



Driver Manual
cifX Device Driver
Windows 2000/XP/Vista/7
V1.1.x.x

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC060701DRV21EN | Revision 21 | English | 2011-12 | Released | Public

Table of Contents

1	Introduction.....	4
1.1	About this Document.....	4
1.2	List of Revisions.....	5
1.3	Terms, Abbreviations and Definitions.....	6
1.4	References.....	6
1.5	Legal Notes.....	7
	1.5.1 Copyright.....	7
	1.5.2 Important Notes.....	7
	1.5.3 Exclusion of Liability.....	8
	1.5.4 Export.....	8
2	Windows 2000, XP, Vista and Windows 7.....	9
2.1	Overview.....	9
2.2	Features.....	10
2.3	Limitations.....	10
2.4	Windows Operating System Timing Behaviour.....	11
2.5	Installation.....	13
2.6	Driver Setup and Test Program.....	14
2.7	Device Time Setting if Device Supports Time Handling.....	16
3	General Overview.....	17
3.1	Data Handling.....	17
3.2	Devices, Channels and Driver Functions.....	18
3.3	Timer Resolution.....	19
4	Simple C-Application Example.....	20
4.1	The Main() Function.....	20
4.2	System Device Example.....	21
4.3	Communication Channel Example.....	23
4.4	Board and Channel Enumeration.....	27
5	CIFX API (Application Programming Interface).....	28
5.1	Header Files.....	28
5.2	Driver Related Functions.....	28
5.3	System Device Related Functions.....	29
5.4	Communication Channel Related Functions.....	30
5.5	Structure definitions.....	32
	5.5.1 Driver Information.....	32
	5.5.2 Board Information.....	32
	5.5.3 System Channel Information.....	33
	5.5.4 Communication Channel Information.....	35
5.6	Driver Related Functions.....	36
	5.6.1 xDriverOpen.....	36
	5.6.2 xDriverClose.....	37
	5.6.3 xDriverGetInformation.....	38
	5.6.4 xDriverGetErrorDescription.....	39
	5.6.5 xDriverEnumBoards.....	40
	5.6.6 xDriverEnumChannels.....	41
	5.6.7 xDriverRestartDevice.....	42
	5.6.8 xDriverMemoryPointer.....	43
5.7	System Device Specific Functions.....	45
	5.7.1 xSysdeviceOpen.....	45
	5.7.2 xSysdeviceClose.....	46
	5.7.3 xSysdeviceInfo.....	47
	5.7.4 xSysdeviceReset.....	48
	5.7.5 xSysdeviceBootstart.....	49
	5.7.6 xSysdeviceGetMBXState.....	50
	5.7.7 xSysdevicePutPacket.....	51
	5.7.8 xSysdeviceGetPacket.....	52
	5.7.9 xSysdeviceDownload.....	53
	5.7.10 xSysdeviceFindFirstFile.....	54
	5.7.11 xSysdeviceFindNextFile.....	55
	5.7.12 xSysdeviceUpload.....	56

	5.7.13 xSysdeviceExtendedMemory	57
5.8	Channel Specific Functions.....	61
	5.8.1 xChannelOpen	61
	5.8.2 xChannelClose	62
	5.8.3 xChannelDownload	63
	5.8.4 xChannelFindFirstFile.....	64
	5.8.5 xChannelFindNextFile	65
	5.8.6 xChannelUpload.....	66
	5.8.7 xChannelGetMBXState	67
	5.8.8 xChannelPutPacket	68
	5.8.9 xChannelGetPacket	69
	5.8.10 xChannelGetSendPacket	70
	5.8.11 xChannelReset.....	71
	5.8.12 xChannelInfo	72
	5.8.13 xChannelIOInfo	73
	5.8.14 xChannelWatchdog	74
	5.8.15 xChannelConfigLock	75
	5.8.16 xChannelHostState	76
	5.8.17 xChannelBusState.....	77
	5.8.18 xChannelControlBlock.....	78
	5.8.19 xChannelCommonStatusBlock	79
	5.8.20 xChannelExtendedStatusBlock	80
	5.8.21 xChannelUserBlock	81
	5.8.22 xChannelIORead.....	82
	5.8.23 xChannelIOWrite	83
	5.8.24 xChannelIOReadSendData	84
	5.8.25 PLC I/O Image Functions	85
	5.8.26 DMA Functions	93
	5.8.27 Notification Functions	94
	5.8.28 Bus-Synchronous Operations.....	98
6	Error Codes.....	100
7	Appendix	103
	7.1 List of Tables	103
	7.2 List of Figures.....	103
	7.3 Contacts	104

1 Introduction

1.1 About this Document

This manual describes the cifX device driver for the Microsoft Windows desktop operating systems Windows 2000, Windows XP, Windows Vista and Windows 7.

Both versions of the cifX driver are offering the same functionality and also the same application programming interface (API) to access a netX based hardware (e.g. cifX, comX boards and the netX chip).

In general, the drivers are supporting various netX based hardware designs described under *Requirements* for each of the driver.

The API (CIFX API) is designed to give the user an easy access to all of the communication board functionalities. This manual also includes a detailed description of the CIFX API functions.

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 (dual-port memory) definitions including the CIFX API functions. The toolkit is described in a separate manual *cifX/netX Toolkit*.

1.2 List of Revisions

Rev	Date	Name	Chapter	Revision
13	2009-03-19	MT		Update for Windows CE 6.0
14	2009-09-24	RM		<ul style="list-style-type: none"> - xChannelHostState() return values updated - SYSTEM_CHANNEL_SYSTEM_STATUS_BLOCK structure elements updated (LEDs removed, CPUload and System time added) - SYSTEM_CHANNEL_SYSTEM_INFO_BLOCK structure added by bDevIdNumber - Correction: CHANNEL_INFORMATION: length for abFWname changed in documentation from 64 to 63. - Correction: CHANNEL_INFORMATION: Element usFWBuild before element usFWRevision
15	2010-04-23	MT/RM		Toolkit Chapters moved to separate manual Added: DMA for Windows Driver Added Notification Callbacks Added Bus-Synchronous operation Windows 7 support added x64 Support added Note: new header file "stdint.h" must included in new projects now
16	2010-08-20	MT		Moved Windows CE manual into separate documentation
17	2011-01-17	MT	5.x	All return values of functions are now documented with a reference to section <i>Error Codes</i> and to function xDriverGetErrorDescription.
18	2011-04-13	RM		<ul style="list-style-type: none"> - Added note of none deterministic Window behavior - xDriverMemoryPtr() / xChannelPLCMemoryPtr() added note of unsupported command 2 = CIFX_MEM_PTR_USR - Notification functions, added note for the notification handling
19	2011-09-21	RM	5.6.6 5.7.5	xDriverEnumChannel parameter description fixed Updated to CIFX Device Driver V1.1.x.x - xSysdeviceBootstart() function added
20	2011-10-19	RM	2.4 5.3, 5.4	New chapter <i>Windows Operating System Timing Behaviour</i> added <ul style="list-style-type: none"> - Added information about Windows driver time issues - Added diagrams with access times Added information about function with performance improvements
21	2011-12-12	RM	5.3 5.7.13 2.7	Updated to CIFX Device Driver V1.1.1.x Added new API function xSysdeviceExtendedMemory() Extended SYSTEM_CHANNEL_SYSTEM_STATUS_BLOCK by ulHWFeatures Added information about device time setting during start-up

Table 1: List of Revisions

1.3 Terms, Abbreviations and Definitions

Term	Description
cifX	Communication Interface based on netX
comX	C ommunication M odule based on netX
PCI	P eripheral C omponent I nterconnect
WDM	W indows D river M odel
DLL	D ynamic L ink L ibrary
API	A pplication P rogramming I nterface
SDO	S ervice D ata O bject
PDO	P rocess D ata O bject
DPM	D ual- P ort M emory Physical interface to all communication board (DPM is also used for PROFIBUS- DP Master).

Table 2: Terms, Abbreviations and Definitions

1.4 References

This document based on the following specification:

- [1] Real-time Communication System for netX
- [2] netX Bootstrap Specification
- [3] netX Program Reference Guide (PCI)
- [4] netX DPM Interface Manual

Table 3: References

1.5 Legal Notes

1.5.1 Copyright

© 2006-2011 Hilscher Gesellschaft für Systemautomation mbH

All rights reserved.

The images, photographs and texts in the accompanying material (manual, accompanying texts, documentation, etc.) are protected by German and international copyright law as well as international trade and protection provisions. You are not authorized to duplicate these in whole or in part using technical or mechanical methods (printing, photocopying or other methods), to manipulate or transfer using electronic systems without prior written consent. You are not permitted to make changes to copyright notices, markings, trademarks or ownership declarations. The included diagrams do not take the patent situation into account. The company names and product descriptions included in this document may be trademarks or brands of the respective owners and may be trademarked or patented. Any form of further use requires the explicit consent of the respective rights owner.

1.5.2 Important Notes

The manual, accompanying texts and the documentation were created for the use of the products by qualified experts, however, errors cannot be ruled out. For this reason, no guarantee can be made and neither juristic responsibility for erroneous information nor any liability can be assumed. Descriptions, accompanying texts and documentation included in the manual do not present a guarantee nor any information about proper use as stipulated in the contract or a warranted feature. It cannot be ruled out that the manual, the accompanying texts and the documentation do not correspond exactly to the described features, standards or other data of the delivered product. No warranty or guarantee regarding the correctness or accuracy of the information is assumed.

We reserve the right to change our products and their specification as well as related manuals, accompanying texts and documentation at all times and without advance notice, without obligation to report the change. Changes will be included in future manuals and do not constitute any obligations. There is no entitlement to revisions of delivered documents. The manual delivered with the product applies.

Hilscher Gesellschaft für Systemautomation mbH is not liable under any circumstances for direct, indirect, incidental or follow-on damage or loss of earnings resulting from the use of the information contained in this publication.

1.5.3 Exclusion of Liability

The software was produced and tested with utmost care by Hilscher Gesellschaft für Systemautomation mbH and is made available as is. No warranty can be assumed for the performance and flawlessness of the software for all usage conditions and cases and for the results produced when utilized by the user. Liability for any damages that may result from the use of the hardware or software or related documents, is limited to cases of intent or grossly negligent violation of significant contractual obligations. Indemnity claims for the violation of significant contractual obligations are limited to damages that are foreseeable and typical for this type of contract.

It is strictly prohibited to use the software in the following areas:

- for military purposes or in weapon systems;
- for the design, construction, maintenance or operation of nuclear facilities;
- in air traffic control systems, air traffic or air traffic communication systems;
- in life support systems;
- in systems in which failures in the software could lead to personal injury or injuries leading to death.

We inform you that the software was not developed for use in dangerous environments requiring fail-proof control mechanisms. Use of the software in such an environment occurs at your own risk. No liability is assumed for damages or losses due to unauthorized use.

1.5.4 Export

The delivered product (including the technical data) is subject to export or import laws as well as the associated regulations of different countries, in particular those of Germany and the USA. The software may not be exported to countries where this is prohibited by the United States Export Administration Act and its additional provisions. You are obligated to comply with the regulations at your personal responsibility. We wish to inform you that you may require permission from state authorities to export, re-export or import the product.

2 Windows 2000, XP, Vista and Windows 7

IMPORTANT NOTE: Windows® is not a deterministic real-time operating system. Any response times to specific hardware or driver functions can not be guaranteed and may differ between different versions of the Windows® operating systems. Furthermore, response times are also depending on the used host hardware, host performance, running services and installed software components.

2.1 Overview

- The cifX Device Driver for the Microsoft desktop operating systems is a kernel mode WDM driver, running in Ring 0 of the operating system. This driver is designed to support the Windows Plug & Play mechanism
- Communication between a user application and the driver is handled by an API DLL. This DLL can be statically or dynamically linked to the application.

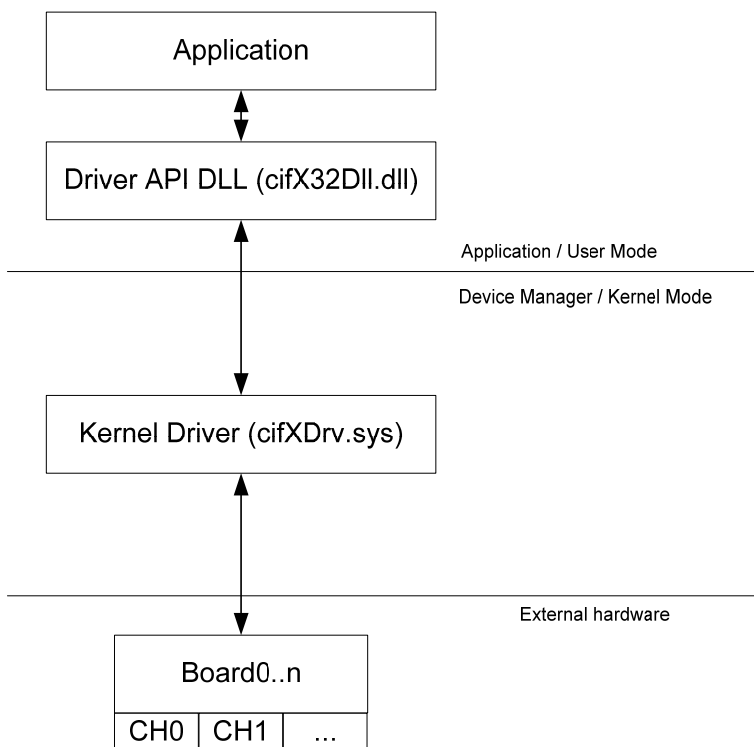


Figure 1: CifX Device Driver - Architecture

2.2 Features

Requirements:

- Operating System: Windows 2000, XP, Vista (32/64Bit) or Windows 7 (32/64Bit)
- cifX board or NXSB-PCA / NXSB100 / NXHX board or NX-PCA-PCI / NXHX

Features:

- Based on the cifX Toolkit source
- Driver architecture based on the Microsoft KMDf (Kernel Mode Driver Framework) architecture
- Compatible to the Plug&Play mechanism of Windows 2000, XP, Vista and Windows 7
- x86 and x64 (also known as AMD64) support for Vista and Windows 7
- Support for PCI, PCIe, PC/104, PCI-104, ISA netX based hardware
- Unlimited number of cifX boards supported
- Support for NXSB-PCA or NX-PCA-PCI boards included (PCI-Adapter to a netX DPM)
- DMA data transfer for I/O data
- Interrupt notification for applications
- Support of 2nd Memory Window for PCI based device (e.g. MRAM)
- Setting the device time during start-up if time handling is supported by the device

2.3 Limitations

- Windows Plug & Play power save mode supported by the driver
Attention: The actual netX hardware states are not stored and will be lost during power down!
On system wake-up the hardware is re-started like on system power-on.
- No IA64 support
- Response times of driver calls are operating system and system load depending. Deterministic response times can't be guaranteed

IMPORTANT NOTE: Windows[®] is not a deterministic real-time operating system. Any response times to specific hardware or driver functions can not be guaranteed and may differ between different versions of the Windows[®] operating systems. Furthermore, response times are also depending on the used host hardware, host performance, running services and installed software components.

2.4 Windows Operating System Timing Behaviour

IMPORTANT NOTE: Windows® is not a deterministic real-time operating system. Any response times to specific hardware or driver functions can not be guaranteed and may differ between different versions of the Windows® operating systems. Furthermore, response times are also depending on the used host hardware, host performance, running services and installed software components.

Depending on the system layout and system load the processing speed of driver calls are more or less deterministic. Under specific circumstances the Windows operating will re-schedule running processes which could lead to very long function call durations (factor 10 to 100 higher than average time).

Researching this behavior shows a possible re-scheduling during transition of the driver function call from “User-Space” to “Kernel-Space” or during processing the IRP in kernel mode.

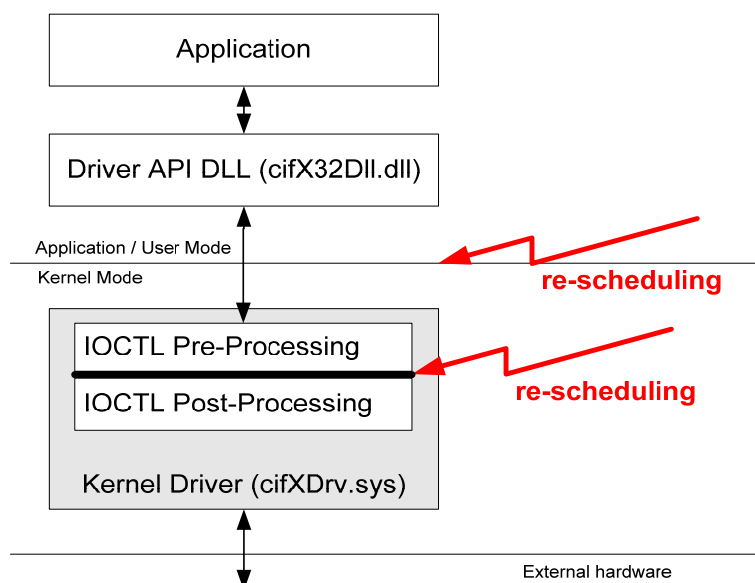


Figure 2: CifX Device Driver – System Architecture

At least, re-scheduling could appear at all stages during the call into the driver. A User-Space, important applications is able to increase its process and thread priority to achieve better performance and lower the impact of other running processes.

At driver level, some of the CIFX API functions, usually used during cyclic device handling, are executed directly at pre-processing stage to prevent re-scheduling.

Both measures are helpful in getting more deterministic function call durations, but there is no 100% guarantee of a deterministic program flow.

Note: Specially handled CIFX API functions are marked in function overview tables of chapters *System Device Related Functions* (page 29) and *Communication Channel Related Functions* (page 30).

Access Time Measurements:

Test System	Windows 7 / 64Bit, Intel Core2Duo E6550 2,33 GHz 1 GB RAM
Process Priority	NORMAL_PRIORITY_CLASS
Thread Priority	THREAD_PRIORITY_TIME_CRITICAL
I/O - Cycles	100000, cycle time 1ms
CIFX Device	CIFX50-DP (PROFIBUS Master V2.3.22.5 / Slave: CB-AB32 (2 Bytes In/Out))

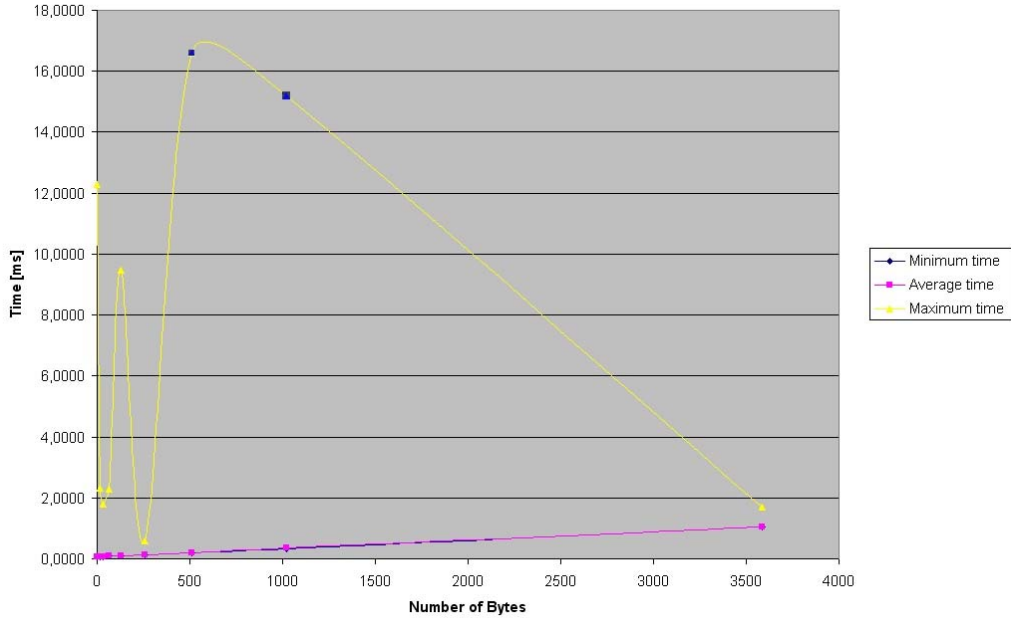


Figure 3: Windows 7 64Bit with standard IOCTL handling

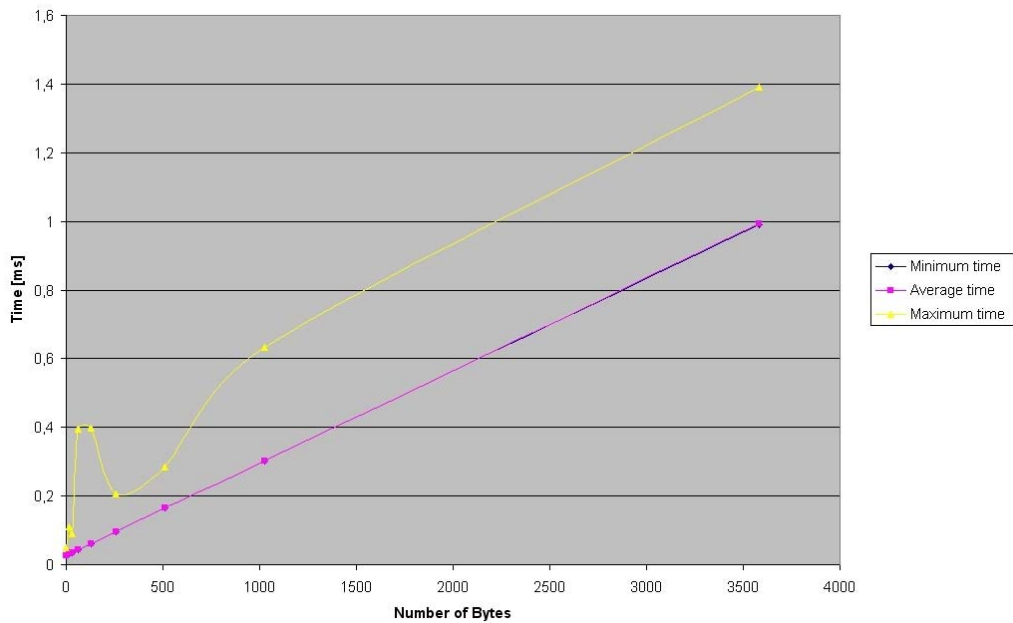


Figure 4: Windows 7 64Bit with direct IOCTL handling

2.5 Installation

The cifX Device Driver is compatible to the Plug and Play functionalities from Windows and offers two types of installation methods.

Installation Methods:

- Installation using the driver *Setup.exe* application (**preferred method**)
The setup application allows a "driver pre-installation" (software first) without hardware and also offers a un-installation.
- Installation using an INF file
This assumes a connected hardware and does not allow an un-installation of the driver and its components (uninstall under Vista and Win7 by Windows device manager)

Both methods are creating several directories on the PC system partition and registry entries to start the driver.

Following steps are processed by the driver setup and INF file:

- Copy necessary driver files to the target system

File name	Description	Destination
cifXDrv.sys	Device driver	.\Windows\System32\drivers
cifX32DLL.dll	Driver API	.\Windows\System32
cifXDrv.cpl	Control applet to start the driver setup or driver test program from the Windows control panel	.\Windows\System32
cifXSetup.exe	Driver setup program	.\Program Files\CIFx Device Driver
cifXTest.exe	Driver test program	.\Program Files\CIFx Device Driver
NETX100-BSL.bin	cifX / netX 100 bootloader	.\Program Files\CIFx Device Driver
NETX50-BSL.bin	netX 50 bootloader	.\Program Files\CIFx Device Driver
x64 only		
cifX32DLL.dll	Compatibility dll for 32 Bit applications running on a 64 Bit Windows	.\Windows\SysWow64

Table 4: cifX Device Driver - Files Installed by the INF File

- Creating driver specific registry entries

Destination
HKLM\System\CurrentControlSet\Services\CIFxDrv

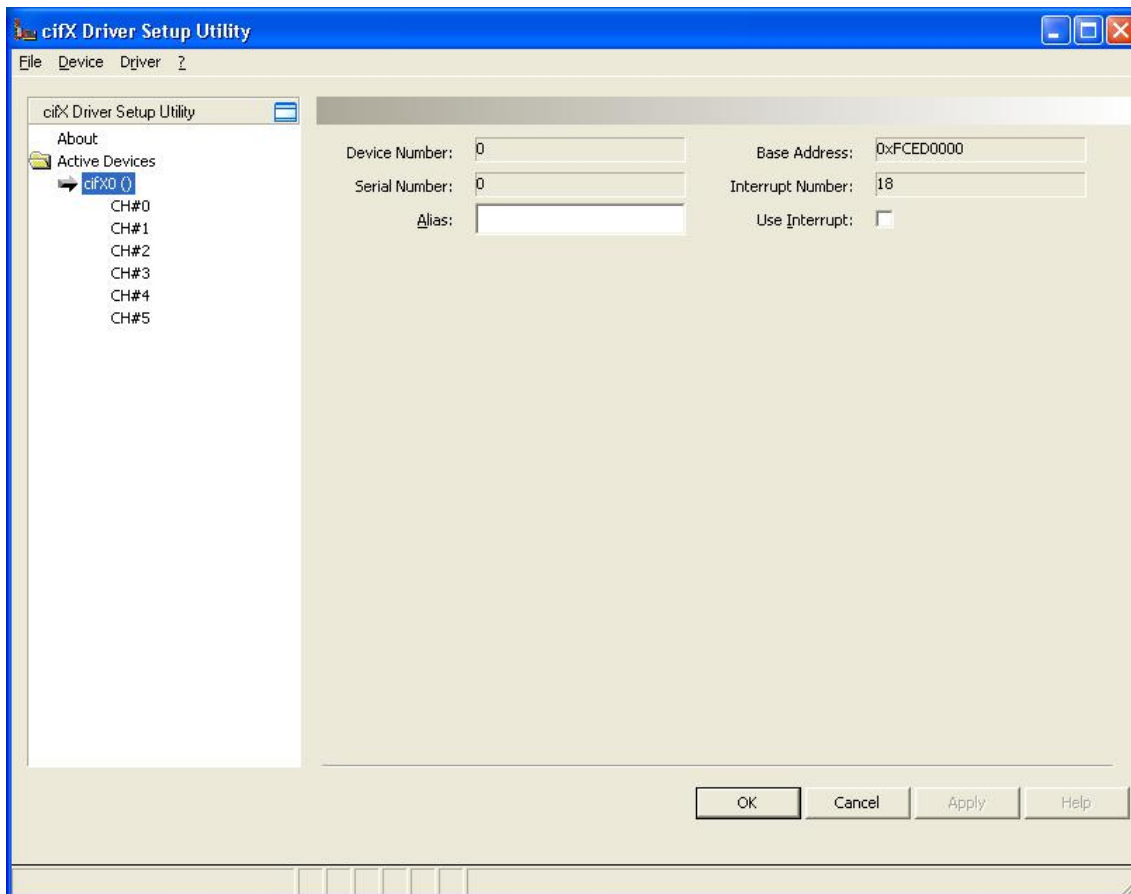
Table 5: cifX Device Driver - Registry Keys created by the INF File

2.6 Driver Setup and Test Program

The *cifX Device Driver* for the Microsoft desktop operating systems is delivered with a separate 'Setup' and 'Test' program. Both programs are automatically installed during the driver installation.

The driver setup program is described in a separate '*cifX Driver Installation*' manual.

Driver Setup Program:



2.7 Device Time Setting if Device Supports Time Handling

The driver will automatically setup the device time if the device signals the time handling feature in the system status information structure (*SYSTEM_CHANNEL_SYSTEM_STATUS_BLOCK / ulHWFeatures*) of the system channel.

Time setting is only executed if the time module of the device signals a not already set device clock. The device clock can be software driven or a physical RTC, which depends on the hardware assembly option and the firmware implementation.

Setting the device time is handled by an asynchronous command sent to the hardware after the firmware is successfully started and the information in the dual port memory signals an available time handling and a not already set time state.

3 General Overview

3.1 Data Handling

The default interface between a host system (e.g. PC) and the cifX hardware (cifX / comX) is a dual-port memory (DPM).

The DPM is a memory area from the hardware, mapped into to the system memory of the host system. All functionalities of the cifX / comX hardware are accessible via this DPM.

A complete description of the DPM, including handling mechanisms, hardware functions and data areas can be found in the "netX DPM Interface Manual".

All driver functions are based on these DPM functions and the hardware offers two mechanisms to transfer user data.

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 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.

3.2 Devices, Channels and Driver Functions

The cifX driver supports an unlimited number of PC card, also named devices. The driver creates an own "DOS device" per PC card and the name of created devices is "cifXn" where "n" is an increment number, depending of the number of cifX cards installed in the host system.

Each device is represented by its dual port memory (DPM). The DPM of each card is divided into 8 channels.

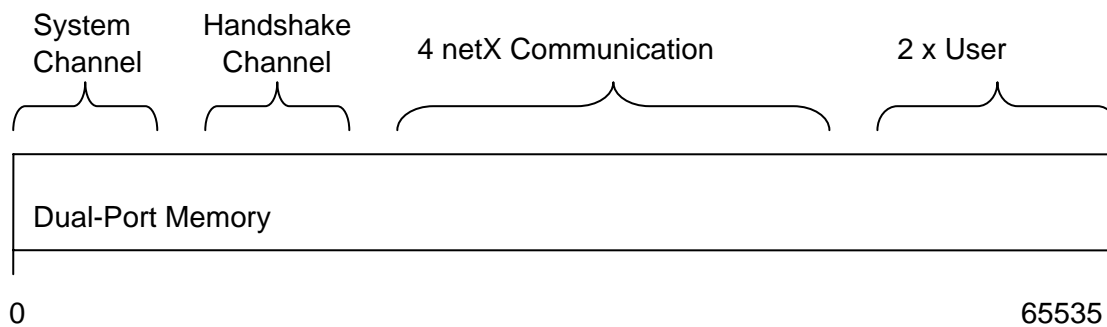
The standard netX firmware offers up to 8 channels with three different channel definitions.

The main channel is the '*System Channel*' also named system device. This one is always available and used for administration functions like hardware reset, firmware download and so on.

'*Communication Channels*' are the second type of defined channels and used to represent fieldbus connections. Up to four communication channels are possible.

'*User channels*' are the third type of channels and designed for user applications, running on the netX chip (two user channels are possible).

General DPM Layout:



This layered design of a device with channels can also be found in the function names of the cifX driver API.

- **xDriver.....()** Global driver functions
- **xSystemDevice....()** System device related functions
- **xChannel.....()** Communication channel related functions

3.3 Timer Resolution

The following table lists some operating systems with its standard minimum timer resolution.

Operating System	Timer Resolution in ms
DOS	54,95 ms (18.2 ticks per second)
Windows 2000/XP/Vista/7	10 ms
Windows CE	Platform dependent (default 1ms)

Table 6: Timer Resolution

Note: Smaller timer resolution needs deeper knowledge of the operating system.

4 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 hardware.

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

4.1 The Main() Function

```
/* *****  
/*! The main function  
* \return 0 on success */  
/* *****  
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;  
}
```

4.2 System Device Example

```

/*****
*! 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\r\n",
      ulSendPktCount, ulRecvPktCount);

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

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

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

return lRet;
}

```

4.3 Communication Channel Example

```

/*****
/*! Function to demonstrate communication channel functionality
*   Packet Transfer and I/O Data exchange
*   \return CIFX_NO_ERROR on success
*/
*****/
int32_t ChannelDemo(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!");
    } 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 filed bus an application is ready */
lRet = xChannelHostState(hChannel,
                        CIFX_HOST_STATE_READY,
                        &ulState,
                        HOSTSTATE_TIMEOUT);

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

        /* Do I/O Data exchange until a key is hit */
        while(!kbhit())
        {
            if(CIFX_NO_ERROR != (lRet = xChannelIORead(hChannel,
                                                    0, 0, sizeof(abRecvData),
                                                    abRecvData,
                                                    IO_WAIT_TIMEOUT)))

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

                {
                    printf("Error writing to IO Data area!\r\n");
                    break;
                } else
                {
                    printf("IOWrite Data:");
                    DumpData(abSendData, sizeof(abSendData));
                    /* Create new output data */
                    memset(abSendData, ulCycle + 1, sizeof(abSendData));
                }
            }
        }
    }
}
}
}

```

```
/* Switch off the bus */
xChannelBusState( hChannel, CIFX_BUS_STATE_OFF, &ulState, 0L);
/* Set Host not ready to stop bus communication */
xChannelHostState(hChannel, CIFX_HOST_STATE_NOT_READY,
                  &ulState,
                  HOSTSTATE_TIMEOUT);
/* Close the communication channel */
xChannelClose(hChannel);
}

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

return lRet;
}
```

4.4 Board and Channel Enumeration

```

/*****
/*! Function to demonstrate the board/channel enumeration
*   \return CIFX_NO_ERROR on success
*/
*****/
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");
}

```

5 CIFX API (Application Programming Interface)

The *CIFX API* is the common driver interface to Hilscher cifx device drivers and the underlying netX based hardware. 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*' functions, so called '*System Device*' related functions and '*Communication Channel*' related functions.

These functions are listed in the following table.

5.1 Header Files

The CIFX API is defined in the *cifXUser.h* file. Error definitions can be found in the *cifXError.h* file.

5.2 Driver Related Functions

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

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

Table 7: Driver Related Functions

5.3 System Device Related Functions

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

Function	Description	Info
xSysdeviceOpen	Opens a connection to a boards system device	
xSysdeviceClose	Closes a connection to a system device	
xSysdeviceInfo	Get System device specific information (e.g. mailbox size)	X
xSysdeviceReset	Perform a device reset	
xSysdeviceBootstart	Perform a device boot start. This will activate the 2 nd Stage bootloader. An available Firmware will not be started. Note: Only possible on FLASH based devices.	
xSysdeviceGetMBXState	Retrieves the system mailbox state	X
xSysdeviceGetPacket	Retrieves a pending packet from the system mailbox	X
xSysdevicePutPacket	Send a packet to the system mailbox	X
xSysdeviceDownload	Downloads a file/configuration/firmware to the device	
xSysdeviceFindFirstFile	Find the first file entry in the given directory	
xSysdeviceFindNextFile	Find the next first file entry in the given directory	
xSysdeviceUpload	Uploads a file/configuration/firmware from the device	
xSysdeviceExtendedMemory	Get a pointer to an available extended memory area	

Table 8: System Device Related Functions

(X) Marked functions are handled with higher priority under Windows Vista/7.

5.4 Communication Channel Related Functions

Each protocol stack is represented as a communication channel by the driver.

Communication channels have 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.

A table with Communication Channel Related Functions, see next page.

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

Function	Description	Info
xChannelPLCIsWriteReady	Checks if the last Write Activation has finished	X
DMA services		
xChannelDMAState	Activate/Deactivate DMA mode	
Bus synchronous operation		
xChannelSyncState	Wait for synchronization event or trigger/acknowledge sync	X
Notification services (only available in Interrupt mode)		
xChannelRegisterNotification	Register a notification callback	
xChannelUnregisterNotification	Un-register a notification callback	

Table 9: Communication Channel Related Functions

(X) Marked functions are handled with higher priority under Windows Vista/7.

5.5 Structure definitions

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

5.5.1 Driver Information

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

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

Table 10: Driver Information Structure

5.5.2 Board Information

The board information structure is used, when enumerating boards. Its layout is as follows

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

Table 11: Board Information Structure

5.5.3 System Channel Information

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

Command: CIFX_INFO_CMD_SYSTEM_INFORMATION

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

Table 12: System Channel Information

Command: CIFX_INFO_CMD_SYSTEM_INFO_BLOCK:

SYSTEM_CHANNEL_SYSTEM_INFO_BLOCK		
Element	Type	Description
abCookie	uint8_t[4]	System channel identifier MUST be "netX"
ulDpmTotalSize	uint32_t	Total size of the dual port in bytes
ulDeviceNumber	uint32_t	Device number (as found on the barcode)
ulSerialNumber	uint32_t	Serial number (as found on the barcode)
ausHwOptions	uint16_t[4]	Array of hardware options for all four possible ports of the netX
usManufacturer	uint16_t	Manufacturer ID
usProductionDate	uint16_t	Production date code
ulLicenseFlags1	uint32_t	License flags
ulLicenseFlags2	uint32_t	Additional License flags
usNetxLicenseID	uint16_t	
usNetxLicenseFlags	uint16_t	
usDeviceClass	uint16_t	
bHwRevision	uint8_t	Hardware revision
bHwCompatibility	uint8_t	Hardware compatibility list
bDevIdNumber	uint8_t	Device identification number (rotary switch)
bReserved	uint8_t	unused/reserved
usReserved	uint16_t	unused/reserved

Table 13: System Channel Info Block

Command: CIFX_INFO_CMD_SYSTEM_CHANNEL_BLOCK:

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

Table 14: System Channel Channel Info Block

Area definitions in cifXUser.h:

CIFX_MAX_NUMBER_OF_CHANNEL_DEFINITION = 8

CIFX_SYSTEM_CHANNEL_DEFAULT_INFO_BLOCK_SIZE = 16

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

Command: CIFX_INFO_CMD_SYSTEM_CONTROL_BLOCK:

SYSTEM_CHANNEL_SYSTEM_CONTROL_BLOCK		
Element	Type	Description
ulSystemCommandCOS	uint32_t	System channel host COS flags
ulReserved	uint32_t	unused/reserved

Table 15: System Channel Control Block

Command: CIFX_INFO_CMD_SYSTEM_STATUS_BLOCK:

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

Table 16: System Channel Status Block

5.5.4 Communication Channel Information

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

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

Table 17: Channel Information Structure

5.6 Driver Related Functions

5.6.1 xDriverOpen

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.6.2 xDriverClose

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

Function call:

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

Arguments:

Argument	Data type	Description
hDriver	CIFXHANDLE	Handle returned by xDriverOpen

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.6.3 xDriverGetInformation

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 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)
{
}
```

5.6.4 xDriverGetErrorDescription

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

Example:

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

5.6.5 xDriverEnumBoards

Enumerate all currently handled boards/cards of the driver.

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

Example:

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

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

5.6.6 xDriverEnumChannels

Enumerate all available channels on a board/card.

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

Example:

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

5.6.7 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:

```
int32 t APIENTRY xDriverRestartDevice(    CIFXHANDLE hDriver,
                                         char*      szBoardName,
                                         void*      pvData);
```

Arguments:

Argument	Data type	Description
hDriver	CIFXHANDLE	Handle to the driver (returned by <i>xDriverOpen</i>)
szBuffer	String	Identifier for the Board. (e.g. "cifX<BoardNumber>")
pvData	void*	For further extensions can be NULL

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.6.8 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 xDriverMemoryPtr( CIFXHANDLE hDriver,
                          uint32_t ulBoard,
                          uint32_t ulCmd,
                          void* pvMemoryInfo)
```

Arguments:

Argument	Data type	Description
hDriver	CIFXHANDLE	Handle to the driver (returned by xDriverOpen)
ulBoard	uint32_t	Board number to return pointer for.
ulCmd	uint32_t	Maps the dual port memory for direct access from an application 1 = CIFX_MEM_PTR_OPEN Map a user specific memory area 2 = CIFX_MEM_PTR_USR -> not supported Release the dual port pointer (same memory structure MUST be passed) 3 = CIFX_MEM_PTR_CLOSE
pvMemoryInfo	void*	Pointer to returned MEMORY_INFORMATION structure 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 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

Description of the MEMORY_INFORMATION Structure:

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

MEMORY_INFORMATION Structure:

```

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

```

Example:

```

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

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

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

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

```

5.7 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.

5.7.1 xSysdeviceOpen

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

Function call:

```
int32 t  xSysdeviceOpen(  CIFXHANDLE    hDriver,
                        char*      szBoard,
                        CIFXHANDLE* phSysdevice);
```

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.7.2 xSysdeviceClose

Close a connection to a system device.

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.7.3 xSysdeviceInfo

Query information about the opened system device.

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.7.4 xSysdeviceReset

Performs a hardware reset on the device.

Note: All channels will be reset.

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.7.5 xSysdeviceBootstart

Performs 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:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.7.6 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:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.7.7 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:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.7.8 xSysdeviceGetPacket

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.7.9 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:

```
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:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.7.10 xSysdeviceFindFirstFile

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.7.11 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:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.7.12 xSysdeviceUpload

Upload a given file from the device.

Function call:

```
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:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.7.13 xSysdeviceExtendedMemory

netX based PCI hardware is able to offer a second PCI memory window used to access 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 (Magnetoresistive 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 allow reading information and getting/returning the pointer to the extended memory.

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

Description of the CIFX_EXTENDED_MEMORY_INFORMATION Structure:

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

ulMemoryInformation:

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

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

Available definitions (see rcX_User.h):

```
#define RCX_SYSTEM_EXTMEM_TYPE_MSK           0x0000000F
#define RCX_SYSTEM_EXTMEM_TYPE_NONE         0x00000000
#define RCX_SYSTEM_EXTMEM_TYPE_MRAM_128K    0x00000001

#define RCX_SYSTEM_EXTMEM_ACCESS_MSK        0x000000C0
#define RCX_SYSTEM_EXTMEM_ACCESS_NONE       0x00000000
#define RCX_SYSTEM_EXTMEM_ACCESS_EXTERNAL   0x00000040
#define RCX_SYSTEM_EXTMEM_ACCESS_INTERNAL   0x00000080
#define RCX_SYSTEM_EXTMEM_ACCESS_BOTH      0x000000C0
```

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

CIFX_EXTENDED_MEMORY_INFORMATION structure:

```

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

```

Example:

```

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

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

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

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

        // Open a DPM memory pointer
        lRet = xSysdeviceExtendedMemory( hSysdevice, CIFX_GET_EXTENDED_MEMORY_INFO,
                                         &tExtMemory);

        if(lRet != CIFX_NO_ERROR)
        {
            // Failed to get the memory mapping
            ShowError( lRet);
        } else
        {
            /* Get an extended memory pointer */
            lRet = xSysdeviceExtendedMemory( hSysdevice, CIFX_GET_EXTENDED_MEMORY_POINTER,
                                             &tExtMemory);

            if(lRet != CIFX_NO_ERROR)
            {
                // Failed to get the memory mapping
                ShowError( lRet);
            }else
            {
                // We have a memory mapping
                uint8_t* pbExtMem = (uint8_t*)tExtMemory.pvMemoryPtr;

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

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

```

```
    printf("Increment the read data:\n");
    for ( uint32_t ulIdx =0; ulIdx < sizeof(abBuffer); ulIdx++)
    {
        abBuffer[ulIdx] +=1;
    }

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

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

if(lRet != CIFX_NO_ERROR)
{
    // Failed to free the memory mapping
    ShowError( lRet);
}
}

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

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

5.8 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.

5.8.1 xChannelOpen

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.2 xChannelClose

Close a connection to a communication channel.

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.3 xChannelDownload

Download a file to a communication channel.

Function call:

```
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:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.4 xChannelFindFirstFile

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.5 xChannelFindNextFile

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.6 xChannelUpload

Upload a given file from the communication channel.

Function call:

```
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:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.7 xChannelGetMBXState

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

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.8 xChannelPutPacket

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.9 xChannelGetPacket

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.10 xChannelGetSendPacket

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

The function is mainly used for debugging aids.

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.11 xChannelReset

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

- CIFX_CHANNELINIT Re-initialization of a communication channel
- CIFX_SYSTEMSTART Restart the whole card

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.12 xChannelInfo

Retrieve the global communication channel information.

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.13 xChannelIOInfo

Retrieve I/O information about the communication channel.

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.14 xChannelWatchdog

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

Function call:

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

Arguments:

Argument	Data type	Description
hChannel	CIFXHANDLE	Handle of the channel.
ulCmd	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 100 is returned. You can use the function `xDriverGetErrorDescription()` to get a description of this error.

5.8.15 xChannelConfigLock

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.16 xChannelHostState

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

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.17 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:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.18 xChannelControlBlock

Reading / writing the communication channel control block.

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.19 xChannelCommonStatusBlock

Read the channels common status block.

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.20 xChannelExtendedStatusBlock

Read the communication channels extended status block.

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.21 xChannelUserBlock

not implemented yet!

5.8.22 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:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.23 xChannellIOWrite

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.24 xChannelIOReadSendData

Reads the last output data image from the host.

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.25 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 *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 *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 *xChannelPLCMemoryPtr()*. This will deliver the necessary pointers to the requested I/O data image.

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

5.8.25.1 xChannelPLCMemoryPtr

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

Function call:

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

Arguments:

Argument	Data type	Description
hChannel	CIFXHANDLE	Handle of the channel.
ulCmd	uint32_t	PLC Memory Pointer Commands: 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 PLC_MEMORY_INFORMATION structure. This structure describes the requested area and also contains the returned memory pointer on success

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

Description of the PLC_MEMORY_INFORMATION Structure:

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

PLC_MEMORY_INFORMATION Structure:

```

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

```

Example:

```

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

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

    long lRet = CIFX_NO_ERROR;

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

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

        /* Define the memory structures for Output data */

```

```

unsigned char*   pabDPMemory_OUT       = NULL;
uint32_t        ulAreaStartOffset_OUT  = 0;
uint32_t        ulAreaSize_OUT         = 0;

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

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

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

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

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

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

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

            lRet = xChannelPLCIsReadReady ( hDevice, 0, &ulReadState);
            if( CIFX_NO_ERROR != lRet)
            {

```

```

    ShowError( lRet);
} else if( l == ulReadState)
{
    /* It is allowed to read the image */
    /* Read 100 Bytes */
    memcpy( abBuffer, pabDPMMemory, sizeof(abBuffer));

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

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

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

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

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

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

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

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

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

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

```

5.8.25.2 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:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.25.3 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 *xChannelPLCsWriteReady()*.

Note: If this function is called multiple times while the actual state is '*not ready*' (use the corresponding *xChannelPLCs.....Ready()* function), the result is unpredictable.

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.25.4 xChannelPLCIsReadReady

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.25.5 xChannelPLCIsWriteReady

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.26 DMA Functions

5.8.26.1 xChannelDMAState

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

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

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.27 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 the “netX Dual Port Memory Interface Manual”.

5.8.27.1 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.

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

Arguments:

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

Continued on next page.

Possible Notification Events:

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

5.8.27.2 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:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.27.3 xChannelUnregisterNotification

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

Function call:

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

Arguments:

Argument	Data type	Description
hChannel	CIFXHANDLE	Handle of the channel.
ulNotification	uint32_t	Possible Notification: 1 = CIFX_NOTIFY_RX_MBX_FULL 2 = CIFX_NOTIFY_TX_MBX_EMPTY 3 = CIFX_NOTIFY_PD0_IN 4 = CIFX_NOTIFY_PD1_IN 5 = CIFX_NOTIFY_PD0_OUT 6 = CIFX_NOTIFY_PD1_OUT 7 = CIFX_NOTIFY_SYNC

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

5.8.28 Bus-Synchronous Operations

Handles bus synchronous operation of the fieldbus protocol stack on the device. In general, synchronous operation distinguishes between device synchronous and host synchronous operation. The difference between the two modes is the component (host / device) which activates the synchronization and the response to the synchronization event.

- **Device Controlled 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()*.
- **Host Controlled Mode**
The host calls *xChannelSyncState()* to signal a synchronization.

Note: Synchronous data operations must be supported by the fieldbus protocol stack and 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, response 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 response timing issues.

Function call:

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

Arguments:

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

Return Values:

CIFX_NO_ERROR if the function succeeds.

If the function fails, a nonzero error code from chapter *Error Codes* from page 100 is returned. You can use the function *xDriverGetErrorDescription()* to get a description of this error.

6 Error Codes

Value	Symbol	Description
0x00000000	CIFX_NO_ERROR	No error
0x800Axxxx		
0x800A0001	CIFX_INVALID_POINTER	An 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 is available
0x800A0004	CIFX_INVALID_HANDLE	An invalid handle was passed to the function
0x800A0005	CIFX_INVALID_PARAMETER	Invalid parameter passed to 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. IO Area is exceeded by Offset and size)
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
0x800A000E	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 fit the device data
0x800A0013	CIFX_MEMORY_MAPPING_FAILED	Error mapping dual port memory
0x800A0014	CIFX_NO_MORE_ENTRIES	No more entries available (e.g. while enumerating directories)
0x800A0015	CIFX_CALLBACK_MODE_UNKNOWN	Unkown callback handling mode
0x800A0016	CIFX_CALLBACK_CREATE_EVENT_FAILED	Failed to create callback events
0x800A0017	CIFX_CALLBACK_CREATE_RECV_BUFFER	Failed to create callback receive buffer
0x800A0018	CIFX_CALLBACK_ALREADY_USED	Callback already used
0x800A0019	CIFX_CALLBACK_NOT_REGISTERED	Callback was not registered before
0x800A001A	CIFX_INTERRUPT_DISABLED	Interrupt is disabled
0x800Bxxxx		
0x800B0001	CIFX_DRV_NOT_INITIALIZED	Driver not initialized
0x800B0002	CIFX_DRV_INIT_STATE_ERROR	Driver init state error
0x800B0003	CIFX_DRV_READ_STATE_ERROR	Driver read state error
0x800B0004	CIFX_DRV_CMD_ACTIVE	Command is active on device
0x800B0005	CIFX_DRV_DOWNLOAD_FAILED	General error during download
0x800B0006	CIFX_DRV_WRONG_DRIVER_VERSION	Wrong driver version

Table 18: Error Codes (1)

Value	Symbol	Description
0x800B0030	CIFX_DRV_DRIVER_NOT_LOADED	CIFx driver is not running
0x800B0031	CIFX_DRV_INIT_ERROR	Failed to initialize the device
0x800B0032	CIFX_DRV_CHANNEL_NOT_INITIALIZED	Channel not initialized (xOpenChannel() not called)
0x800B0033	CIFX_DRV_IO_CONTROL_FAILED	IOControl call failed
0x800B0034	CIFX_DRV_NOT_OPENED	Driver was not opened
0x800C0010	CIFX_DEV_DPM_ACCESS_ERROR	Dual port memory not accessible (board not found)
0x800C0011	CIFX_DEV_NOT_READY	Device not ready (ready flag failed)
0x800C0012	CIFX_DEV_NOT_RUNNING	Device not running (running flag failed)
0x800C0013	CIFX_DEV_WATCHDOG_FAILED	Watchdog test failed
0x800C0015	CIFX_DEV_SYSERR	Error in handshake flags
0x800C0016	CIFX_DEV_MAILBOX_FULL	Send mailbox is full
0x800C0017	CIFX_DEV_PUT_TIMEOUT	Send packet timeout
0x800C0018	CIFX_DEV_GET_TIMEOUT	Receive packet timeout
0x800C0019	CIFX_DEV_GET_NO_PACKET	No packet available
0x800C001A	CIFX_DEV_MAILBOX_TOO_SHORT	Mailbox is too short for a packet
0x800C0020	CIFX_DEV_RESET_TIMEOUT	Reset command timeout
0x800C0021	CIFX_DEV_NO_COM_FLAG	Communication flag not set
0x800C0022	CIFX_DEV_EXCHANGE_FAILED	I/O data exchange failed
0x800C0023	CIFX_DEV_EXCHANGE_TIMEOUT	I/O data exchange timeout
0x800C0024	CIFX_DEV_COM_MODE_UNKNOWN	Unknown I/O exchange mode
0x800C0025	CIFX_DEV_FUNCTION_FAILED	Device function failed
0x800C0026	CIFX_DEV_DPMSIZE_MISMATCH	DPM size differs from configuration
0x800C0027	CIFX_DEV_STATE_MODE_UNKNOWN	Unknown state mode
0x800C0028	CIFX_DEV_HW_PORT_IS_USED	Device is still accessed
0x800C0029	CIFX_DEV_CONFIG_LOCK_TIMEOUT	Configuration locking timeout
0x800C002A	CIFX_DEV_CONFIG_UNLOCK_TIMEOUT	Configuration unlocking timeout
0x800C002B	CIFX_DEV_HOST_STATE_SET_TIMEOUT	Set HOST state timeout
0x800C002C	CIFX_DEV_HOST_STATE_CLEAR_TIMEOUT	Clear HOST state timeout
0x800C002D	CIFX_DEV_INITIALIZATION_TIMEOUT	Timeout during channel initialization
0x800C002E	CIFX_DEV_BUS_STATE_ON_TIMEOUT	Timeout setting bus on flag
0x800C002F	CIFX_DEV_BUS_STATE_OFF_TIMEOUT	Timeout setting bus off flag
0x800C0040	CIFX_DEV_MODULE_ALREADY_RUNNING	Module already running
0x800C0041	CIFX_DEV_MODULE_ALREADY_EXISTS	Module already exists

Table 19: Error Codes (2)

Value	Symbol	Description
0x800C0050	CIFX_DEV_DMA_INSUFF_BUFFER_COUNT	Number of configured DMA buffers insufficient
0x800C0051	CIFX_DEV_DMA_BUFFER_TOO_SMALL	DMA buffers size too small (min size 256Byte)
0x800C0052	CIFX_DEV_DMA_BUFFER_TOO_BIG	DMA buffers size too big (max size 63,75KByte)
0x800C0053	CIFX_DEV_DMA_BUFFER_NOT_ALIGNED	DMA buffer alignment failed (must be 256Byte)
0x800C0054	CIFX_DEV_DMA_HANSHAKEMODE_NOT_SUPPORTED	I/O data uncontrolled handshake mode not supported
0x800C0055	CIFX_DEV_DMA_IO_AREA_NOT_SUPPORTED	I/O area in DMA mode not supported (only area 0 possible)
0x800C0056	CIFX_DEV_DMA_STATE_ON_TIMEOUT	Set DMA ON Timeout
0x800C0057	CIFX_DEV_DMA_STATE_OFF_TIMEOUT	Set DMA OFF Timeout
0x800C0058	CIFX_DEV_SYNC_STATE_INVALID_MODE	Device is in invalid mode for this operation
0x800C0059	CIFX_DEV_SYNC_STATE_TIMEOUT	Waiting for synchronization event bits timed out

Table 20: Error Codes (3)

7 Appendix

7.1 List of Tables

Table 1: List of Revisions	5
Table 2: Terms, Abbreviations and Definitions.....	6
Table 3: References	6
Table 4: cifX Device Driver - Files Installed by the INF File.....	13
Table 5: cifX Device Driver - Registry Keys created by the INF File	13
Table 6: Timer Resolution	19
Table 7: Driver Related Functions.....	28
Table 8: System Device Related Functions.....	29
Table 9: Communication Channel Related Functions	31
Table 10: Driver Information Structure	32
Table 11: Board Information Structure	32
Table 12: System Channel Information	33
Table 13: System Channel Info Block	33
Table 14: System Channel Channel Info Block.....	34
Table 15: System Channel Control Block.....	34
Table 16: System Channel Status Block	34
Table 17: Channel Information Structure	35
Table 18: Error Codes (1).....	100
Table 19: Error Codes (2).....	101
Table 20: Error Codes (3).....	102

7.2 List of Figures

Figure 1: CifX Device Driver - Architecture.....	9
Figure 2: CifX Device Driver – System Architecture.....	11
Figure 3: Windows 7 64Bit with standard IOCTL handling	12
Figure 4: Windows 7 64Bit with direct IOCTL handling	12

7.3 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 - 110 025
Phone: +91 11 40515640
E-Mail: info@hilscher.in

Italy

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

Support

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

Japan

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

Support

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

Korea

Hilscher Korea Inc.
Suwon, 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