>
<tr>
<td>0x800C002B</td>
<td>CIFX_DEV_HOST_STATE_SET_TIMEOUT</td>
</tr>
<tr>
<td></td>
<td>Wait time expires during xChannelHostState() without reaching CIFX_HOST_STATE_READY.</td>
</tr>
<tr>
<td></td>
<td>(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)</td>
</tr>
<tr>
<td>0x800C002C</td>
<td>CIFX_DEV_HOST_STATE_CLEAR_TIMEOUT</td>
</tr>
<tr>
<td></td>
<td>Wait time expires during xChannelHostState() without reaching CIFX_HOST_STATE_NOT_READY.</td>
</tr>
<tr>
<td></td>
<td>(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)</td>
</tr>
<tr>
<td>Error Code</td>
<td>Symbol / Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x800C002D</td>
<td>CIFX_DEV_INITIALIZATION_TIMEOUT</td>
</tr>
<tr>
<td>0x800C002E</td>
<td>CIFX_DEV_BUS_STATE_ON_TIMEOUT</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0x800C002F</td>
<td>CIFX_DEV_BUS_STATE_OFF_TIMEOUT</td>
</tr>
<tr>
<td>0x800C0040</td>
<td>CIFX_DEV_MODULE_ALREADY_RUNNING</td>
</tr>
<tr>
<td>0x800C0041</td>
<td>CIFX_DEV_MODULE_ALREADY_EXISTS</td>
</tr>
<tr>
<td>0x800C0050</td>
<td>CIFX_DEV_DMA_INSUFF_BUFFER_COUNT</td>
</tr>
<tr>
<td>0x800C0051</td>
<td>CIFX_DEV_DMA_BUFFER_TOO_SMALL</td>
</tr>
<tr>
<td>0x800C0052</td>
<td>CIFX_DEV_DMA_BUFFER_TOO_BIG</td>
</tr>
<tr>
<td>0x800C0053</td>
<td>CIFX_DEV_DMA_BUFFER_NOT_ALIGNED</td>
</tr>
<tr>
<td>0x800C0054</td>
<td>CIFX_DEV_DMA_HANSHAKEMODE_NOT_SUPPORTED</td>
</tr>
<tr>
<td>0x800C0055</td>
<td>CIFX_DEV_DMA_IO_AREA_NOT_SUPPORTED</td>
</tr>
<tr>
<td>0x800C0056</td>
<td>CIFX_DEV_DMA_STATE_ON_TIMEOUT</td>
</tr>
<tr>
<td>0x800C0057</td>
<td>CIFX_DEV_DMA_STATE_OFF_TIMEOUT</td>
</tr>
<tr>
<td>0x800C0058</td>
<td>CIFX_DEV_SYNC_STATE_INVALID_MODE</td>
</tr>
<tr>
<td>0x800C0059</td>
<td>CIFX_DEV_SYNC_STATE_TIMEOUT</td>
</tr>
</tbody>
</table>
8 Appendix

8.1 List of Tables
Table 1: List of Revisions ................................................................. 4
Table 2: Terms, Abbreviations and Definitions ........................................ 5
Table 3: References to Documents ........................................................ 5
Table 4: List of API Functions – Driver Functions .......................... 8
Table 5: List of API Functions – System Device Functions .................. 8
Table 6: List of API Functions – Communication Channel Functions .... 10
Table 7: Driver Functions .................................................................. 14
Table 8: System Device Functions ..................................................... 15
Table 9: Communication Channel Functions ...................................... 17
Table 10: Driver Information Structure .............................................. 18
Table 11: Board Information Structure ............................................. 18
Table 12: System Channel Information ............................................. 19
Table 13: System Channel Info Block ............................................... 19
Table 14: System Channel - Channel Info Block ................................ 20
Table 15: System Channel Control Block ........................................... 20
Table 16: System Channel Status Block ........................................... 20
Table 17: Channel Information Structure ........................................... 21
Table 18: General Error Codes ............................................................ 102
Table 19: Driver Related Error Codes ................................................. 103
Table 20: Device / Communication Related Error Codes .................... 105

8.2 List of Figures
Figure 1: CIFX API Components .......................................................... 6
Figure 2: Component Overview for netX Applications .......................... 7
8.3 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.
New Delhi - 110 065
Phone: +91 11 43055431
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.
Suwon, Gyeonggi, 443-734
Phone: +82 (0) 31-695-5515
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