Toolkit Manual

cifX/netX Toolkit

DPM

V2.1

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC090203TK11EN | Revision 11 | English | 2019-04 | Released | Public

Table of contents

1 Introduction

1.1 About this document

The *cifX/netX Toolkit* consists of C-source and header files allowing abstract access to the dual-port memory (DPM) defined by Hilscher for cifX and comX devices and netX based components.

It contains the user interface functions (CIFX API) as well as generic access functions needed to handle the Hilscher DPM.

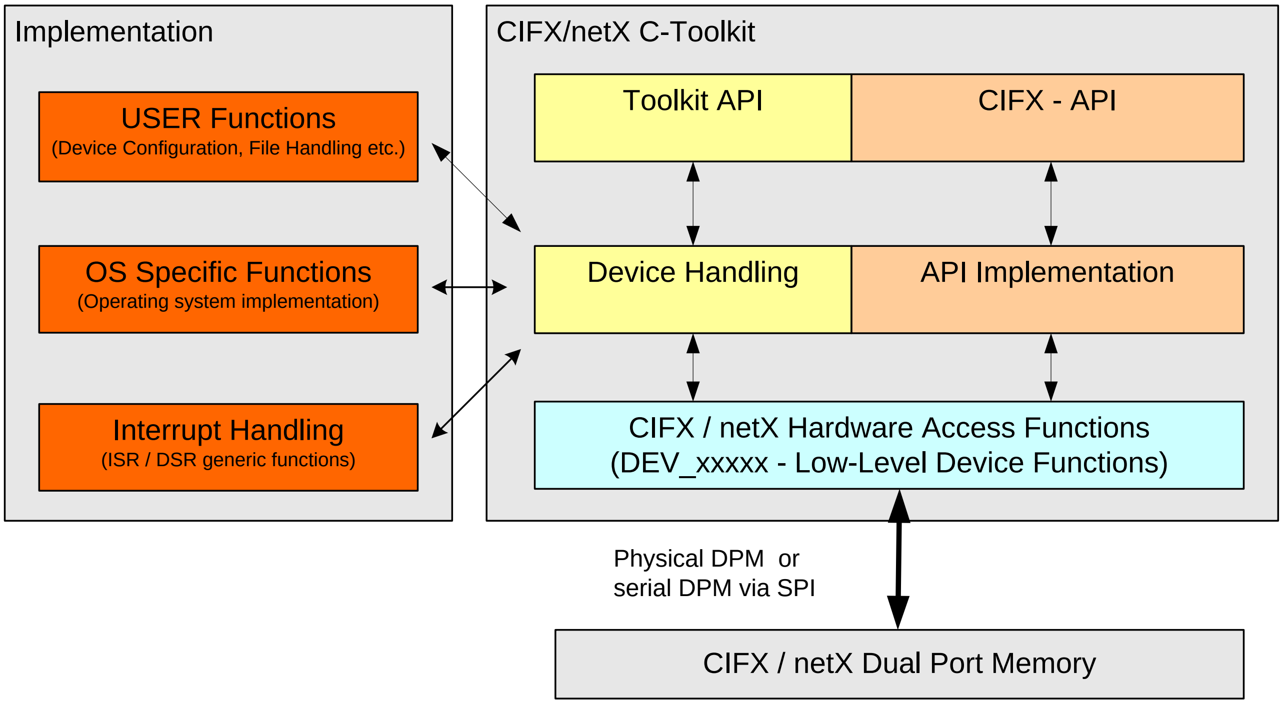

Figure 1: Toolkit overview

All Hilscher CIFX/COMX device drivers are based on the toolkit and the structure of the toolkit is designed to be portable and adjustable to different operating system. Therefore all operating depended functions (OS_ functions) and the so called USER functions (USER_ functions), needed for the device start-up, download and configuration handling are placed in separate source modules.

Furthermore, the toolkit hardware access functions (DEV_ functions) can be used to create small Microcontroller based applications.

To adapt the toolkit, only the separate modules (described in *OS Abstraction* on page 60 and *USER implemented functions* on page 84) must be implemented according to the used operating system.

Note:	The CIFX API is described in the *CIFX API - Application Programming Interface* manual.

This manual describes the implementation of the cifX/netX Toolkit and the porting to own operating systems.

1.2 List of revisions

Rev	Date	Name	Chapter	Revision
10	2018-08-29	RMA		Toolkit V1.5
			4.1.1	Section *cifX Toolkit CD* added.
			4.5	Section *FLASH-based vs RAM-based devices* added.
			5.1.1	Section *User definable data in the DEVICEINSTANCE structure*: eDeviceType description added.
			6.1.3	Section *cifXTKitAddDevice*: description of ptDevInst argument fixed.
			6.2.7	Section *PCI routines*: note to store / restore the complete PCI_COMMON_CONFIG structure added.
11	2019-04-26	ALM / LCO		Toolkit V2.1
			3.1.2	Sections *OS_SpiInit*, *OS_SpiLock*, and *OS_SpiUnlock* added.
			4.5	Note about firmware update handling in case of netX90/netX4000 added.
			4.6	File extensions NXI, NAI added.
			4.6.1.2	Note about firmware update handling for netX90/netX4000 added.
			7.2	Note about netX4000 PCIe devices added.
			7.2.1	CIFX4000 PCI information/IDs added.
			9	Section *Error codes* updated.

Table 1: List of revisions

1.3 Terms, abbreviations and definitions

Term	Description
cifX	**C**ommunication **I**nter**f**ace based on netX
comX	**Co**mmunication **M**odule based on netX
DPM	**D**ual-**P**ort **M**emory Physical interface to all communication board (DPM is also used for PROFIBUS-**DP M**aster).
PCI	**P**eripheral **C**omponent **I**nterconnect
API	**A**pplication **P**rogramming **I**nterface
NXF	File extension of a Hilscher **n**et**X F**irmware or Base OS Firmware
NXO	File Extension of a Hilscher **n**et**X** Firmware m**o**dule
SDO	**S**ervice **D**ata **O**bject
PDO	**P**rocess **D**ata **O**bject

Table 2: Terms, abbreviations and definitions

All variables, parameters, and data used in this manual have the LSB/MSB (“Intel”) data format. This corresponds to the convention of the Microsoft C Compiler.

All IP addresses in this document have host byte order.

1.4 References to documents

This document based on the following documents and specifications:

[1] Hilscher Gesellschaft für Systemautomation mbH: CIFX API - Application Programming Interface, Revision 6, english, 2019.

[2] Hilscher Gesellschaft für Systemautomation mbH: Dual-Port Memory Interface Manual, Revision 15, english, 2019.

[3] Hilscher Gesellschaft für Systemautomation mbH: Packet API, netX Dual-Port Memory, Packet-based services (netX 10/50/51/52/100/500), Revision 3, english, 2019.

[4] Hilscher Gesellschaft für Systemautomation mbH: Packet API, netX Dual-Port Memory, Packet-based services (netX 90/4000/4100), Revision 2, english, 2019.

[5] Hilscher Gesellschaft für Systemautomation mbH: Driver Manual, cifX Device Driver, Windows 2000/XP/Vista/7 V1.1.x.x. Revision 23, english, 2016.

[6] Hilscher Gesellschaft für Systemautomation mbH: Getting Started Guide, Serial Dual-Port Memory Interface with netX, Revision 6, english, 2018.

Table 3: References to documents

1.5 Features

- Support of PCI / ISA and DPM based connections to the Hilscher DPM
- Support of memory and FLASH based devices
- netX100/500, netX50, netX51/52 Bootstrap support
- Basic interrupt functions included
- Event handling for I/O and packet transfer functions
- Support of *Loadable Firmware Modules* (NXO files) consisting of a *Base OS Module* and *Loadable Protocol Stack Modules*
- 64 Bit support
- **Options:**
  - Little Endian / Big Endian support (selectable via toolkit definition)
  - DMA support for I/O data transfer (selectable via a toolkit definition)
  - Extended Parameter Check of Toolkit Functions (selectable via a toolkit definition)
  - Device time setting during start-up
  - Custom Hardware Access Interface (e.g. DPM via SPI, selectable via a toolkit definition)

1.6 Restrictions

The following restrictions apply when using the cifX/*netX Toolkit*:

- Several functions must be implemented by the user, before being able to use the toolkit
- Basic Interrupt support is included. Only the start-up phase is done in polling mode. The interrupts will be activated after the device has been fully configured
- Hardware recognition like PCI scanning routines are not included
- On *Big Endian* CPUs, the user application will need to convert communication channel and send/receive packet content to/from *Little Endian* representation.
  This is NOT automatically done inside the toolkit.
  Only device global data from the system channel are converted by the toolkit.
- The sample project, created for Win32, does not allow PCI cards (CIFX50 / CIFX90 etc.) being completely restarted (Hardware Reset), because PCI registers are not accessible from a Win32 user application.

2 How to port the cifX Toolkit

This is a short instruction on how to port the *cifX Toolkit* to an own embedded system. In general the Toolkit is independent of any operating system and can be used with or without an operating system and it is scalable.

The Toolkit can be ported to use the whole functionalities with inter-process synchronization, interrupts, multi device support, automatic firmware and configuration download etc. or just using the low-level device functions to access a physical dual port memory offered by netX based hardware.

The Toolkit can be used for the following solutions:

- Creating a function library for embedded Systems offering the CIFX API
- Creating an operating system based device driver (e.g. Windows, Linux, VxWorks) offering the CIFX API
- Creating a solution for a Microcontroller based host system using just the Low-Level dual port memory access functions to a netX based hardware

Depending on the solution, the available functionalities may be more or less complex.

Some example implementations are already available (Windows / MQX / none-OS) showing the work to be done to port the Toolkit to an own hardware platform.

Also a Low-Level DPM function example is available showing the use of the Toolkits Low-Level device functions.

2.1 General Procedure

This chapter describes the general handling to port the Toolkit to an own platform.

Basics:

There are two different types of devices being handled by the Toolkit:

- FLASH-based devices (like a comX) which have their firmware stored in a flash
- RAM-based devices (like a cifX50) which get their firmware loaded by the driver / toolkit.

Depending on the type of device, the toolkit has different initialization and start-up functions to get the netX hardware up and running.

Stub out Toolkit functions not necessary for the target:

To stub out a function means implementing a function to always return success (e.g. returning a valid handle or returning a successful wait for timeout).

This means, the functions are still called in the toolkit handling progress but the function return values are evaluated by the toolkit without an error and therefore the Toolkit will keep working.

This is valid for all USER_ and OS_ functions which must be implemented for the target system.

Example: “Stub out” the OS_Mutex function:

```
/*****************************************************************************/
/*! Create an Mutex object for locking code sections
*   \return handle to the mutex object                                       */
/*****************************************************************************/
void* OS_CreateMutex(void)
{
  return (void*)0x12345678;
}

/*****************************************************************************/
/*! Wait for mutex
*   \param pvMutex    Handle to the Mutex locking object
*   \param ulTimeout  Wait timeout
*   \return !=0 on succes                                                    */
/*****************************************************************************/
int OS_WaitMutex(void* pvMutex, uint32_t ulTimeout)
{
  return 1;
}

/*****************************************************************************/
/*! Release a mutex section
*   \param pvMutex Handle to the locking object                              */
/*****************************************************************************/
void OS_ReleaseMutex(void* pvMutex)
{
  return;
}

/*****************************************************************************/
/*! Delete a Mutex object
*   \param pvMutex Handle to the mutex object being deleted                  */
/*****************************************************************************/
void OS_DeleteMutex(void* pvMutex)
{

}
```


2.1.1 Step-by-Step Guide - What needs to be done

- Copy the Source Folder (which contains the whole Toolkit) to your project.
- Implement the OS Abstraction layer (according to the toolkit documentation) in an own / separate C-file.

  You may take a look at *"OSAbstraction\OS_Win32.c"* to see how this is done under Windows. You don't need to implement all functions, depending to your “Use Case”

  Options:

  1. When not using cifX PCI cards or any other RAM-based device with the netX directly connected to the PCI bus, you can stub out the functions *OS_ReadPCIConfig()* / *OS_WritePCIConfig())*
  2. When **not** using Interrupt you can stub out the “Event” functions (*OS_CreateEvent()* / *OS_SetEvent()*, *OS_ResetEvent()* / *OS_DeleteEvent()* / *OS_WaitEvent()*)
  3. If you **don't** have a multitasking environment you can stub out the “Mutex” functions (*OS_CreateMutex()* / *OS_WaitMutex()* / *OS_ReleaseMutex()* / *OS_DeleteMutex()*), as the mutexes are only used to prevent re-entrant function calls.

     Note: As the “Mutexes” are expected to work, the toolkit does not know about your O/S you will need to return a value != 0 out of *OS_CreateMutex ()* and *OS_WaitMutex()*.

     Attention: Doing this in a multitasking environment will result in undefined behavior as function re-entrancy cannot be controlled.
  4. If you only have a comX or another netX with flashed firmware and f you don't want to use the automatic file download / update feature of the toolkit which checks and updates the Firmware during system start-up, you may stub out the “File” functions (*OS_FileOpen()* / *OS_FileRead()* / *OS_FileClose()*) too

     Attention: When using RAM-based devices these functions **must** be implemented.

- Implement the USER functions in an own / separate C-file.
- You may take a look at *"User\TKitUser.c"* to see how this is done under Windows.

  Options:
  1) If you only have a comX or another netX hardware with flashed firmware you may stub out the firmware / bootloader functions

  - *USER_GetOSFile() / USER_GetBootloaderFile()*)

  - *USER_GetFirmwareFileCount() / USER_GetFirmwareFile()*

  - *USER_GetConfigurationFileCount() / USER_GetConfigurationFile()*

  If you don't want to use the automatic update feature of the toolkit, which checks and updates the Firmware during start-up.

  Attention: When using RAM-based devices these functions **must** be implemented.

- Implement a cyclic timer (e.g. 500ms) which calls the function cifXTKitCyclicTimer(). This is needed if any of your devices is used in polling mode (not necessary if all devices are used in interrupt mode).
- Call the Toolkit initialization function *cifXTKitInit()*) from your application or driver framework
- Add all your netX / cifX / comX devices under Toolkit control by:

  1. Allocate a *DEVICEINSTANCE* structure

  2. Filling in all needed parameters into the *DEVICEINSTANCE* structure

  Note 1: You can use the element pvOSDependent to store any user parameter (non-toolkit parameters) for each device and use the information in the USER or OS dependent functions

  Note 2: You can override the type of the device by adjusting the element *eDeviceType* if it is not correctly auto-detected by the toolkit.

  COMX Example:
```
OS_Memset(ptDevInstance, 0, sizeof(*ptDevInstance));
ptDevInstance->fPCICard      = 0;
ptDevInstance->pbDPM         = <Insert pointer to DPM here>;
ptDevInstance->ulDPMSize     = <Insert accessible size of DPM here>;
OS_Strncpy(ptDevInstance->szName, "cifX0", sizeof(ptDevInstance->szName));
ptDevInstance->pvOSDependent = MyDeviceData;
```
  CIFX Example:
```
OS_Memset(ptDevInstance, 0, sizeof(*ptDevInstance));
ptDevInstance->fPCICard      = 1;
ptDevInstance->pbDPM         = <Insert pointer to DPM here>;
ptDevInstance->ulDPMSize     = <Insert accessible size of DPM here>;
OS_Strncpy(ptDevInstance->szName, "cifX0", sizeof(ptDevInstance->szName));
ptDevInstance->pvOSDependent = MyDeviceData;
```
  3. Call *cifXTKitAddDevice()* to add them under Toolkit control

- Now you can use any of the cifX API functions to access your devices

2.1.2 Additional Toolkit Functions and Options

- **Optional:** big-endian CPU support:

  You will need to enable big-endian support in the toolkit by setting the pre-processor definition "*CIFX_TOOLKIT_BIGENDIAN*", which instructs the toolkit to convert DPM access endianness.

  Attention: The toolkit will not swap packet data contents or I/O data as it does not know the structured data behind these data areas. So the user has to do the endianess conversion before calling *xChannelPutPacket()* / *xChannelIOWrite()* and after *xChannelGetPacket()* / *xChannelIORead()* calls. Same is valid for system device and some other block access functions (e.g. extended status block).

  See section *Big Endian Support* on page 25 for more information.

- **Optional:** Use DMA on PCI devices

  Attention: This is only supported if the netX is directly connected to the PCI Bus (e.g. cifX). It does not work with NXPCA-PCI boards (or any other PCI<-->DPM Bridge)

  To use DMA you will need to do the following:

  1. Insert the pre-processor define "*CIFX_TOOLKIT_DMA*"

  2. Pass 8 DMA buffers which need to be aligned on a 256 byte boundary. These buffers must be a multiple of 256 Bytes in size with a maximum size of 63.75kB

  DMA Example:

```
ptDevInstance->ulDMABufferCount = 8;
ptDevInstance->atDmaBuffers[0].ulSize             = 8192;
ptDevInstance->atDmaBuffers[0].ulPhysicalAddress  = <Insert phys. address here>;
ptDevInstance->atDmaBuffers[0].pvBuffer           = <Insert virtual / cpu
   accessible pointer here>;
ptDevInstance->atDmaBuffers[0].pvUser             = MyDMAData;
...
ptDevInstance->atDmaBuffers[7].ulSize             = 8192;
ptDevInstance->atDmaBuffers[7].ulPhysicalAddress  = <Insert phys. address here>;
ptDevInstance->atDmaBuffers[7].pvBuffer           = <Insert virtual / cpu
   accessible pointer here>;
ptDevInstance->atDmaBuffers[7].pvUser             = MyDMAData;
```

  See section *DMA handling for I/O data transfers* on page *34* for more information.

- **Optional:** Dual Port Memory access via custom hardware access interface

  The Dual-Port-Memory access functions (read / write) can be exchange by customer specific functions. An example on how this can be done is shown in an example where the memory access is done via an SPI interface.

  See chapter *Toolkit low-level hardware access functions* on page 99 for more information.

- **Optional:** Extended toolkit function parameter checking

  By default, the toolkit functions are only doing a minimal parameter checking (e.g. no NULL pointer checking). This can be changed toolkit by setting the pre-processor definition "*CIFX_TOOLKIT_PARAMETER_CHECK*"

  See chapter *Extended parameter check of Toolkit functions* on page 36 for more information.

- **Optional:** Device time setting during start-up

  The toolkit offers an option to set the device time during device start-up. This is handled after a firmware start and if the device firmware signals a time handling feature.

  The device time setting is enabled by setting the pre-processor definition "*CIFX_TOOLKIT_TIME*"

2.1.3 Creating an own Device Driver

Creating an operating system dependent device driver is a special case of using the Toolkit inside of such a device driver.

A device driver has to follow the implementation guidelines of an operating system on one side and has to expose the Hilscher CIFX API functions on the other side, to enable user applications to work with netX based hardware.

The main task of a driver would be collecting the netX hardware resource information, initializing the toolkit using this information and create the connection between the internal CIFX API functions in the toolkit to a function interface usable by a user application.

The general procedure would also be the porting of the Toolkit to the target system (like described earlier in this chapter) and calling the Toolkit global functions (e.g. cifXTKitInit() / cifXTKitDeinit() etc.), usually called in a *Main()* function from an application, somewhere in the context of a device driver.

- The Toolkit global function definitions can be found in cifXToolkit.h

```
/* Toolkit Global Functions */
int32_t cifXTKitInit              (void);
void    cifXTKitDeinit            (void);
int32_t cifXTKitAddDevice         (PDEVICEINSTANCE ptDevInstance);
int32_t cifXTKitRemoveDevice      (char* szBoard, int fForceRemove);
void    cifXTKitDisableHWInterrupt(PDEVICEINSTANCE ptDevInstance);
void    cifXTKitEnableHWInterrupt (PDEVICEINSTANCE ptDevInstance);
void    cifXTKitCyclicTimer       (void);
```

- The Hilscher CIFX API function definitions can be found in cifX_USER.h)

2.2 Creating an Application using the Toolkit Low-Level DPM Functions

Another use case of the Toolkit could be a very small Microcontroller based platform which should be extended by a netX and where access to the netX hardware dual port memory (DPM), with its Hilscher default memory layout, is necessary.

The CIFX Toolkit offers also low-level netX DPM access functions (so called DEV_ functions). These functions can be used without an operating system and where only generic access to one netX DPM is necessary. The only requirement, which is necessary to use the DEV functions, is the initialization of some pre-defined data structures with the netX hardware dependent information like DPM address, DPM size and so on.

Note: The CIFX API functions (e.g. *xChannelOpen()*) are not available when using the Toolkit low-level device functions

Note: See section *Toolkit low-level hardware access functions* on page 99 for a detailed description on how to use these functions

The following example shows the usage of the Toolkit DEV_ functions in such an environment.

Usage of the Toolkit DEV_ functions:

```
/*****************************************************************************/
/*! Hardware function example
 * \return 0 on success                                                      */
/*****************************************************************************/
int32_t cifXHWSample( void)
{
  int32_t  lDemoRet  = DEV_NO_ERROR;
  int32_t  lRet      = CIFX_NO_ERROR;

  uint8_t*          pbDPM         = NULL;       /* This pointer must be loaded to the DPM address */
  uint32_t          ulDPMSize     = 0;          /* Size of the DPM in bytes */
  DEVICEINSTANCE    tDevInstance;               /* Global deveice data structure used by all DEV_xxx functions */

  /* Get pointer to the hardware dual-port memory and check if it is available */
  if ( FALSE == cifXTkHWFunctions_GetDPMPointer( &pbDPM, &ulDPMSize))
    /* Failed to get the hardware DPM pointer and size */
    return -1;

#ifdef CIFX_TOOLKIT_HWIF
  tDevInstance.pfnHwIfRead  = cifXHwFnRead;    /* relizes read access to the system dependant DPM interface  */
  tDevInstance.pfnHwIfWrite = cifXHwFnWrite;   /* relizes write access to the system dependant DPM interface */
#endif

  /* Initialize the necessary data structures */
  if ( DEV_NO_ERROR == cifXTkHWFunctions_InitializeDataStructures( pbDPM, ulDPMSize, &tDevInstance, 10000))
  {
    /*----------------------------------------------------------------*/
    /* Read actual device states                                      */
    /*----------------------------------------------------------------*/
    PCHANNELINSTANCE ptSystemDevice = &tDevInstance.tSystemDevice;
    PCHANNELINSTANCE ptChannel      = tDevInstance.pptCommChannels[COM_CHANNEL];

    /* Wait for State acknowlede by the firmware */
    OS_Sleep(100);              /* Wait a bit */

    /* read the host flags of the system device, first time to synchronize our internal status */
    DEV_ReadHostFlags( ptSystemDevice, 0);

    /* read the host flags of the communication channel, first time to synchronise our internal status */
    DEV_ReadHostFlags( ptChannel, 0);

    /* check if "system device" is ready... */
    if (!DEV_IsReady( ptSystemDevice))
    {
      /* System device is not ready! */
      lDemoRet = ERR_DEV_SYS_READY;

    /* check if "communication channel" is ready... */
    } else if ( !DEV_IsReady(ptChannel))
    {
      /* Communication channel is not ready! */
      lDemoRet = ERR_DEV_COM_READY;
```

```
    } else
    {
      /*---------------------------------------------------------------------*/
      /* At this point we should have a running device and a configured         */
      /* communication channel.                                                  */
      /* Procced with "NORMAL Stack Handling!                                    */
      /*---------------------------------------------------------------------*/
      /* Signal Host application is available */
      lRet = DEV_SetHostState( ptChannel, CIFX_HOST_STATE_READY, 1000);

      /* Configure the device */
      lDemoRet = cifXTkHWFunctions_ConfigureDevice( ptChannel, ptSystemDevice);
      //if( DEV_NO_ERROR != lDemoRet)
      //  printf("Error");

      /* Initialize and activate interrupt if configured */
      DEV_InitializeInterrupt ( &tDevInstance);

      if (DEV_NO_ERROR == lDemoRet)
      {
        /*---------------------------------------------------------------------*/
        /* At this point we should have a running device and a configured         */
        /* communication channel if no error is shown                             */
        /*---------------------------------------------------------------------*/
        uint32_t ulState = 0;

        /* Signal Host application is available */
        lRet = DEV_SetHostState( ptChannel, CIFX_HOST_STATE_READY, 1000);

        /* Switch ON the BUS communication */
        lRet = DEV_BusState( ptChannel, CIFX_BUS_STATE_ON, &ulState, 3000);

        /* TODO: Decide to wait until communication is available or just go to */
        /*       to the cyclic data handling and check the state there         */
        /* Wait for communication is available or do this during the cyclic program handling*/
        lDemoRet = cifXTkHWFunctions_WaitUntilCommunicating( ptChannel);

        /*---------------------------------------------------------------------*/
        if (lDemoRet == DEV_NO_ERROR)
        {
          /* device is "READY", "RUNNING" and "COMMUNICATING" */
          /* Start cyclic demo with I/O Data-Transfer and packet data transfer */
          unsigned long ulCycCnt = 0;
          //uint32_t ulTriggerCount = 0;

          /* Cyclic I/O and packet handling for 'ulCycCnt'times */
          while( ulCycCnt < DEMO_CYCLES)
          {
            /* Start and trigger watchdog function, if necessary */
            //DEV_TriggerWatchdog(ptChannel, CIFX_WATCHDOG_START, &ulTriggerCount);

            /* Handle I/O data transfer */
            IODemo      (ptChannel);

            /* Handle rcX packet transfer */
            #ifdef FIELDBUS_INDICATION_HANDLING
              Fieldbus_HandleIndications( ptChannel);
            #else
              PacketDemo ( ptChannel);
            #endif

            ulCycCnt++;
          }

          /* Stop watchdog function, if it was previously started */
          //DEV_TriggerWatchdog(ptChannel, CIFX_WATCHDOG_STOP, &ulTriggerCount);
        }

        /* Switch OFF the BUS communication / dont't wait */
        lRet = DEV_BusState( ptChannel, CIFX_BUS_STATE_OFF, &ulState, 0);

        /* Signal Host application is not available anymore / don't wait */
        lRet = DEV_SetHostState( ptChannel, CIFX_HOST_STATE_NOT_READY, 0);
      }

      /* Uninitialize interrupt */
      DEV_UninitializeInterrupt ( &tDevInstance);
    }
  }

  /* Cleanup all used nenory areas and pointers */
  cifXTkHWFunctions_UninitializeDataStructures( &tDevInstance);

  /* cifXTkHWFunctions cleanup */
  cifXTkHWFunctions_FreeDPMPointer( pbDPM);

  return lDemoRet;
}
```

Note: The complete example can be found on the toolkit CD.

3 How to Access Serial DPM via SPI

The serial DPM connection is realized by a proprietary protocol, converting parallel read and write accesses into serialized commands streams transfered via a standard SPI master controller. The Toolkit comes with a target independent function module (*Serial DPM Interface*) which implements the proprietary serial DPM protocol for any available netX derivate. This function module becomes part of the toolkit software architecture by implementing the "Custom Hardware Access Interface", which replaces the direct memory accesses (default handling for parallel DPM) with customized routines for serial DPM access. As handling of the SPI master controller differs highly with regard to hardware and operating system, the user has to implement a small set of target specific access routines to perform a raw data transfer according to specification of the used SPI controller.

Note: A general description of the Custom Hardware Access Interface is given in section *Custom hardware access interface / Serial DPM* on page 39 of this manual.

The use of the Serial DPM Interface neither requires any deeper knowledge about the proprietary serial DPM protocol nor a complete insight into the concept of the “Custom Hardware Access Interface”. While this is the most convenient way of getting started with a Serial DPM based scenario, a target specific implementation of the serial DPM protocol may offer improvements in terms of execution performance and code size.

Note: A Getting Started Guide: *Serial Dual-Port Memory Interface with netX [6]* including hardware interface specification, detailed protocol description and some basic examples is provided on the Toolkit CD (NXDRV-TKIT).

Block Diagram:

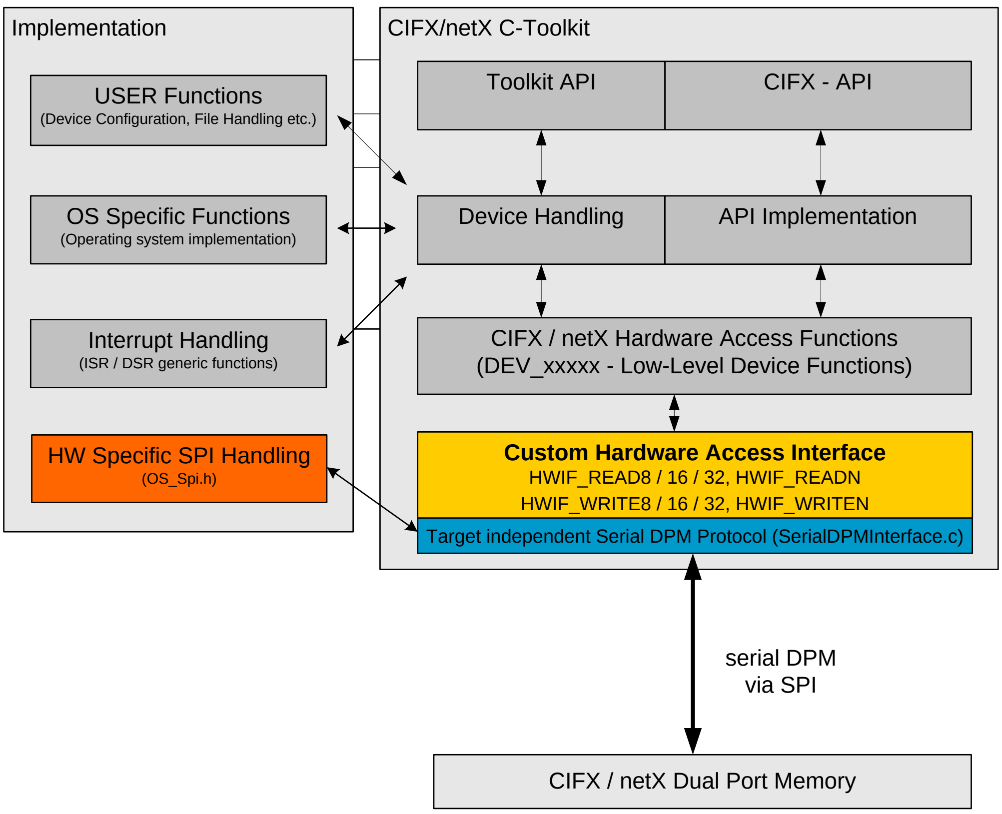

Figure 2: Block Diagram: Custom Hardware Access Interface

3.1 Serial DPM Interface Functions

The Serial DPM Interface functions are divided into two parts:

- Serial DPM Interface Initialization
  Serial DPM Interface requires initialization before passing control to general toolkit functions
- OS/HW specific SPI access functions
  To keep the Serial DPM Interface independent of the SPI hardware, the user needs to implement a basic set of SPI access functions

3.1.1 Serial DPM Interface Initialization

The initialization of the serial DPM interface must be done prior to passing the device to toolkit control (via *cifXTKitAddDevice()*). The initialization includes auto-detection of the connected serial DPM device and populating the toolkit's device instance structure according to the connected netX chip type:

- Assign pointer to Hardware Access Function (*pfnHwIfRead* and *pfnHwIfWrite*)
- Adjust DPM pointer *pbDPM* to zero (Serial DPM is accessed via offset into DPM)
- Set the *fPCICard* flag to FALSE

Note: The user application is still expected to correctly initialize the remaining elements of the device structure (e.g. access name, interrupt number).

Function Call

```
int SerialDPM_Init ( DEVICEINSTANCE* ptDevice);
```

Arguments

Argument	Data type	Description
ptDevice	DEVICEINSTANCE *	Toolkit device instance

Return Values

Return Values	
SERDPM_NETX10	netX10 based serial DPM is connected
SERDPM_NETX50	netX50 based serial DPM is connected
SERDPM_NETX51	netX51, netX52, and netX90 based serial DPM is connected
SERDPM_NETX100	netX100 based serial DPM is connected
SERDPM_UNKNOWN	Serial DPM device is not connected

3.1.2 SPI Access Functions

As SPI handling itself relies highly on hardware platform and operating system environment, the user has to provide a hardware/operating system specific implementation of a small set of SPI access functions.

SPI Access Functions	Description
OS_SpiInit	Initialize SPI components (e.g. driver)
OS_SpiAssert	Assert the chip select line
OS_SpiDeassert	Deassert the chip select line
OS_SpiLock	Lock the SPI bus
OS_SpiUnlock	Unlock the SPI bus
OS_SpiTransfer	Perform SPI transfer

Table 4: SPI Access Functions

3.1.2.1 OS_SpiInit

Initializes the components required to handle SPI transfers (e.g. drivers). This function returns CIFX_NO_ERROR on success.

Function Call

```
long OS_SpiInit (void* pvOSDependent)
```

Arguments

Argument	Data type	Description
pvOSDependent	void*	Device-specific parameter passed with toolkit initialization

3.1.2.2 OS_SpiAssert

Asserts the chip select line which is connected to the netX serial DPM slave device. The serial DPM requires a falling edge of the chip select signal to initiate a read or write process.

Function Call

```
void OS_SpiAssert (void* pvOSDependent)
```

Arguments

Argument	Data type	Description
pvOSDependent	void*	Device-specific parameter passed with toolkit initialization

3.1.2.3 OS_SpiDeassert

Deasserts the chip select line which is connected to the netX serial DPM slave device. The end of a transaction on the netX serial DPM is signaled via a rising edge of the chip select signal.

Function Call

```
void OS_SpiDeassert (void* pvOSDependent)
```

Arguments

Argument	Data type	Description
pvOSDependent	void*	Device specific parameter passed with toolkit initialization

3.1.2.4 OS_SpiLock

Locks the SPI bus to deny parallel access to the bus.

Function Call

```
void OS_SpiLock (void* pvOSDependent)
```

Arguments

Argument	Data type	Description
pvOSDependent	void*	Device-specific parameter passed with toolkit initialization

3.1.2.5 OS_SpiUnlock

Unlocks the SPI bus.

Function Call

```
void OS_SpiUnlock (void* pvOSDependent)
```

Arguments

Argument	Data type	Description
pvOSDependent	void*	Device-specific parameter passed with toolkit initialization

3.1.2.6 OS_SpiTransfer

Initiates a data transfer with the netX serial DPM. Data bytes in the send buffer are clocked out to the serial DPM, while received bytes are stored in the receive buffer. Consider that send and receive buffers are optional, thus the routine must be capable of sending dummy bytes (in case pbSend == NULL) and discard receive bytes (if pbRecv == NULL). The caller may not pass any buffer at all, to initiate an idle transfer (protocol dependent wait cycles).

Function Call

```
void OS_SpiTransfer (void* pvOSDependent, uint8_t* pbSend,
                     uint8_t* pbRecv, uint32_t ulLen)
```

Arguments

Argument	Data type	Description
pvOSDependent	void*	Device-specific parameter passed with toolkit initialization
pbSend	uint8*	Send buffer
pbRecv	uint8*	Receive buffer
ulLen	uint32_t	Length of SPI transfer

3.2 Example

The following example shows the usage of the Serial DPM Interface:

```
#include <cifXToolkit.h>
#include <CIFXErrors.h>
#include <SerialDPMInterface.h>
#include <OS_Spi.h>

/* Toolkit device instance */
static DEVICEINSTANCE s_tDevInstance = {.pvOSDependent = &s_tDevInstance,
                                        .ulDPMSize     = 0x10000,
                                        .szName        = "cifX0"};

/*****************************************************************************/
/*! Assert chip select
*   \param pvOSDependent OS Dependent parameter                             */
/*****************************************************************************/
void OS_SpiAssert(void* pvOSDependent)
{
  /* HW/OS specifc implementation to access SPI bus */
}

/*****************************************************************************/
/*! Deassert chip select
*   \param pvOSDependent OS Dependent parameter                             */
/*****************************************************************************/
void OS_SpiDeassert(void* pvOSDependent)
{
  /* HW/OS specifc implementation to access SPI bus */
}

/*****************************************************************************/
/*! Transfer byte stream via SPI
*   \param pvOSDependent OS Dependent parameter
*   \param pbSend        Send buffer (Can be NULL for polling data from slave)
*   \param pbRecv        Receive buffer (Can be NULL if slaves received data
                         is discarded by caller)
*   \param ulLen         Length of SPI transfer                             */
/*****************************************************************************/
void OS_SpiTransfer(void* pvOSDependent, uint8_t* pbSend, uint8_t* pbRecv, uint32_t ulLen)
{
  /* HW/OS specifc implementation to access SPI bus */
}

/*****************************************************************************/
/*! Serial DPM Example                                                      */
/*****************************************************************************/
void SerialDPM_Example( void)
{
  int32_t lTkRet = CIFX_NO_ERROR;

  /* First of all initialize toolkit */
  lTkRet = cifXTKitInit();

  if(CIFX_NO_ERROR == lTkRet)
  {
    int iSerDPMType;

    if (SERDPM_UNKNOWN == (iSerDPMType = SerialDPM_Init(&s_tDevInstance)))
    {
      /* Serial DPM protocol could not be recognized! */
    } else
    {
      /* iSerDPMType contains connected netX chip type */

      /* Add the device to the toolkits handled device list */
      lTkRet = cifXTKitAddDevice(&s_tDevInstance);

      /* If it succeeded do device tests */
      if(CIFX_NO_ERROR != lTkRet)
      {
        /* Uninitialize Toolkit, this will remove all handled boards from the toolkit and
           deallocate the device instance */
        cifXTKitDeinit();
      } else
      {
        /* Start working with cifX API */
      }
    }
  }
}
```


4 The cifX/netX Toolkit

The toolkit consists of several C modules and header files which offer abstract access to the cifX dual ported memory (DPM). All functions known from the cifX driver are made available in the toolkit. Also the underlying hardware access functions are included.

4.1 Directory Structure and Content

4.1.1 cifX Toolkit CD

CD Content

Directory	Contents
cifXToolkit	Operating system independent C source code of the toolkit (see above)
Documentation	All documents available with the toolkit
Examples	Example implementation of the toolkit source for different operating systems

Table 5: Toolkit Directory Structure

4.1.2 cifXToolkit

Directory	Contents
This directory contains the cifX Toolkit C source code	
BSL	Example Second Stage Boot Loader, necessary for none FLASH-based hardware (e.g. CIFX50)
Common	Common header files used by the toolkit.
Source	All toolkit header files and C-modules
OSAbstraction	Operating system abstraction layer used by the toolkit. **Note:** This needs to be implemented by the user.
User	C-Modules that need to be implemented by the user for the toolkit to work properly. E.g. Passing bootloader / firmware and configuration files to the toolkit functions
SerialDPM	Target independent SPI protocol implementation
doxygen	Doxygen components, to create an internal documentation of the toolkit
Doc	A doxygen generated documentation of the toolkit

Table 6: Toolkit Directory Structure - cifXToolkit

4.1.3 Documentation

Directory	Contents
cifX netX Toolkit - DPM TK xx EN.pdf	This documentation
Second Stage Bootloader netX.pdf	Description of the netX bootloader functions
netX Dual-Port Memory Interface DPM xx EN.pdf	Description of the netX default dual port memory interface
CIFX API PR xx EN.pdf	Description of the CIFX API
Error Codes EN xx.pdf	Error code summary (Driver/Toolkit, Firmware, Protocol Stacks)
cifX netX Application Programmers Guide xx EN.pdf	Programmers introductions
.\SerialDPM	
Serial DPM interface with netX GS xx EN.pdf	Getting started with netX serial DPM
netX 51 52 Programming Reference Guide PRG xx EN.pdf	netX51/52 programming reference guide
netX10_Technical_Reference_Guide_xx.pdf	netX10 technical reference guide
netX50_Program_Reference_Guide_Recxx.pdf	netX50 programming reference guide
SPI_Slave_DPM_netX_100_500_HAL_xx_EN.pdf	netX100/500 SPI Slave interface as DPM

Table 7: Toolkit Directory Structure - Documentation

4.1.4 Examples\cifXToolkit

<table>
<tr><th>Directory</th><th colspan="2">Contents</th></tr>
<tr><td colspan="3">CIFX Toolkit implementation examples for different operating systems.
Including example source code to exchange the parallel DPM access functions by serial DPM functions (SPI host examples).</td></tr>
<tr><td rowspan="2">nonOS</td><td colspan="2">Operating system independent implementation including SPI Host functions</td></tr>
<tr><td>\netX</td><td>None OS based example for the
netX10 / 50 / 100 ARM based controllers
Note: Only SPI Host example implementation available</td></tr>
<tr><td>rcX</td><td colspan="2">Implementation for the Hilscher rcX RTOS
Note: Only SPI Host implementation available (no parallel DPM functions)
Note: An rcX version must be already available to run the example</td></tr>
<tr><td>Win32</td><td colspan="2">Windows 32Bit implementation (Only running as a USER Mode Application)
Note: Only parallel DPM example implementation available</td></tr>
</table>

Table 8: Toolkit Directory Structure - Examples\cifXToolkit

4.1.5 Examples\cifXTkitHWFunctions

<table>
<tr><th>Directory</th><th>Contents</th></tr>
<tr><td colspan="2">Containing the Low-Level DPM access functions from the toolkit to directly access one netX DPM.
Implementation examples for different operating systems</td></tr>
<tr><td>nonOS</td><td>None OS based example</td></tr>
<tr><td>Win32</td><td>Windows 32Bit example
(Only running as a USER Mode Application . CIFX Device Driver must be installed)</td></tr>
</table>

Table 9: Toolkit Directory Structure - Examples\cifXTKitHWFunctions

4.2 Data Packing

Data structures in the DPM of netX devices and packet based command structures are partially byte aligned. To ensure correct data packing of rcX data structures used in the toolkit, all structures are byte aligned by default.

4.3 Big Endian Support

The *netX Toolkit* supports "*Big Endian*" host systems. This means, the netX toolkit offers a compiler switch to change the default data representation from standard "little endian" to "big endian".

Note: Protocol stacks on the netX are only "Little Endian" aware, because they are execute on a target system which has a little Endian data representation.

Attention: Endianness also depends on the physical access (Byte/Word/DWORD) to the DPM. On systems which are only supporting 16Bit access to peripheral memory (e.g. Freescale MCF51CN128), a Byte access to a 16Bit connected DPM does not result in the expected data of seeing the Byte content in Bit [0:7] of CPU register.

The Toolkit is not aware of such hardware access behaviours and the internal "BIG ENDIAN" macros are not working in such an environment, because there is no "Byte exchange" and DWORD swapping will also deliver wrong results in the CPU registers.

In such an environment use either a 8Bit access mode, change/rewrite the macros and the access to the DPM or use the CIFX_TOOLKIT_HWIF read/write functions to manipulate the resulting data content to have a correct data representation.

The "Big Endian" data representation covers the device initialization and standard informational data structures of a netX based device. This means all functions executed inside of the toolkit and the standard data and information structures, reachable via the "*xSystemdevice*" functions are endianness aware.

All data structures which are protocol dependent (state information / diagnostic data / runtime I/O data / protocol stack specific requests, confirmation, indications etc.) and exchanged between the user application and the protocol stack must be converted by the user application.

Also the packet header of acyclic commands which are exchanged by rcX packets between the hardware and the user application are not converted by the toolkit.

Note: All packets send via *xSysdevicePutPacket()* / *xChannelPutPacket()*, need to be converted by the application in to the little Endian format of the netX device.. Packets which are received via *xSysdeviceGetPacket()*/*xChannelGetPacket()* / *xChannelGetSendPacket()* will have the little Endian format of the netX device and must be converted to big Endian.

Note: Automatic conversion for packets will NOT be available. For samples on how the data conversion can be done, take a look at the toolkit module *cifXEndianess.c*.

"Big Endian" support is enabled by setting the "CIFX_TOOLKIT_BIGENDIAN" define in your project.

```
#define CIFX_TOOLKIT_BIGENDIAN
```

4.4 64-bit support

The toolkit supports 64-bit processor, by using fixed width data types defined in ISO C99 (stdint.h). For Compilers that don't support ISO C99 standard, the developer needs to provide an equivalent header file.

The following data types must be at least present:

Data Type / typedef	Description
signed types	
int8_t	signed 8-bit data type
int16_t	signed 16-bit data type
int32_t	signed 32-bit data type
int64_t	signed 64-bit data type
unsigned types	
uint8_t	unsigned 8-bit data type
uint16_t	unsigned 16-bit data type
uint32_t	unsigned 32-bit data type
uint64_t	unsigned 64-bit data type

Further documentation of this header file can be found here:

http://en.wikipedia.org/wiki/Stdint.h

4.5 FLASH-based vs RAM-based devices

A general definition for using netX-based devices is the device type *eDeviceType* defined in the *DEVICEINSTANCE* structure. This type defines whether the device is a RAM-based or FLASH-based device and therefore the general handling in the cifX toolkit.

Device Type Definition:

- **RAM-based Device**
  For RAM-based devices, the firmware and the configuration files are **not** stored on the hardware. On each power-up sequence, all executables have to be loaded to the hardware in order to get the hardware running. Therefore the user application or device drivers have to provide the firmware and configuration files at start-up time. Example: Most PC card CIFX and PC card CIFX express are RAM-based devices.
- **FLASH-based Device**
  For a Flash-based device, the firmware and the configuration are stored in a local Flash chip. If the power supply is switched on, the device starts, loads and executes the stored firmware. User provided firmware and configuration files are not always downloaded to the hardware, to protect the live time of the Flash, instead the file internal MD5 checksums are verified and only downloaded to the hardware if they are different. Example: COMX modules, netIC modules and CIFX4000 are Flash-based devices.

Note: netX90/netX4000 are Flash-based devices too, but they are not handled the same way as described above. The handling is: A firmware is never downloaded automatically even if the checksums are different. A firmware download always has to be initiated by the user.

Device Type	Value	Description
eCIFX_DEVICE_AUTODETECT	0	Autodetection: fPCIcard = 1 => RAM based device Cookie available => FLASH based device Cookie not available => RAM based device
eCIFX_DEVICE_RAM_BASED	2	Handle device as a RAM based device
eCIFX_DEVICE_FLASH_BASED	3	Handle device as a FLASH based device
eCIFX_DEVICE_DONT_TOUCH	4	Expect the device as up and running **Note:** This setting is only used for debugging purpose, to prevent any changes at the device during startup and expecting an already initialized device.

Table 10: Device types

Drivers and user applications using *eDeviceType* = eCIFX_DEVICE_AUTODETECT only if they want to dynamically detect the correct device handling.

If the device is already defined (like COMX etc.) than the specific device type should be used to make sure the device is handled in the expected way and any malfunction is correctly reported by the toolkit.

4.6 Loadable Firmware Files

The netX Toolkit supports monolithic firmware files (.NXF, .NXI, .NAI) and the usage of loadable modules (.NXO).

A monolithic firmware is one file containing the operating system and one or more communication protocol stacks.

Loadable modules are files that only contain a communication protocol stack without the operating system and the operating system is located in an own file named "Base OS Firmware".

While loadable modules are defined by an own file header and file extension, the base OS module uses the same file header structure and file extension like a monolithic firmware.

File Extension:

File extension		**Identifies**
.NXF	**n**et**X** **F**irmware	Monolithic Firmware / Base OS Firmware
.NXI	**n**et**X** Firmware for Communication CPU (**i**nternal Flash memory)	Monolithic Firmware (netX90/netX4000)
.NAI	**n**etX Firmware for **A**pplication CPU (**i**nternal Flash memory)	Monolithic Firmware (netX90)
.NXO	**n**et**X** Firmware M**o**dule	Loadable Firmware Module

The file header structure definitions can be found in the header file *HilFileHeaderV3.h*, located in the toolkit source directory.

The toolkit allows using the listed types of firmware files.

4.6.1 Initialization process using a monolithic firmware

The following figures show the process of adding a device to the toolkit and the Function Calls being made by the toolkit. Depending on the type of device (RAM based / FLASH based).

There are two major approaches to initializing a card

- The device is FLASH based and will already have all things up and running (e.g. comX)
- The device is RAM only based and must be prepared before it can be used (e.g. cifX PCI cards)

4.6.1.1 Using a RAM-based device

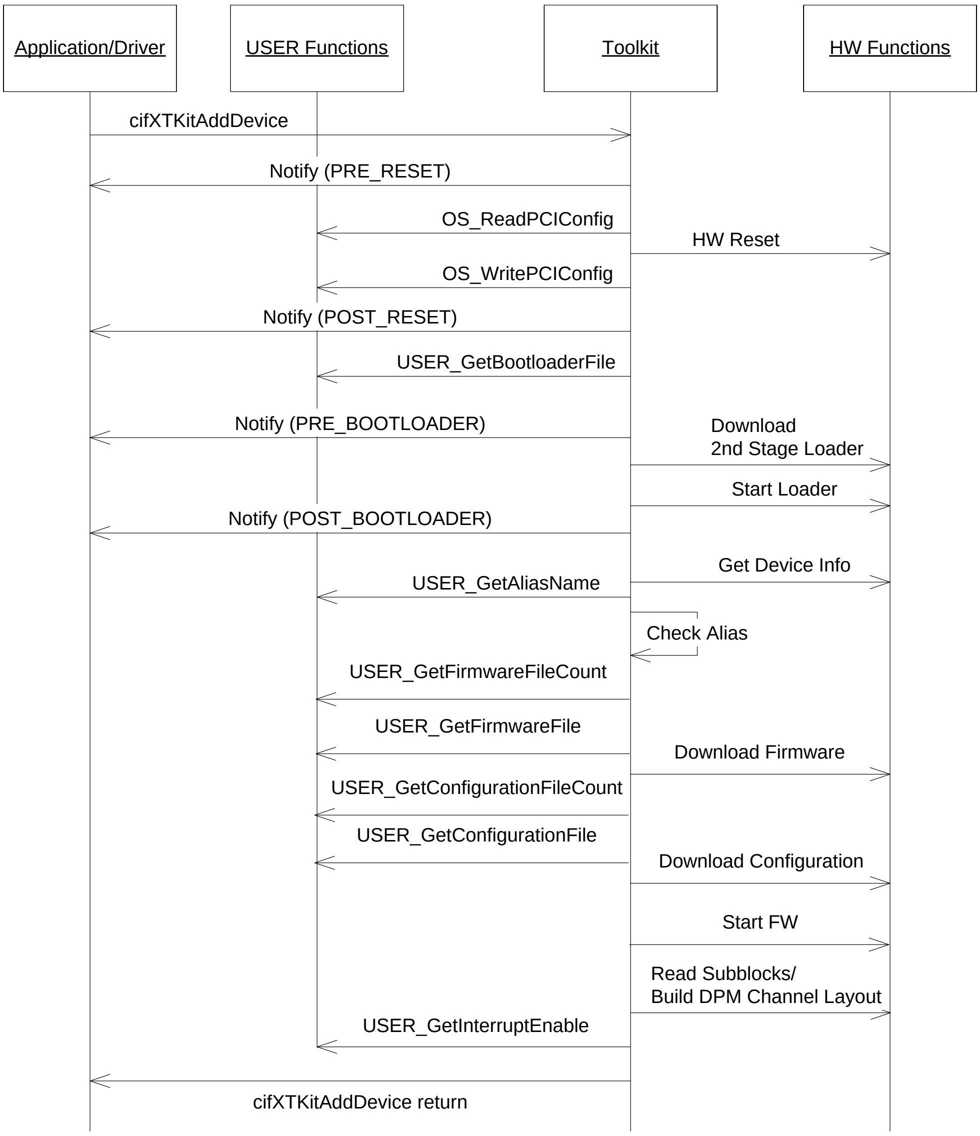

Figure 3: Initialization Sequence of a RAM-based device

4.6.1.2 Using a Flash-based device

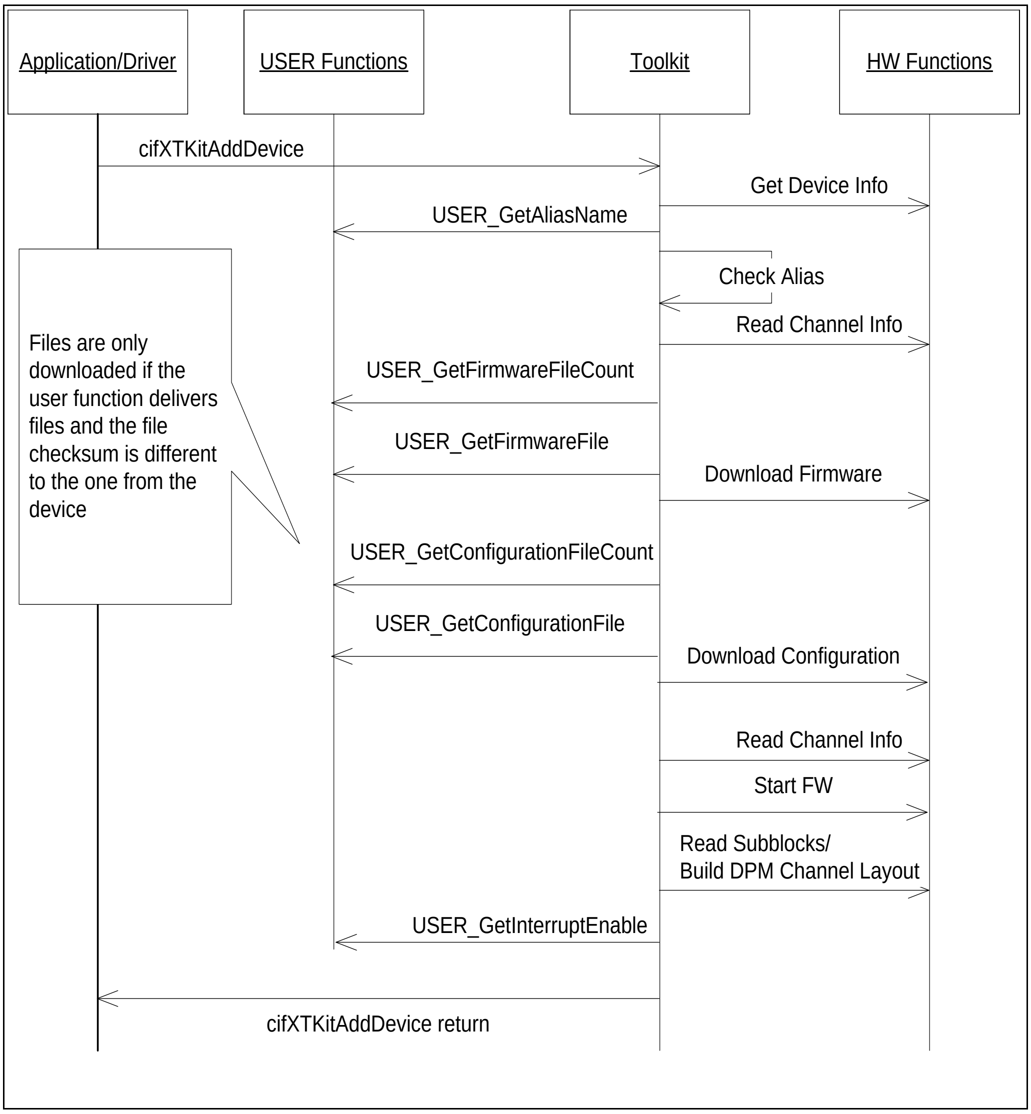

Figure 4: Initialization Sequence of a Flash-based device (firmware already running)

Note: netX90/netX4000 are Flash-based devices too, but the handling differs from the sequence above. The handling for netX90/netX4000 is: A firmware is never downloaded automatically even if the checksums are different. A firmware download always has to be initiated by the user.

4.6.2 Initialization process using Loadable Firmware Modules

The following figures show the process of adding a device to the toolkit and the Function Calls being made by the toolkit. Depending on the type of device (RAM based / FLASH based).

There are two major approaches to initializing a card

- The device is FLASH based and will already have all things up and running (e.g. comX)
- The device is RAM only based and must be prepared before it can be used (e.g. cifX PCI cards)

4.6.2.1 Using a RAM-based device

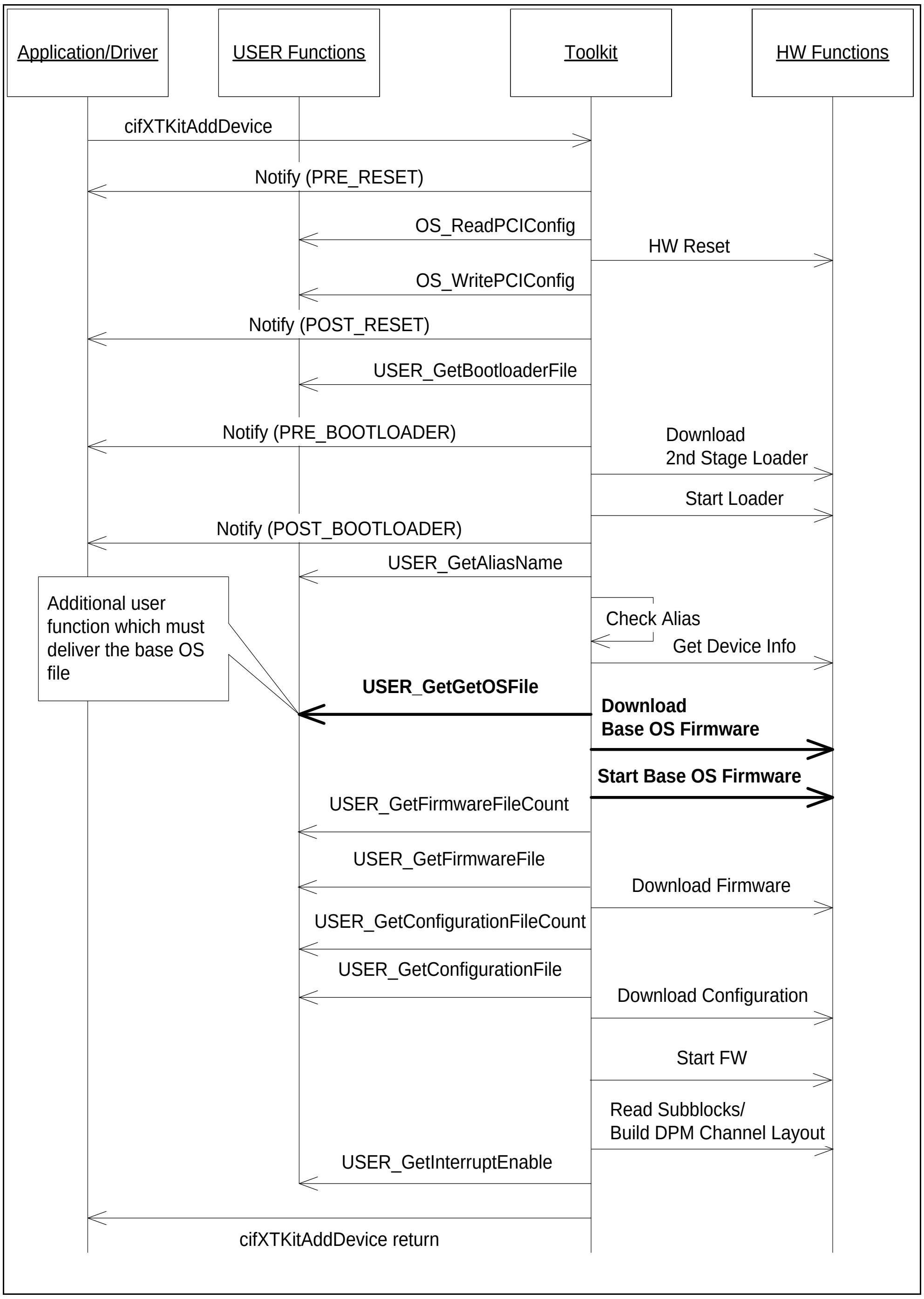

Figure 5: Initialization Sequence of a RAM-based device

4.6.2.2 Using a Flash-based device

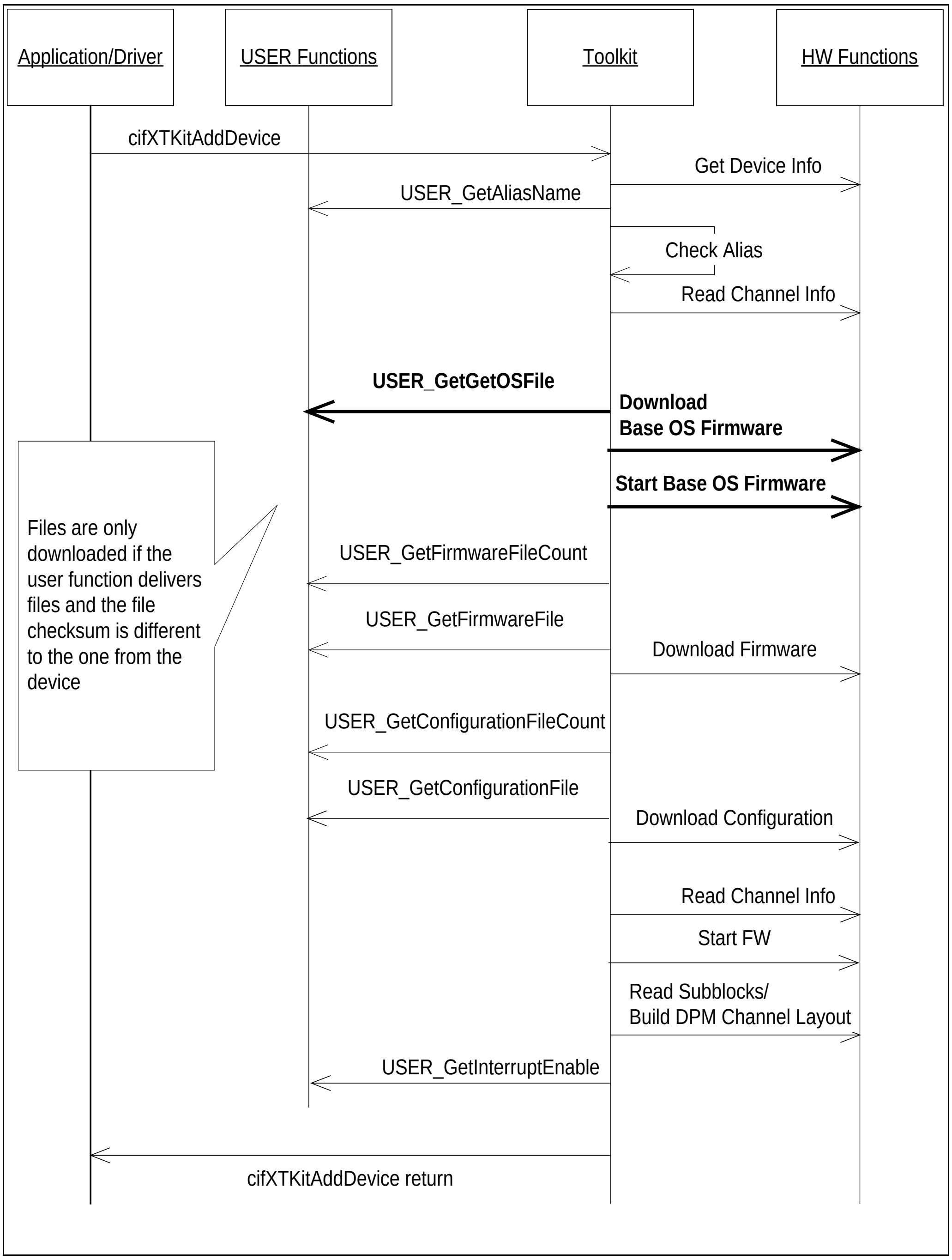

Figure 6: Initialization Sequence of a Flash-based device (Firmware already running)

4.7 Interrupt handling

The interrupt handling in the toolkit is separated into two functions. An ISR (Interrupt Service Routine) function getting the actual interrupt information of the hardware and acknowledges the interrupt and a DSR (Deferred Service Routine) functions which processes the interrupt information.

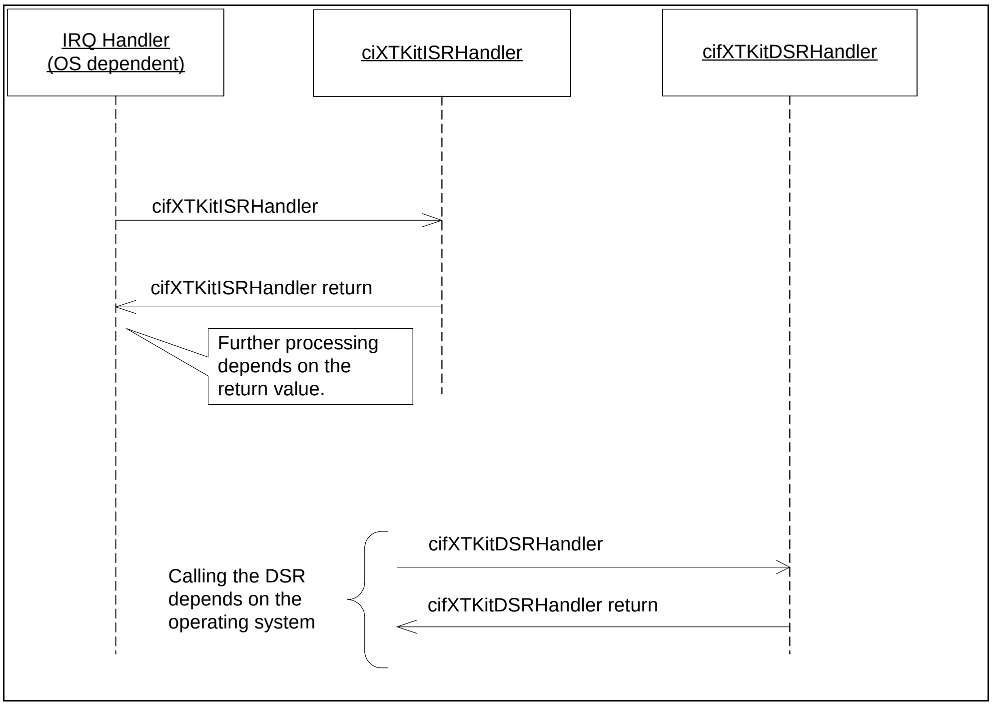

Figure 7: Interrupt handling

The separation is done to support operating systems which expect the implementation of a deferred interrupt handler function to be able to leave the hardware interrupt level which usually does not allow to call operating system specific interprocess communication functions (e.g. Event etc.).

4.8 DMA handling for I/O data transfers

The cifX/*netX Toolkit* supports bus master DMA transfers of I/O image data on PCI cards. This feature must be explicitly enabled through a general toolkit definition in the user project or compiler option. Activating the DMA data transfer expects to definition of the necessary DMA buffers in the *DEVICE_INSTANCE* structure before adding the device to the toolkit

```
#define CIFX_TOOLKIT_DMA
```

Note: DMA handling needs specific hardware/firmware support and toolkit initialization

Note: Only I/O area 0 is supported when DMA is used!

DMA Mode can only be enabled on devices if the netX is directly connected to the PCI Bus (e.g. CIFX-50).

The host needs to provide 8 DMA buffers before adding the device to the toolkit. These buffers are automatically assigned to the appropriate I/O Blocks according to the following table:

Buffer Number	Comm. Channel	Block
0	0	Input Area 0
1	0	Output Area 0
2	1	Input Area 0
3	1	Output Area 0
4	2	Input Area 0
5	2	Output Area 0
6	3	Input Area 0
7	3	Output Area 0

Table 11: DMA buffer sssignment

The user created DMA buffers must meet the following restrictions:

- Aligned on a 256 Byte boundary
- Minimal Size = 256 Byte
- Maximal Size = 63,75 kB
- Size must be a multiple of 256 Bytes
- All 8 Buffers must be supplied, if DMA is to be used
- Buffers must be a continued memory area and non cached

Note: The DMA transfers are always handled and controlled by the netX chip. The transfer is activated by the standard toolkit functions *xChannelORead()* / *xChannelIOWrite()* and transparent to the user application.

Example

```
  /* Initialize the cifX Toolkit */
  int32_t lRet = cifXTKitInit();
  if(CIFX_NO_ERROR == lRet)
  {
    uint32_t ulIdx;
    PDEVICEINSTANCE ptDevInstance =
                  (PDEVICEINSTANCE)OS_Memalloc(sizeof(*ptDevInstance));
    OS_Memset(ptDevInstance, 0, sizeof(*ptDevInstance));
    ptDevInstance->fPCICard           = 1; /* This must be set for DMA */
    ptDevInstance->pvOSDependent      = <insert use specific data>;
    ptDevInstance->pbDPM              = <insert pointer to DPM>;
    ptDevInstance->ulDPMSize          = <insert size of DPM>;
    OS_Strncpy(ptDevInstance->szName,
               "cifX0",
               sizeof(ptDevInstance->szName));
    /* Add DMA Buffers */
    ptDevInstance->ulDMABufferCount = CIFX_DMA_BUFFER_COUNT;
    for(ulIdx = 0; ulIdx < CIFX_DMA_BUFFER_COUNT; ++ulIdx)
    {
      CIFX_DMABUFFER_T* ptDMABuffer = &ptDevInstace->atDmaBuffers[ulIdx];
      ptDMABuffer->ulSize            = <Size of the DMA Buffer>
      ptDMABuffer->ulPhysicalAddress = <Physical address (32Bit) to DMA Buffer>
      ptDMABuffer->pvBuffer          = <Pointer to the DMA Buffer>
      ptDMABuffer->pvUser            = <Insert user specific data>
    }
    /* Add the device to the toolkits handled device list */
    lRet = cifXTKitAddDevice(ptDevInstance);

===> Work with the cifX Driver API
```


4.9 Extended parameter check of Toolkit functions

Several Toolkit API function calls expect valid pointer and handles passed via its parameter list. By default these parameters are not validated by the Toolkit functions, thus it is under the responsibility of the caller that the pointers and handles passed to the Toolkit functions are valid.

The Toolkit provides a feature which enables a simple validation of the pointer parameters, i.e. the function returns with an error (CIFX_INVALID_POINTER) if a NULL pointer is passed to the function. Additional driver, system device and channel handles are validated, i.e. only those handles are accepted which are returned by the appropriate open function call (otherwise returns error CIFX_INVALID_HANDLE). Both features must be explicitly enabled through a general toolkit definition in the user project or compiler option.

```
#define CIFX_TOOLKIT_PARAMETER_CHECK
```

Note:	As the parameter validation has influence on the performance of the function call, time-critical cifX API calls like xChannelIORead() or xChannelPutPacket() are not affected by the parameter validation.

Note:	The predominant majority of invalid pointers are not NULL, thus the simple pointer check provided by the Toolkit does not relieve the caller to supply a reliable memory management.

4.10 Device time setting

The cifX/*netX Toolkit* supports an optional device time setting function. Time setting is handled during the start-up phase of the device (cifXInit.c / cifXStartDevice()). If the firmware is up and running and signals a time handling feature (RTC type != 0 and RTC status = 0), a corresponding time set command is created and send to the devices system channel.

```
#define CIFX_TOOLKIT_TIME
```

The time handling feature of the device is evaluated by *ulHWFeatures* in the NETX_SYSTEM_STATUS_BLOCK.

ulHWFeatures

						RTC			**Extended Memory**								
31..16	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	

Type :
00 = No RTC
01 = RTC internal
10 = RTC external
11 = RTC emulated

Status:
0 = not Set
1 = Set

Unused set to 0

Definitions for ulHWFeatures:

```
/* RTC */
#define RCX_SYSTEM_HW_RTC_MSK                                0x00000700
#define RCX_SYSTEM_HW_RTC_TYPE_MSK                           0x00000300
#define RCX_SYSTEM_HW_RTC_TYPE_NONE                          0x00000000
#define RCX_SYSTEM_HW_RTC_TYPE_INTERNAL                      0x00000100
#define RCX_SYSTEM_HW_RTC_TYPE_EXTERNAL                      0x00000200
#define RCX_SYSTEM_HW_RTC_TYPE_EMULATED                      0x00000300
#define RCX_SYSTEM_HW_RTC_STATE                              0x00000400
```

OS_Time() Function:

To be able to use the time setting feature of the toolkit an *OS_Time()* function must be implemented in the *OS_Abstraction.c* module.

Time Format:

Base Time: **POSIX/UNIX/ISO 8601 = > 01.01.1970 / 00:00:00 (midnight)**

Tick resolution: **“Seconds” since “Base Time”**

Time Command:

```
/*****************************************************************************
 * Packet: RCX_TIME_COMMAND_REQ/RCX_TIME_COMMAND_CNF
 *
 */

/* Time command codes */
#define TIME_CMD_GETSTATE                  0x00000001
#define TIME_CMD_GETTIME                   0x00000002
#define TIME_CMD_SETTIME                   0x00000003

/* Time RTC information */
#define TIME_INFO_RTC_MSK                  0x00000007
#define TIME_INFO_RTC_TYPE_MSK             0x00000003
#define TIME_INFO_RTC_RTC_STATE            0x00000004

typedef __TLR_PACKED_PRE struct RCX_TIME_CMD_DATA_Ttag
{
  TLR_UINT32    ulTimeCmd;
  TLR_UINT32    ulData;
  TLR_UINT32    ulReserved;
} __TLR_PACKED_POST RCX_TIME_CMD_DATA_T;

/***** request packet *****/

typedef __TLR_PACKED_PRE struct RCX_TIME_CMD_REQ_Ttag
{
  TLR_PACKET_HEADER_T     tHead;     /* packet header     */
  RCX_TIME_CMD_DATA_T     tData;     /* packet data  */
} RCX_TIME_CMD_REQ_T;


/***** confirmation packet *****/

typedef __TLR_PACKED_PRE struct RCX_TIME_CMD_CNF_Ttag
{
  TLR_PACKET_HEADER_T     tHead;     /* packet header     */
  RCX_TIME_CMD_DATA_T     tData;     /* packet data  */
} RCX_TIME_CMD_CNF_T;
```


4.11 Custom hardware access interface / Serial DPM

The cifX/*netX Toolkit* supports an optional custom hardware interface to access the DPM of a netX based device. This interface allows to exchange the default read/write access functions from the *Toolkit* (e.g. memcpy() / pointer access) to the DPM by customer specific read/write functions. This feature must be explicitly enabled through a general toolkit definition (*#define CIFX_TOOLKIT_HWIF*) in the user project or compiler option.

Overview Custom Hardware Access Interface:

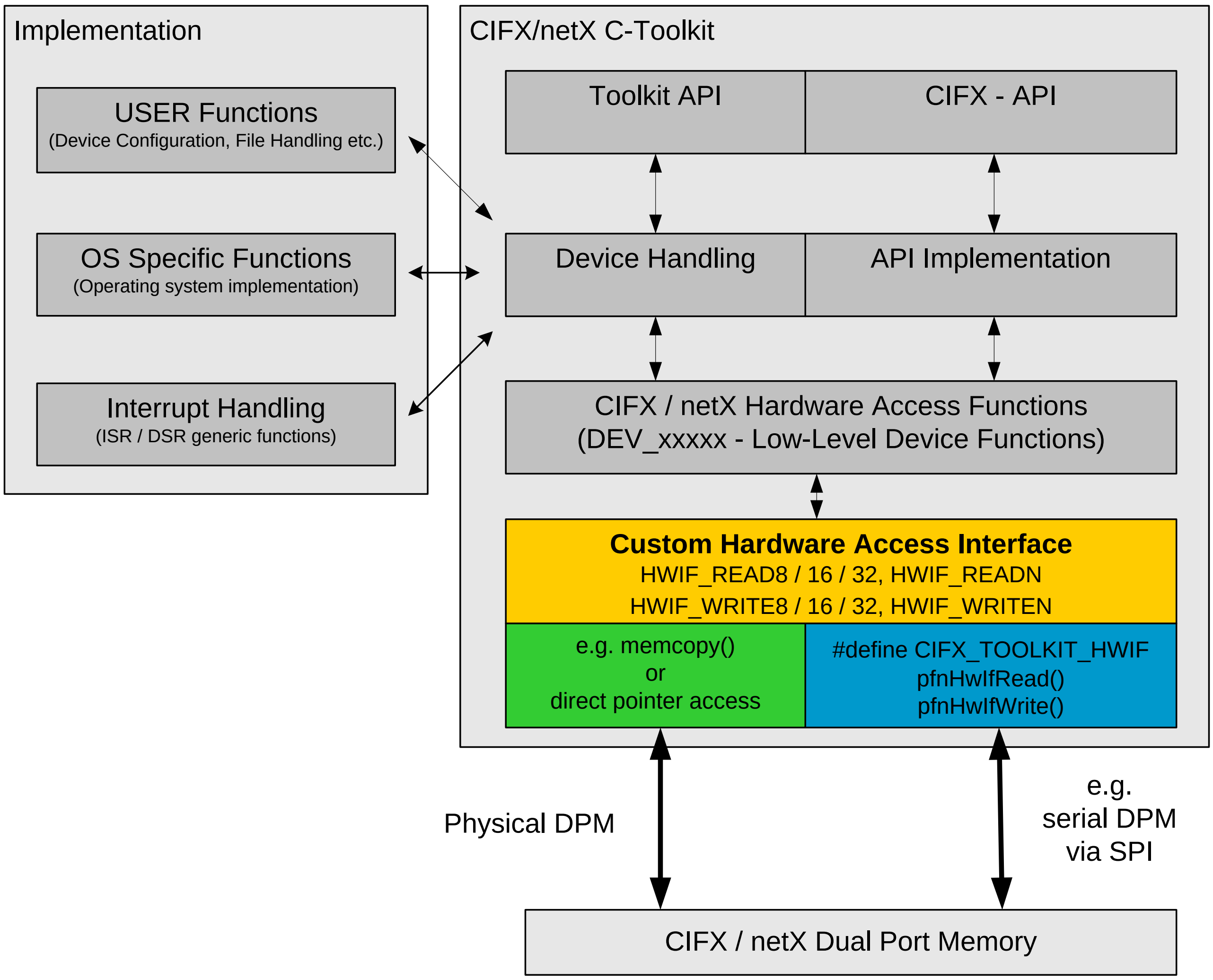

Figure 8: Overview custom hardware access interface

Calling Sequence of a Default DPM Access and a Custom Function Access:

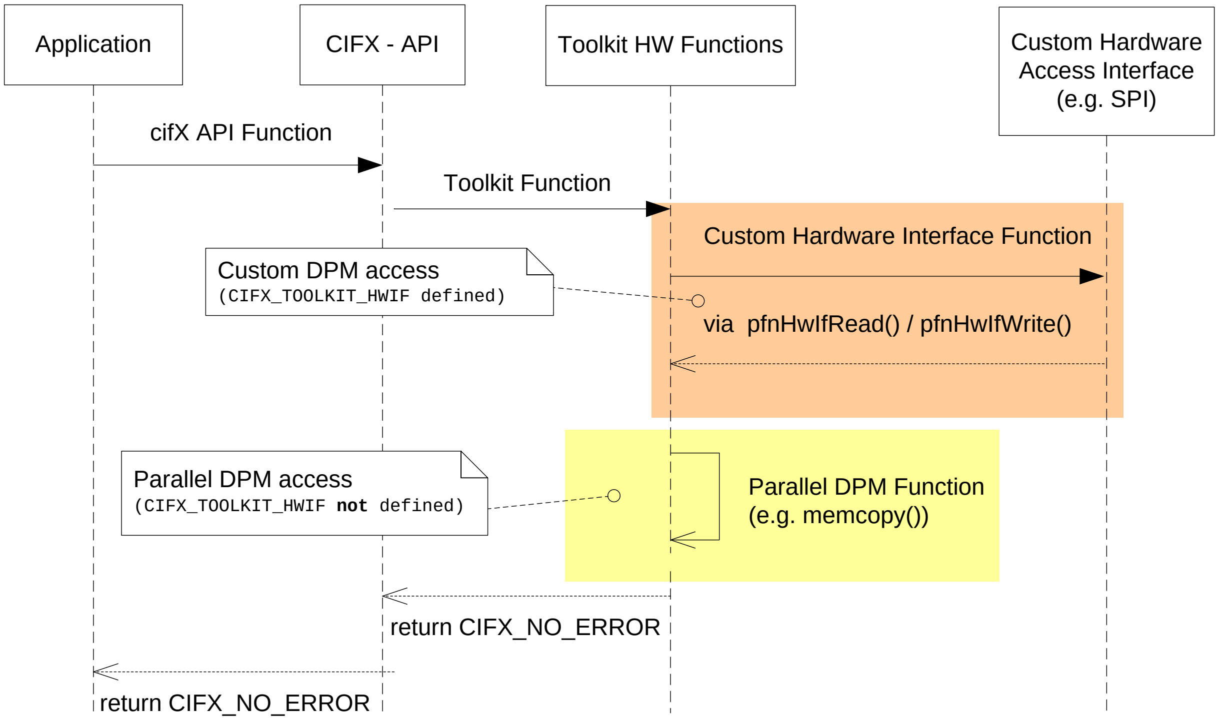

Figure 9: Calling sequence of a Default DPM Access and a Custom Function Access

The following diagram illustrates the functional principle on basis of the *xChannelGetMBXState()* call.

Calling Sequence Example: *xChannelGetMBXState()*

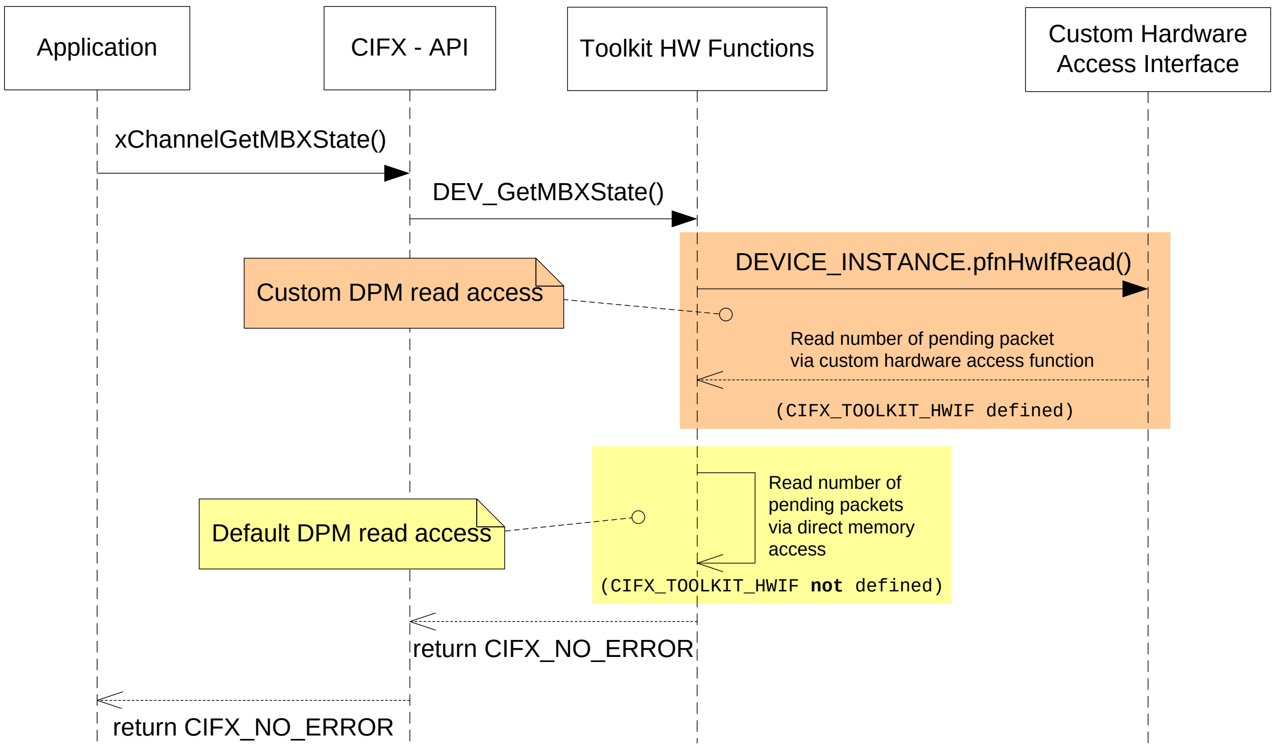

Figure 10: Calling Sequence Example: xChannelGetMBXState()

4.11.1 Defining and adding custom access functions

To use the custom hardware access interface, a read and a write access function must be implemented and announced, per device, to the *Toolkit* by assigning the *pfnHwIfRead* and *pfnHwIfWrite* function pointer of the DEVICE_INSTANCE structure with own read/write functions.

The *Toolkit* later uses the *pfnHwIfRead* and *pfnHwIfWrite* pointer whenever a DPM read or write should be processed.

Adding customer functions to the Toolkit:

- Setting the global toolkit definition to activate the custom hardware function handling

```
#define CIFX_TOOLKIT_HWIF
```

Activation of the custom hardware access interface expects the definition of the necessary hardware access functions in the DEVICE_INSTANCE structure before adding the device to the toolkit.

- Announcing / Passing the functions pointers of the custom read/write functions to the *Toolkit*

```
    /* Announce custom read/write access function */
    ptDevInstance->pfnHwIfRead     = <insert pointer to read access function>;
    ptDevInstance->pfnHwIfWrite    = <insert pointer to write access function>;

    /* Add the device to the toolkits handled device list */
    lRet = cifXTKitAddDevice(ptDevInstance);
```


4.11.1.1 Prototype of the Read Function (pfnHwIfRead())

Whenever the toolkit needs to read data from the DPM, the custom read access function is invoked.

Function Call

```
void* pfnHwIfRead (void* pvDevInstance, uint32_t ulAddr, void* pvData, uint32_t ulLen)
```

Arguments

Argument	Data type	Description
pvDevInstance	void*	Device instance of the device which should be accessed
ulAddr	uint32_t	Pointer to the source inside the DPM where the content is to be read from
pvData	void*	Pointer to the destination where the data read from DPM are copied to
ulLen	uint32_t	Number of bytes to read from DPM

Return Value

pvData is returned

4.11.1.2 Prototype of the Write Function (pfnHwIfWrite())

Whenever the toolkit needs to write data to the DPM, the custom write access function is invoked.

Function Call

```
void* pfnHwIfWrite (void* pvDevInstance, uint32_t ulAddr, void* pvData, uint32_t ulLen)
```

Arguments

Argument	Data type	Description
pvDevInstance	void*	Device instance of the device which should be accessed
ulAddr	uint32_t	Pointer/Offset to the destination inside the DPM where the content is to be written to
pvData	void*	Pointer to the source of data which should be copied to the DPM
ulLen	uint32_t	Number of bytes to write to DPM

Return Value

ulAddr is returned

4.11.2 Example

The following example code demonstrates the usage of the hardware access interface. Every read access to the DPM is processed via the *DPM_Read()* routine, every write access via the *DPM_Write()* routine, respectively.

```
/*****************************************************************************/
/*! Read a number of bytes from DPM interface
*   \param pvDevInstance Toolkit device instance (not used)
*   \param ulDpmAddr     Address in DPM to read data from
*   \param pvDst         Buffer to store read data
*   \param ulLen         Number of bytes to read                             */
/*****************************************************************************/
void* DPM_Read ( void* pvDevInstance, uint32_t ulAddr, void* pvData, uint32_t ulLen)
{
  uint8_t* pbSrc = (uint8_t*)ulAddr;
  uint8_t* pbDst = (uint8_t*)pvData;

  while (ulLen--)
    *pbDst++ = *pbSrc++;

  return pvData;
}

/*****************************************************************************/
/*! Write a number of bytes to DPM interface
*   \param pvDevInstance Toolkit device instance (not used)
*   \param ulDpmAddr     Address in DPM to store data to
*   \param pvDst         Buffer holding data to store
*   \param ulLen         Number of bytes to store                            */
/*****************************************************************************/
void* DPM_Write ( void* pvDevInstance, uint32_t ulAddr, void* pvData, uint32_t ulLen)
{
  uint8_t* pbSrc = (uint8_t*)pvData;
  uint8_t* pbDst = (uint8_t*)ulAddr;

  while (ulLen--)
    *pbDst++ = *pbSrc++;

  return (void*)ulAddr;
}
```

Before adding the cifX device to toolkit control, announce the DPM read/write access function by assigning the hardware access function pointer in the DEVICE_INSTANCE structure.

```
    /* Announce custom read/write access function */
    ptDevInstance->pfnHwIfRead     = DPM_Read;
    ptDevInstance->pfnHwIfWrite    = DPM_Write;

    /* Add the device to the toolkits handled device list */
    lRet = cifXTKitAddDevice(ptDevInstance);
```

4.11.3 Serial DPM Access via SPI

By introducing the new netX10 and netX51 controllers, SPI becomes a standard interface for accessing such netX based hardware. Please see section *How to Access Serial DPM via SPI* on page 17 of this manual to get further information about the serial DPM interface.

5 Toolkit initialization and usage

The following chapters are describing the toolkit specific functions which need to be called, to initialize all management functions and to add devices to the toolkit.

There is no hardware detection function included in the toolkit because such functions are very hardware specific and can't be implemented in a standard to meet all possible requirements (e. g. PCI bus scan, DPM address bus connection etc.).

Note: Hardware detection and enumeration (e.g. PCI) etc. is not part of the toolkit and need to be done by the user application or frame work.

The minimum information the toolkit needs to be able to access a device is a pointer to the DPM area of the netX based device (*ptDevInstance->pbDPM*) and the size of the DPM area (*ptDevInstance->ulDPMSize*).

Note: If a custom hardware interface is used, the access functions must be defined before adding the device to the toolkit.

This simple C-Source example shows the initialization process of the cifX/*netX Toolkit*.

```
  /* Initialize the cifX Toolkit */
  int32_t lRet = cifXTKitInit();
  if(CIFX_NO_ERROR == lRet)
  {
    PDEVICEINSTANCE ptDevInstance =
                  (PDEVICEINSTANCE)OS_Memalloc(sizeof(*ptDevInstance));
    OS_Memset(ptDevInstance, 0, sizeof(*ptDevInstance));
    ptDevInstance->fPCICard          = 0;
    ptDevInstance->pvOSDependent     = NULL;
    ptDevInstance->pbDPM             = <insert pointer to DPM>;
    ptDevInstance->ulDPMSize         = <insert size of DPM>;
    #ifdef CIFX_TOOLKIT_HWIF
      ptDevInstance->pfnHwIfRead     = <insert pointer to read access function>;
      ptDevInstance->pfnHwIfWrite    = <insert pointer to write access function>;
    #endif
    OS_Strncpy(ptDevInstance->szName,
               "cifX0",
               sizeof(ptDevInstance->szName));
    /* Add the device to the toolkits handled device list */
    lRet = cifXTKitAddDevice(ptDevInstance);
    /* If it succeeded do device tests */
    if(CIFX_NO_ERROR == lRet)
    {
       // Work with the device
    }
  }

===> Work with the cifX Driver API

  /* Uninitialize the cifX Toolkit if done */
  cifXTKitDeinit();
```


5.1 DEVICEINSTANCE structure

The DEVICEINSTANCE structure is the global management structure for each device. The buffer for this structure must be allocated and initialized by the user application. The pointer to the structure must be passed to the toolkit by calling the *cifXTKitAddDevice()* function.

5.1.1 User definable data in the DEVICEINSTANCE structure

Structure name: DEVICEINSTANCE, PDEVICEINSTANCE		
Element	**Type**	**Description**
Data to be inserted by user		
ulPhysicalAddress	uint32_t	Physical DPM address
bIrqNumber	uint8_t	Assigned interrupt number
fIrqEnabled	int	0 = Not using interrupts 1 = Interrupt should be used **Note:** This will indirectly be set via a *USER_GetInterruptEnable()* call
fPCICard	int	0 = None PCI/PCIe card **Note:** None PCI cards will be checked for a running firmware before attempting a reset 1 = PCI/PCIe card **Note:** PCI/PCIe cards are usually reset every time they are added to the toolkit. Except *eDeviceType* is set to eCIFX_DEVICE_TYP_DONT_TOUCH.
eDeviceType	CIFX_TOOLKIT_DEVICETYPE_E	Type of the device (RAM / Flash based) 0 = eCIFX_DEVICE_AUTODETECT (**default**) Autodetection → (PCI=RAM, DPM=Flash based) 1 = eCIFX_DEVICE_AUTODETECT_ERROR Internally used if autodetection fails 2 = eCIFX_DEVICE_RAM_BASED RAM based devices are reset during startup 3 = eCIFX_DEVICE_FLASH_BASED FLASH based device with running Firmware expected 4 = eCIFX_DEVICE_DONT_TOUCH Leave the device in the current state and try to connect to it
Note: eDeviceType is used to distinguish between the different firmware behaviors in conjunction with the hardware. In general 2 types of hardware are defined and supported: ▪ RAM-based hardware (firmware and configuration **not stored** on the hardware and must always be loaded) ▪ Flash-based hardware (firmware and configuration **stored** on the hardware in Flash) eDeviceType can be used to change the default device handling inside the toolkit (see also section *FLASH-based vs RAM-based devices* on page 27).		

Structure name: DEVICEINSTANCE, PDEVICEINSTANCE		
Element	**Type**	**Description**
Data to be inserted by user		
pvOSDependent	void*	Pointer to user dependent data, not used by the Toolkit. **Note:** This pointer can be used to pass user dependent data to the *USER_xxx* and *OS_xxx* functions. If the Toolkit is used inside a device driver, this pointer is used to pass operating system dependent data to the Toolkit functions (e.g. used for PCI cards to store PCI register information)
pbDPM	uint8_t*	Pointer to the dual ported memory
ulDPMSize	uint32_t	Total/mapped dual ported memory size
szName	char[16]	Device name (e.g. "cifX0")
szAlias	char[16]	Alias name for the card. Asynchronously fetched from user by a call to *USER_GetAliasName()*, during device initialization
pfnNotify	PFN_CIFXTK_NOTIFY	Notification callback function during hardware initialization This callback function can be used if additional handling between the different initialization stages of the hardware is necessary. (e.g. adjust DPM settings (8Bit / 16Bit) if they are different between ROM- and Bootloader startup) Available notifications: defined in CIFX_TOOLKIT_NOTIFY_E 0 = eCIFX_TOOLKIT_EVENT_PRERESET 1 = eCIFX_TOOLKIT_EVENT_POSTRESET 2 = eCIFX_TOOLKIT_EVENT_PRE_BOOTLOADER 3 = eCIFX_TOOLKIT_EVENT_POST_BOOTLOADER
DMA Mode only		
ulDMABufferCount	uint32_t	Number of mapped DMA buffers
atDmaBuffers	CIFX_DMABUFFER_T[8]	8 DMA Buffers that can be used by the toolkit. **Note:** These buffers must be a multiple of 256 in size, and must by physically contiguous
Custom Hardware Access Interface only **Note:** Usable only if the global Toolkit option "CIFX_TOOLKIT_HWIF" is defined		
pfnHwIfRead	PFN_HWIF_MEMCPY	Function pointer if user defined functions should be used to read data from the DPM
pfnHwIfWrite	PFN_HWIF_MEMCPY	Function pointer if user defined functions should be used to write data to the DPM
Extended Memory Information (additional target memory) **Note:** This information is used by xSysdeviceExtendedMemory()		
pbExtendedMemory	uint8_t*	Pointer to an extended memory area
ulExtendedMemorySize	uint32_t	Size of the extended memory area

Table 12: Device instance structure - User provided data

5.1.2 Toolkit internal data in the DEVICEINSTANCE structure

Structure name: DEVICEINSTANCE, PDEVICEINSTANCE		
Element	**Type**	**Description**
Toolkit internal data		
lInitError	int32_t	Device initialization error, if any
ptGlobalRegisters	PNETX_GLOBAL_REGBLOCK	Pointer to the netX global register block at the end of the DPM
ulSerialNumber	uint32_t	Serial number (read during startup)
ulDeviceNumber	uint32_t	Device number (read during startup)
tSystemDevice	CHANNELINSTANCE	System device instance (this must exist once)
ulCommChannelCount	uint32_t	Number of found communication channels
pptCommChannels	PCHANNELINSTANCE*	Array of channel instances
iIrqToDsrBuffer	int	IRQ/DSR synchronization buffer number
atIrqToDsrBuffer	NETX_HANDSHAKE_ARRAY[]	Two synchronization buffers for ISR/DSR
ulIrqCounter	uint32_t	IRQ counters (informational use)
pbHandshakeBlock	uint8_t*	Pointer to the handshake block
eChipType	CIFX_TOOLKIT_CHIPTYPE_E	Type of the chip. This is detected during *cifXTKitAddDevice()* call.
ulSlotNumber	uint32_t	Slot number for cifX cards with rotary switch. This variable can be accessed in *USER_GetFirmwareFile()* / *USER_GetConfigurationFile()* functions for selecting a proper firmware. **Note:** Cards without rotary switch will return 0 as slot number
fResetActive	int	Indicated an active system reset. This flag is used to synchronize handshake flag access between DSR and DEV_DoSystemStart

Table 13: Device instance structure - Internal data

5.2 CHANNELINSTANCE structure

The CHANNELINSTANCE structure is used to manage the system channel and communication channels per device. A system channel instance is always available. Communication channel structures are allocated during the device startup phase in the toolkit.

Structure name: CHANNELINSTANCE, P CHANNELINSTANCE		
Element	**Type**	**Description**
pvDeviceInstance	void*	Pointer to the device instance belonging to this channel
pvInitMutex	void*	Device is currently initializing, e.g. while doing a reset
pbDPMChannelStart	uint8_t	Virtual start address of channel block
ulDPMChannelLength	uint32_t	Length of channel block in bytes
ulChannelNumber	uint32_t	Number of the communication channel (0...n)
ulBlockID	uint32_t	Dual port memory block number (0...7)
pvLock	void*	Lock for synchronizing interrupt accesses to flags
ulOpenCount	uint32_t	Reference counter for calls to *xChannelOpen() / xChannelClose()*
fIsSysDevice	int	!=0 if the channel instance belong to a system device
fIsChannel	int	!=0 if the channel belongs to a communication channel
tFirmwareIdent	NETX_FW_IDENTIFICATION	Firmware Identification
tSendMbx	NETX_TX_MAILBOX_T	Send mailbox administration structure
tRecvMbx	NETX_TX_MAILBOX_T	Receive mailbox administration structure
usHostFlags	uint16_t	Copy of the last actual command flags
usNetxFlags	uint16_t	Copy of the last read status flags
ulDeviceCOSFlags	uint32_t	Device COS flags (copy, updated when COS Handshake is recognized)
ulDeviceCOSFlagsChanged	uint32_t	Bit mask of changed bits since last COS Handshake
ulHostCOSFlags	uint32_t	Host COS flags (copy)
ptControlBlock	NETX_CONTROL_BLOCK*	Pointer to channel control block
bControlBlockBit	uint8_t	Handshake bit associated with control block
ulControlBlockSize	uint32_t	Size of the control block in bytes
ptCommonStatusBlock	NETX_COMMON_STATUS_BLOCK*	Pointer to channel common status block
bCommonStatusBit	uint8_t	Handshake bit associated with Common status block
ulCommonStatusSize	uint32_t	Size of the common status block in bytes
ptExtendedStatusBlock	NETX_EXTENDED_STATUS_BLOCK*	Pointer to channel extended status block
bExtendedStatusBit	uint8_t	Handshake bit associated with Extended status block
ulExtendedStatusSize	uint32_t	Size of the extended status block in bytes

Structure name: CHANNELINSTANCE, P CHANNELINSTANCE		
Element	**Type**	**Description**
bHandshakeWidth	unit8_t	Width of the handshake cell
ptHandshakeCell	NETX_HANDSHAKE_CELL*	Pointer to channel handshake cell
ahHandshakeBitEvents	void*	Event handle for each handshake bit pair. (used in interrupt mode)
pptIOInputAreas	PIOINSTANCE*	Array of input areas on this channel
ulIOInputAreas	uint32_t	Number of input areas
pptIOOutputAreas	PIOINSTANCE*	Array of output areas on this channel
ulIOOutputAreas	uint32_t	Number of Output areas
pptUserAreas	PUSERINSTANCE*	Array of user areas on this channel
ulUserAreas	uint32_t	Number of user areas
tSynch	NETX_SYNC_DATA_T	Sync handling data

Table 14: CHANNELINSTANCE structure

6 Toolkit functions

The toolkit functions are divided into three different parts:

- General toolkit functions
  General Functions are used to implement the toolkit into an own environment.
- OS abstraction for operating system independent implementation
  Internal handling of the DPM expects some functionalities which are potentially operating system or compiler depending. These functions are placed into an OS specific module to keep the toolkit independent from such dependencies.
- USER functions
  User environment specific functions to adapt the user environment to the toolkit (e.g. trace functions, file access functions, configuration information etc.).

6.1 General Toolkit functions

These functions are used by a user application or frame work to integrate the toolkit and its functions.

General Toolkit functions	Description
cifXTKitInit	Initialize the Toolkit
cifXTKitDeinit	Un-initialize the Toolkit
cifXTKitAddDevice	Add a device (card) to be handled the Toolkit
cifXTKitRemoveDevice	Remove a device from being handled by the Toolkit
cifXTKitCyclicTimer	Cyclic Toolkit function for poll devices
cifXTKitISRHandler	Interrupt service handler
cifXTKitDSRHandler	Deferred service routine for interrupt handling

Table 15: General Toolkit Functions

6.1.1 cifXTKitInit

This function initializes the whole toolkit. It can also be called to re-initialize the toolkit allowing starting over. This function must be called before using any of the toolkit functions.

Function Call

```
int32_t cifXTKitInit(void)
```

Arguments

Argument	Data type	Description
none		

Return Values

Return Values	
CIFX_NO_ERROR	Toolkit initialization successful

6.1.2 cifXTKitDeinit

Un-initializes the toolkit. This call will remove all handled devices and frees all allocated memory. Any access to the toolkit functions may result in an access violation if any access is made after the toolkit is un-initialized.

Function Call

```
void cifXTKitDeinit(void)
```

Arguments

Argument	Data type	Description
none		

Return Values

Return Values	
none	

6.1.3 cifXTKitAddDevice

This function adds a device to be handled by the toolkit. A user application has to pass the access name (e.g. "cifX0") and the pointer to the dual ported memory.

Informational data like physical address, interrupt number etc. can also be passed, but will only be used on calls to information functions. The passed device instance must be correctly initialized for the toolkit to behave properly.

Note: Because of the different handling of so called DPM based devices (comX) and PCI based device (cifX). It is important to correctly set the *fPCICard* flag in the *ptDevInst* structure.

Note: This function might return successfully even if underlying components has failed to initialize. The initialization will not be aborted due to a hardware failure. The status can be evaluated later using CIFX-API functions.
`lInitError`, provided in `PDEVICEINSTANCE` structure can be used to evaluate possible “internal” errors.

```
int32_t cifXTKitAddDevice(PDEVICEINSTANCE ptDevInst)
```

Arguments

Argument	Data type	Description
ptDevInst	PDEVICEINSTANCE	Pointer to the user allocated device instance structure which is being handled by the toolkit.

Return Values

Return Values	
CIFX_NO_ERROR	Successfully added device
CIFX_INVALID_POINTER	Invalid device instance pointer passed (NULL)
CIFX_MEMORY_MAPPING_FAILED	Dual ported memory was not accessible. (e.g. wrong DPM Pointer passed or the OS_PCIRead/WriteRegisters does not correctly work on the PC card, leaving the card in an unsafe mode after a reset)
CIFX_DRV_INIT_STATE_ERROR	Card could not correctly be reset. This could rely on an invalid DPM pointer describing accessible memory which does not belong to the card. The card has a bootable firmware in its FLASH and does not answer to PCI download routines.
CIFX_FILE_OPEN_FAILED	The bootloader/firmware/configuration file could not be opened. Check your *USER_GetXXX()* function.
Check `lInitError`, in `PDEVICEINSTANCE` for possible internal errors if necessary (see note above).	

Example:

```
/* Initialize the Toolkit first */
int32_t lRet = cifXTKitInit();



if(CIFX_NO_ERROR == lRet)
{
  PDEVICEINSTANCE ptDevInstance = (PDEVICEINSTANCE)OS_Memalloc( sizeof(*ptDevInstance));
  OS_Memset(ptDevInstance, 0, sizeof(*ptDevInstance));

  ptDevInstance->fPCICard          = 0;
  ptDevInstance->pvOSDependent     = NULL;
  ptDevInstance->pbDPM             = <insert DPM pointer>;
  ptDevInstance->ulDPMSize         = <insert DPM size>;
  OS_Strncpy(ptDevInstance->szName,
             "cifX0",
             sizeof(ptDevInstance->szName));

  /* Add the device to the toolkits handled device list */
  lRet = cifXTKitAddDevice(ptDevInstance);

  if(CIFX_NO_ERROR == lRet)
  {
    /* From this point the CIFX API can be used to access the device */
  }else
  {
    /* Failed to add a device to the toolkit, free the previously allocated device */
    free(ptDevInstance);
  }
}



/* Uninitialize Toolkit at the end of the program */
/* this will removes all handled boards from the toolkit */
cifXTKitDeinit();
```

6.1.4 cifXTKitRemoveDevice

This function removes a device from the toolkit. The device is selected by passing the access name (e.g. "cifX0"). The device instance, passed to the toolkit during initialization, will be freed automatically by a call to *OS_Memfree()*.

Function Call

```
int32_t cifXTKitRemoveDevice(   char* szBoard,
                                int   fForceRemove)
```

Arguments

Argument	Data type	Description
szBoard	char*	ASCII string describing the device. This can be the initially passed name.
fForceRemove	int	This parameter can be used to force the removing of the device from the toolkit, even if any references are still open. ATTENTION: This can raise an access violation if an application is still accessing the device!!!

Return Values

Return Values	
CIFX_NO_ERROR	Successfully removed device
CIFX_INVALID_BOARD	Board with the given name was not found
CIFX_DEV_HW_PORT_IS_USED	There is still an open reference to the board. This error is only returned if fForceRemove == 0

6.1.5 cifXTKitCyclicTimer

This function must be called by the user to cyclically check device (non-irq mode) for change of state (COS) commands from the hardware. This function processes all devices and channels to check any pending COS handshake bit changes (only on polled devices), even when no application is running.

Note: The recommended cycle is about 500ms or less.

Function Call

```
void cifXTKitRemoveDevice( void)
```

Arguments

Argument	Data type	Description
none		

Return Values

Return Values	
none	

6.1.6 cifXTKitISRHandler

Interrupt service routine for cifX devices. This function must be called by the user if an interrupt for a given device is signaled. On PCI busses the function is able to detect if the interrupt was issued by the selected device.

The ISR handler function will read the hardware interrupt flags and stores the flags in the give device instance for later processing in the *cifXTKitDSRHandler()*.

Reading the interrupt flags also acknowledges and deletes the physical hardware interrupt. Splitting the interrupt processing into an ISR and DSR function is done for operating systems which do not allow to calling inter-process communication functions at the physical interrupt level.

Note: The user is responsible to pass the correct device instance for the occurred interrupt.

Function Call

```
int cifXTKitISRHandler(    PDEVICEINSTANCE ptDevInstance
                           int             fPCIIgnoreGlobalIntFlag)
```

Arguments

Argument	Data type	Description
ptDevInstance	PDEVICEINSTANCE	Device instance the interrupt occurred for
fPCIIgnoreGlobalIntFlag	int	Ignore the global interrupt flag on PCI cards, to detect shared interrupts. This might be necessary if the user has already filtered out all shared IRQs 0 = Handle global interrupt flag 1 = Ignore global interrupt flag

Return Values

Return Values	
CIFX_TKIT_IRQ_OTHERDEVICE	The interrupt was issued by another device on the shared PCI bus
CIFX_TKIT_IRQ_HANDLED	The interrupt was handled, and does not need any further processing
CIFX_TKIT_IRQ_DSR_REQUESTED	The interrupts was acknowledged, but needs further handling in a deferred service routine. The user is expected to call a DSR in an interruptible context on this return value.

6.1.7 cifXTKitDSRHandler

Deferred service handler routine for cifX devices. This function must be called by the ISR handler returned CIFX_TKIT_IRQ_DSR_REQUESTED. The DSR is expected to be interruptible and will process the interrupt events in non-interrupt mode.

The user is responsible to pass the correct device instance for the occurred interrupt.

Function Call

```
void cifXTKitDSRHandler(   PDEVICEINSTANCE ptDevInstance)
```

Arguments

Argument	Data type	Description
ptDevInstance	PDEVICEINSTANCE	Device instance the interrupt occurred for

Return Values

Return Values	
none	

6.2 OS Abstraction

The OS Abstraction Layer is introduced to allow the toolkit to run under several operating systems, without needing to change the toolkit components. The OS Abstraction needs to be implemented by the user and is only included for Win32 user mode applications.

OS Abstraction	
Memory Functions	
OS_Memalloc	Allocate memory
OS_Memfree	Free allocated memory
OS_Memrealloc	Change size of an allocated memory block
OS_Memset	Set a memory area
OS_Memcpy	Copy a memory area
OS_Memcmp	Compare a memory area
OS_Memmove	Move a memory area
PCI Functions	
OS_ReadPCIConfig	Read PCI configuration information
OS_WritePCIConfig	Write PCI configuration information
Interrupt Functions	
OS_EnableInterrupts	Enable device interrupt
OS_DisableInterrupts	Disable device interrupt
File Function	
OS_FileOpen	Open a file
OS_FileRead	Read a file
OS_FileClose	Close a file
Timing Function	
OS_GetMilliSecCounter	Get a millisecond counter value
OS_Sleep	Suspend a process for a given time
Synchronisation Function (Critical Section)	
OS_CreateLock	Create a lock object
OS_EnterLock	Enter a locked program region
OS_LeaveLock	Leave a locked program region
OS_DeleteLock	Delete a lock object
Synchronisation Function (Mutual Exclusion)	
OS_CreateMutex	Create a Mutex (Mutual Exclusion) object
OS_WaitMutex	Wait for a Mutex
OS_ReleaseMutex	Release a Mutex
OS_DeleteMutex	Delete a Mutex object
Synchronisation Function (Event)	
OS_CreateEvent	Create an event object
OS_SetEvent	Set an event object into a signaled state
OS_ResetEvent	Reset an event object to a none signaled state

OS Abstraction	
OS_DeleteEvent	Delete an event object
OS_WaitEvent	Wait for an event to be signaled
String Functions(Mutal Exclusion)	
OS_Strcmp	Copy a string
OS_Strlen	Get the length of a string
OS_Strncpy	Compare two strings
OS_Strnicmp	Compare two strings (case-insensitive)
Memory Mapping Functions	
OS_MapUserPointer	Map a memory region to be accessible by a user application
OS_UnmapUserPointer	Unmap a previously mapped memory region

Table 16: OS Abstraction Functions

6.2.1 Initialization

Some operating systems must run a special initialization before any functions can be called. Therefore the toolkit calls the following two functions during initialization / un-initialization.

6.2.1.1 OS_Init

Initialization of the operating system abstraction layer (OS layer).

Function Call

```
int32_t OS_Init(void)
```

Arguments

Argument	Data type	Description
none		

Return Values

Return Values	
CIFX_NO_ERROR	successfully initialized OS Layer

6.2.1.2 OS_Deinit

Un-initialization of the operating system abstraction layer (OS layer).

Function Call

```
void OS_Deinit(void)
```

Arguments

Argument	Data type	Description
none		

Return Values

Return Values	
none	

6.2.2 Memory operations

Memory allocation and operation differ between operating systems and even inside the operating system, depending on the mode the application/driver is running. The memory routines are included in the OS Abstraction to allow easy adaptation and modification.

6.2.2.1 OS_Memalloc

Memory allocation routine.

Function Call

```
void* OS_Memalloc(uint32_t ulSize)
```

Arguments

Argument	Data type	Description
ulSize	uint32_t	Size in bytes to allocate

Return Values

A pointer to the allocated memory is returned. NULL indicates memory allocation failure.

6.2.2.2 OS_Memfree

Memory freeing function.

Function Call

```
void OS_Memfree(void* pvMem)
```

Arguments

Argument	Data type	Description
pvMem	void*	Memory block to free

6.2.2.3 OS_Memrealloc

Memory resize / reallocation Function

Function Call

```
void* OS_Memrealloc(void* pvMem, uint32_t ulNewSize)
```

Arguments

Argument	Data type	Description
pvMem	void*	Memory block to resize
ulNewSize	uint32_t	New size of block in bytes

Return Values

A pointer to the reallocated memory is returned. NULL indicates memory reallocation failure.

6.2.2.4 OS_Memcpy

Copy function for non-overlapping memory areas which copies one block to another.

Function Call

```
void OS_Memcpy( void*           pvDest,
void*           pvSrc,
uint32_t        ulSize)
```

Arguments

Argument	Data type	Description
pvDest	void*	Destination memory
pvSrc	void*	Source memory
ulSize	uint32_t	Size in bytes being copied

6.2.2.5 OS_Memmove

Move overlapping memory areas from one block to another.

Function Call

```
void OS_Memmove(     void*           pvDest,
                     void*           pvSrc,
                     uint32_t        ulSize)
```

Arguments

Argument	Data type	Description
pvDest	void*	Destination memory
pvSrc	void*	Source memory
ulSize	uint32_t	Size in bytes being moved

6.2.2.6 OS_Memset

Initialize a memory block to a predefined value.

Function Call

```
void OS_Memset( void*           pvMem,
                uint8_t         bFill,
                uint32_t        ulSize)
```

Arguments

Argument	Data type	Description
pvMem	void*	Memory block to initialize
bFill	uint8_t	Fill byte
ulSize	uint32_t	Size in bytes being initialized

6.2.2.7 OS_Memcmp

Compare the content of two memory blocks.

Function Call

```
int OS_Memcmp(  void*           pvBuf1,
                void*           pvBuf2,
                uint32_t        ulSize)
```

Arguments

Argument	Data type	Description
pvBuf1	void*	First compare buffer
pvBuf2	void*	Second compare buffer
ulSize	uint32_t	Number of bytes to compare

Return Values

Return Values	
0	Memory contents equal
<0	pvBuf1 < pvBuf2
>0	pvBuf1 > pvBuf2

6.2.3 String operations

String operations are used inside the toolkit for the board/alias name handling and also for accessing ASCII strings inside the firmware information. The implementation should rely on ASCII / MBCS characters.

6.2.3.1 OS_Strncpy

Copy one string into another, considering the length of the destination buffer.

Function Call

```
char* OS_Strncpy(    char*           szDest,
                     const char*     szSource,
                     uint32_t        ulLen)
```

Arguments

Argument	Data type	Description
szDest	char*	Destination string buffer
szSource	const char*	Source string buffer
ulLen	uint32_t	Maximum length to copy

Return Values

Pointer to *szDest.*

6.2.3.2 OS_Strlen

Count the number of characters inside a string.

Function Call

```
int OS_Strlen(  const char* szText)
```

Arguments

Argument	Data type	Description
szText	const char*	String to determine length from

Return Values

Length of string in characters.

6.2.3.3 OS_Strcmp

Compare the content of two strings.

Function Call

```
int OS_Strcmp(  const char*     pszBuf1,
                const char*     pszBuf2)
```

Arguments

Argument	Data type	Description
pszBuf1	const char*	First compare string
pszBuf2	const char*	Second compare string

Return Values

Return Values	
0	String are equal
<0	pszBuf1 less than pszBuf2
>0	pszBuf1 greater than pszBuf2

6.2.4 Event handling

Events are used to indicate changes in interrupt mode from the interrupt routine to the user functions.

6.2.4.1 OS_CreateEvent

Create a new, unnamed, automatic reset event.

Function Call

```
void* OS_CreateEvent(void)
```

Arguments

Argument	Data type	Description
none		

Return Values

Return Values	
NULL	Event creation error
otherwise	Handle to an event object

6.2.4.2 OS_DeleteEvent

Delete a previously created event.

Function Call

```
void OS_DeleteEvent(void* pvEvent)
```

Arguments

Argument	Data type	Description
pvEvent	void*	Event handle to delete

Return Values

Return Values	
none	

6.2.4.3 OS_SetEvent

Signal an event.

Function Call

```
void OS_SetEvent(void* pvEvent)
```

Arguments

Argument	Data type	Description
pvEvent	void*	Event handle to signal

Return Values

Return Values	
none	

6.2.4.4 OS_ClearEvent

Reset a signaled event.

Function Call

```
void OS_ResetEvent(void* pvEvent)
```

Arguments

Argument	Data type	Description
pvEvent	void*	Event handle to reset

Return Values

Return Values	
none	

6.2.4.5 OS_WaitEvent

Wait for the occurrence of a given event

Function Call

```
uint32_t OS_WaitEvent(    void*      pvEvent,
                          uint32_t   ulTimeout)
```

Arguments

Argument	Data type	Description
pvEvent	void*	Event handle to wait for being signaled
ulTimeout	uint32_t	Time in ms to wait for event

Return Values

Return Values	
CIFX_EVENT_SIGNALLED (0)	Event was signaled during wait
CIFX_EVENT_TIMEOUT (1)	Timeout waiting for event

6.2.5 File handling

Depending on the used platform, the device may have a file system or not. Depending where the firmware and configuration files are stored, the file routines may access other devices like FLASH etc.

6.2.5.1 OS_FileOpen

Open a file for reading in binary mode.

Function Call

```
void* OS_FileOpen(   char*          szFilename,
                     uint32_t*      pulFileSize)
```

Arguments

Argument	Data type	Description
szFilename	char*	Name of the file to open
pulFileSize	uint32_t*	Returned file size in bytes of opened file

Return Values

Return Values	
NULL	File could not be opened
otherwise	Handle to the open file

6.2.5.2 OS_FileClose

Close a previously opened file.

Function Call

```
void OS_FileClose(   void* pvFile)
```

Arguments

Argument	Data type	Description
pvFile	void*	Handle to the file being closed

Return Values

Return Values	
none	

6.2.5.3 OS_FileRead

Read binary data from an open file.

Function Call

```
uint32_t OS_FileRead(      void*           pvFile,
                           uint32_t        ulOffset,
                           uint32_t        ulSize,
                           void*           pvBuffer)
```

Arguments

Argument	Data type	Description
pvFile	void*	Handle to the file being read from
ulOffset	uint32_t	Offset inside file the read should start at
ulSize	uint32_t	Number of bytes to be read
pvBuffer	void*	Buffer to place read data in

Return Values

The function returns the actual number of bytes that were read from the file.

6.2.6 Synchronization / Locking / Timing

6.2.6.1 OS_CreateLock

Creates a new synchronization object (e.g. Critical Section).

Function Call

```
void* OS_CreateLock(void)
```

Arguments

Argument	Data type	Description
none		

Return Values

Return Values	
NULL	Object creation error
otherwise	Handle to a synchronization object

6.2.6.2 OS_DeleteLock

Delete a previously created synchronization object (e.g. Critical Section).

Function Call

```
void OS_DeleteLock(void* pvLock)
```

Arguments

Argument	Data type	Description
pvLock	void*	Synchronization object to delete

Return Values

None

6.2.6.3 OS_EnterLock

Lock the synchronization object for the current context. This call blocks until the lock has been acquired.

Function Call

```
void OS_EnterLock(void* pvLock)
```

Arguments

Argument	Data type	Description
pvLock	void*	Synchronization object to enter

Return Values

none

6.2.6.4 OS_LeaveLock

Unlock the synchronization object for the current context.

Function Call

```
void OS_LeaveLock(void* pvLock)
```

Arguments

Argument	Data type	Description
pvLock	void*	Synchronization object to leave

Return Values

None

6.2.6.5 OS_CreateMutex

Create a Mutex (Mutal Exclusion Object). Mutexes are used to prevent some functions to be accessed re-entrant.

Function Call

```
void* OS_CreateMutex (void)
```

Arguments

Argument	Data type	Description
none		

Return Values

Handle to the Mutex (NULL on error).

6.2.6.6 OS_DeleteMutex

Delete a Mutex.

Function Call

```
void OS_CreateMutex (void* pvMutex)
```

Arguments

Argument	Data type	Description
pvMutex	void*	Pointer to the Mutex to delete

Return Values

None

6.2.6.7 OS_WaitMutex

Wait to acquire a Mutex.

Function Call

```
int OS_WaitMutex (void* pvMutex, uint32_t ulTimeout)
```

Arguments

Argument	Data type	Description
pvMutex	void*	Handle of the Mutex to wait for
ulTimeout	uint32_t	Timeout in ms to wait for Mutex

Return Values

None zero if Mutex is acquired successfully.

6.2.6.8 OS_ReleaseMutex

Release a previously acquired Mutex.

Function Call

```
void OS_ReleaseMutex (void* pvMutex)
```

Arguments

Argument	Data type	Description
pvMutex	void*	Handle of the Mutex to release

Return Values

None

6.2.6.9 OS_Sleep

Delay execution of a program by the given time in milliseconds. This call is allowed to do a task switch, but can also be implemented as stall execution.

Function Call

```
void OS_Sleep(uint32_t ulSleepTimeMs)
```

Arguments

Argument	Data type	Description
ulSleepTimeMs	uint32_t	Time in ms to sleep

Return Values

None

6.2.6.10 OS_GetMilliSecCounter

Retrieve the free running millisecond counter of the operating system. The resolution influences the timeout monitoring accuracy.

Function Call

```
uint32_t OS_GetMilliSecCounter(void)
```

Arguments

Argument	Data type	Description
none		

Return Values

Actual value of the systems millisecond counter

6.2.7 PCI routines

These functions are needed, if PCI cards should be handled. The PCI cifX cards are being reset during startup and need to have their PCI configuration registers restored after a reset.

A hardware reset will also reset the PCI core of the netX and all previously inserted PCI configuration information is lost. Therefore the toolkit offers two functions which are called before and after the execution of a hardware reset.

The following table shows the values which needs to be recovered:

Value	Data type	Description
BAR0	uint32_t	PCI Base Address Register 0
BAR1	uint32_t	PCI Base Address Register 1
BAR2	uint32_t	PCI Base Address Register 2
Interrupt Line	uint32_t	PCI Interrupt Line Register
Command/State	uint32_t	PCI Command/Status Register

The PCI specification defines the PCI registers settings in a defined structure (PCI_COMMON_CONFIG structure) and the whole structure should be stored / restored to make sure to restore the information 1:1. The size of the structure is 256 Byte.

Note: Store / restore the complete PCI hardware configuration registers (PCI_COMMON_CONFIG structure).

Note: Make sure to restore the Command/State register as the last one and all other registers are already valid.

6.2.7.1 OS_ReadPCIConfig

Read the actual PCI configuration registers and store them.

Function Call

```
void* OS_ReadPCIConfig(void* pvOSDependent)
```

Arguments

Argument	Data type	Description
pvOSDependent	void*	OS dependent object that has been passed in the device instance during *cifXTKitAddDevice()*

Return Values

Pointer to the stored PCI registers data. Depending on the content of *pvOSDependent* the register content can also be stored in this object.

Returns NULL in case the PCI registers could not be accessed/saved.

6.2.7.2 OS_WritePCIConfig

Write a previously stored PCI configuration to the device.

Function Call

```
void OS_WritePCIConfig(    void*      pvOSDependent,
                           void*      pvPCIConfig)
```

Arguments

Argument	Data type	Description
pvOSDependent	void*	OS dependent object that has been passed in the device instance during cifXTKitAddDevice
pvPCIConfig	void*	Pointer returned from OS_ReadPCIConfig

Return Values

None

6.2.8 Interrupt routines

These functions are needed, to allow the toolkit to enable/disable device interrupts. This function should register and enable the devices interrupt on the operating system (e.g. connecting a interrupt on Windows) and not for the complete CPU.

6.2.8.1 OS_EnableInterrupts

Enable the physical interrupt for the given device.

Function Call

```
void OS_EnableInterrupts( void* pvOSDependent)
```

Arguments

Argument	Data type	Description
pvOSDependent	void*	OS dependent object that has been passed in the device instance during *cifXTKitAddDevice()*

Return Values

None

6.2.8.2 OS_DisableInterrupts

Disable the interrupt on the given device.

Function Call

```
void OS_DisableInterrupts( void* pvOSDependent)
```

Arguments

Argument	Data type	Description
pvOSDependent	void*	OS dependent object that has been passed in the device instance during *cifXTKitAddDevice()*

Return Values

None

6.2.9 Memory mapping functions

The memory mapping functions are needed, if pointers are passed from the toolkit to an application. If the driver is running in kernel mode, it may be needed to map the pointer to the caller. This is used inside the functions which return pointers to the DPM areas.

6.2.9.1 OS_MapUserPointer

Map a pointer to be usable in the applications context.

Function Call

```
void* OS_MapUserPointer(  void*           pvDriverMem,
                          uint32_t        ulMemSize,
                          void**          ppvMappedMem,
                          void*           pvOSDependet)
```

Arguments

Argument	Data type	Description
pvDriverMem	void*	Pointer that is valid inside driver context
ulMemSize	uint32_t	Size of the memory to map
ppvMappedMem	void**	Returned mapped pointer
pvOsDependent	void*	OS dependent object that has been passed in the device instance during *cifXTKitAddDevice()*

Return Values

Handle to the mapped memory area.

NULL signals mapping failed.

This value will be returned to *OS_UnmapUserPointer()* to invalidate and free the mapping.

6.2.9.2 OS_UnmapUserPointer

Unmap a previously mapped pointer.

Function Call

```
int OS_UnmapUserPointer(   void*      phMapping,
                           void*      pvOSDependet)
```

Arguments

Argument	Data type	Description
phMapping	void*	Handle returned from *OS_MapUserPointer()*
pvOsDependent	void*	OS dependent object that has been passed in the device instance during *cifXTKitAddDevice()*

Return Values

None zero return value indicates success.

6.3 USER implemented functions

Some functions must be implemented by the user to allow using of different file storages by the toolkit. Some cards are getting their firmware from the toolkit and need the appropriate files to be downloaded.

To allow the user to use flexible storages for these information's, several functions are predefined and called by the toolkit.

USER Functions	
USER_GetFirmwareFileCount	Get the number of firmware files to be downloaded to the hardware.
USER_GetFirmwareFile	Get the file information for a firmware file which should be downloaded to the hardware.
USER_GetConfigurationFileCount	Get the number of configuration files to be downloaded to the hardware.
USER_GetConfigurationFile	Get the file information for a configuration file which should be downloaded to the hardware.
USER_GetWarmstartParameters	Get the warm start parameters which should be downloaded to the hardware.
USER_GetAliasName	Get the alias name for a specific device.
USER_GetBootloaderFile	Get the bootloader file for a device
USER_GetInterruptEnable	Ask if the interrupt for a specific device should be enabled.
USER_GetOSFile	Get a base firmware filename (basically an rcX without any fieldbus stack running). Note: This is needed for loadable module support
USER_Trace	Do debug and error trace outputs
DMA Mode only	
USER_GetDMAMode	Ask if the DMA mode should be enabled / disabled on this card

Table 17: User implementation functions

6.3.1 USER_GetFirmwareFileCount

Retrieve the number of firmware files to be downloaded to a specific device and channel.

Function Call

```
uint32_t USER_GetFirmwareFileCount( PCIFX_DEVICE_INFORMATION ptDevInfo)
```

Arguments

Argument	Data type	Description
ptDevInfo	PCIFX_DEVICE_INFORMATION	Device information (Device/Serial number) and Channel to get number of firmware files for

Return Values

Number of files that can be queried by *USER_GetFirmwareFile()*.

6.3.2 USER_GetFirmwareFile

Retrieve the name of a firmware file for the given device.

Function Call

```
int USER_GetFirmwareFile (PCIFX_DEVICE_INFORMATION  ptDevInfo
                          uint32_t                  ulIdx,
                          PCIFX_FILE_INFORMATION    ptFileInfo)
```

Arguments

Argument	Data type	Description
ptDevInfo	PCIFX_DEVICE_INFORMATION	Device information (Device/Serial number) and Channel to get number of firmware files for
ulIdx	uint32_t	Number of firmware file (0..USER_GetFirmwareFileCount - 1)
ptFileInfo	PCIFX_FILE_INFORMATION	Returned file information

Return Values

None zero return value indicates success.

6.3.3 USER_GetConfigurationFileCount

Retrieve the number of configuration files to be downloaded to a specific device and channel.

Function Call

```
uint32_t USER_GetConfigurationFileCount(PCIFX_DEVICE_INFORMATION ptDevInfo)
```

Arguments

Argument	Data type	Description
ptDevInfo	PCIFX_DEVICE_INFORMATION	Device information (Device/Serial number) and Channel to get number of configuration files for

Return Values

Number of files that can be queried by *USER_GetConfigurationFile()*.

6.3.4 USER_GetConfigurationFile

Retrieve the name of a configuration file for the given device.

Function Call

```
int USER_GetConfigurationFile ( PCIFX_DEVICE_INFORMATION   ptDevInfo
                                uint32_t                   ulIdx,
                                PCIFX_FILE_INFORMATION     ptFileInfo)
```

Arguments

Argument	Data type	Description
ptDevInfo	PCIFX_DEVICE_INFORMATION	Device information (Device/Serial number) and Channel to get number of configuration files for
ulIdx	uint32_t	Number of configuration file (0..USER_GetConfigurationFileCount - 1)
ptFileInfo	PCIFX_FILE_INFORMATION	Returned file information

Return Values

None zero return value indicates success.

6.3.5 USER_GetWarmstartParameters

Return the filename for the warm start parameters. These parameters are saved in a binary file containing the warm start packet itself. Additionally to a header it includes also the fieldbus type and the total length of the message.

Retrieve the name of a warmstart configuration file for the given device.

Function Call

```
int USER_GetWarmstartParameters(    PCIFX_DEVICE_INFORMATION  ptDevInfo
                                    PCIFX_FILE_INFORMATION    ptFileInfo)
```

Arguments

Argument	Data type	Description
ptDevInfo	PCIFX_DEVICE_INFORMATION	Device information (Device/Serial number) and Channel to get warm start file for
ptFileInfo	PCIFX_FILE_INFORMATION	Returned file information

Return Values

None zero return value indicates success.

6.3.6 USER_GetAliasName

Return an alias name for the passed device. The alias name should be an empty string if no alias is to be assigned.

Function Call

```
void USER_GetAliasName(    PCIFX_DEVICE_INFORMATION   ptDevInfo
                           uint32_t                   ulMaxLen,
                           char*                      szAlias)
```

Arguments

Argument	Data type	Description
ptDevInfo	PCIFX_DEVICE_INFORMATION	Device information (Device/Serial number) and Channel to get alias for
ulMaxLen	uint32_t	Maximum length of alias
szAlias	char*	Buffer to receive assigned alias

6.3.7 USER_GetBootloaderFile

Return the path and filename to the cifX bootloader that is being loaded to a device if the reset is completed.

Function Call

```
void USER_GetBootloaderFile(    PDEVICEINSTANCE           ptDevInstance,
                                PCIFX_FILE_INFORMATION    ptFileInfo)
```

Arguments

Argument	Data type	Description
ptDevInstance	PDEVICEINSTANCE	Instance of the device requesting the bootloader. eChipType needs to be evaluated if different netX should be supported
ptFileInfo	PCIFX_FILE_INFORMATION	Returned file information

6.3.8 USER_GetInterruptEnable

This function is called from the toolkit to determine if the interrupt for the specified device should be enabled.

Function Call

```
int USER_GetInterruptEnable(PCIFX_DEVICE_INFORMATION ptDevInfo)
```

Arguments

Argument	Data type	Description
ptDevInfo	PCIFX_DEVICE_INFORMATION	Device information of the device, the interrupt enable flag is requested for

Return Values

None zero return value will enable the interrupt for the specified device.

6.3.9 USER_GetOSFile

This function is called from the toolkit to determine if a base firmware should be loaded to the specified device. This function is needed for loadable modules (.NXO files)

Function Call

```
int USER_GetOSFile(  PCIFX_DEVICE_INFORMATION  ptDevInfo,
                     PCIFX_FILE_INFORMATION    ptFileInfo)
```

Arguments

Argument	Data type	Description
ptDevInfo	PCIFX_DEVICE_INFORMATION	Device information
ptFileInfo	PCIFX_FILE_INFORMATION	Returned file data.

Return Values

Returns 0 if no OS file is configured.

When 0 is returned it will not be possible to use loadable modules (.NXO files).

6.3.10 USER_Trace

The toolkit can provide additional trace information like debug and error messages to the user. The amount of trace output is controlled through a global variable "*g_ulTraceLevel*".

The USER_Trace function is implemented by the user and will receive the trace level in the *ulTraceLevel* argument.

Variable	Data type	Description

g_ulTraceLevel	uint32_t	Control the amount of trace output. Valid values are:
		`0x00000001: TRACE_LEVEL_DEBUG`
		`0x00000002: TRACE_LEVEL_INFO`
		`0x00000004: TRACE_LEVEL_WARNING`
		`0x00000008: TRACE_LEVEL_ERROR`
		g_ulTraceLevel is evaluated using a bitwise `AND` operation.

Function Call

```
void USER_Trace(PDEVICEINSTANCE      ptDevInstance,
                uint32_t             ulTraceLevel,
                const char*          szFormat,
                ...)
```

Arguments

Argument	Data type	Description
ptDevInstance	PDEVICEINSTANCE	Device instance the trace is made for
ulTraceLevel	uint32_t	Trace level the message is output for
szFormat	string	printf() style format string
…		Variable argument list for printf

6.3.11 USER_GetDMAMode

This function is called from the toolkit to determine if the DMA for the specified device should be enabled.

Note: This function will only be called if *CIFX_TOOLKIT_DMA* is defined

Function Call

```
int USER_GetDMAMode(PCIFX_DEVICE_INFORMATION ptDevInfo)
```

Arguments

Argument	Data type	Description
ptDevInfo	PCIFX_DEVICE_INFORMATION	Device information of the device, the DMA mode is requested for

Return Values

Value	Definition	Description
0	eDMA_MODE_LEAVE	Don't change the current DMA mode on the card.
1	eDMA_MODE_ON	Automatically turn DMA mode on (if supported by firmware)
2	eDMA_MODE_OFF	Disable DMA during startup

7 Additional information

7.1 Special interrupt handling

7.1.1 Locking DSR against ISR

Depending on the interrupt handling of the operating system, it might be necessary to lock some code of the DSR routine against occurring device interrupts to ensure correct access to shared data.

To enable this feature it is necessary to implement the functions *OS_IrqLock()* and *OS_IrqUnlock()*, and setting the following pre-processor define:

```
#define CIFX_TOOLKIT_ENABLE_DSR_LOCK
```

7.1.1.1 OS_IrqLock

This functions needs to provide a lock against the interrupt service routine of the device. The easiest way is an IRQ lock but some operating systems provide a way to lock against a specific interrupt

Note: This function will only be called if *CIFX_TOOLKIT_ENABLE_DSR_LOCK* is defined

Function Call

```
void OS_IrqLock(void* pvOSDependent)
```

Arguments

Argument	Data type	Description
pvOSDependent	void*	OS dependent variable passed during *cifXTKitAddDevice()*

7.1.1.2 OS_IrqUnlock

This function re-enables the device's interrupt service routine.

Note: This function will only be called if *CIFX_TOOLKIT_ENABLE_DSR_LOCK* is defined

Function Call

```
void OS_IrqUnlock(void* pvOSDependent)
```

Arguments

Argument	Data type	Description
pvOSDependent	void*	OS dependent variable passed during *cifXTKitAddDevice()*

7.1.1.3 Sequence

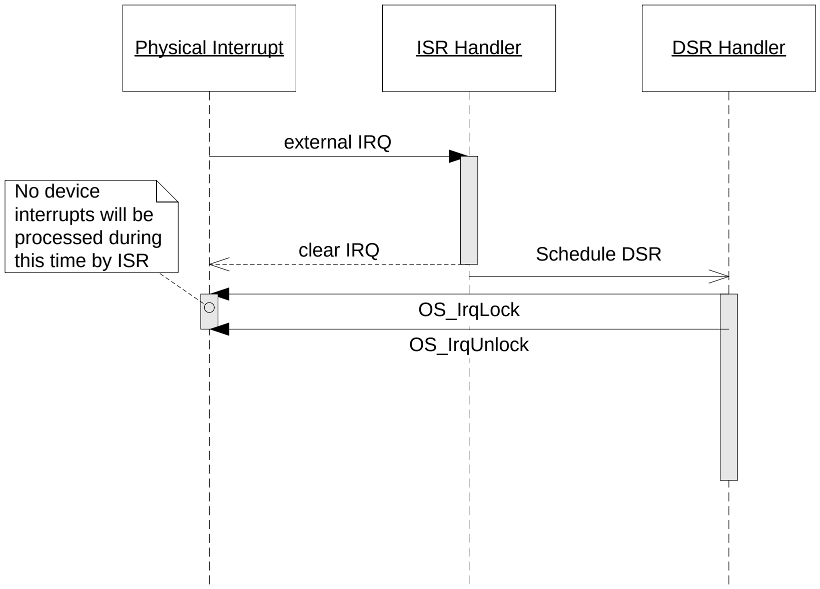

Figure 11: IRQ Handling with Locking

7.1.2 Deferred enabling of interrupts

Depending on the operating system it might be necessary to not enable the interrupts right within the *cifXTKitAddDevice()* call but at a later point.

In this case the following pre-processor define must be set:

```
#define CIFX_TOOLKIT_MANUAL_IRQ_ENABLE
```

Additionally the developer must call the functions *cifXTKitEnableHWInterrupt()* / *cifXTKitDisableHWInterrupt()* when the driver framework is ready to handle interrupts.

7.2 PCI device information

The cifX/net Toolkit does not offer PCI hardware detection functions because such functions are operating system dependent. Most common operating systems like Windows or Linux are Plug and Play aware and using own functionalities to detect PCI devices and their resources.

Writing an own PCI detection, at least the PCI Vendor and Device IDs for card detection and address and size of the dual port memory are necessary.

Note: The following information is only valid for netX 500/100 PCI or netX 4000 PCIe based devices

7.2.1 PCI/PCIe Vendor and Device IDs

Currently Defined PCI devices:

Device	Vendor ID	Device ID	SUB Vendor ID	SUB Device ID	Description
CIFX 50/70/80/90 CIFX104C	0x15CF	0x0000	0x0000	0x0000	Standard PCI and PCIe devices. (RAM-based only)
netPLC	0x15CF	0x0010	0x15CF	0x0000	RAM-based device
				0x0001	FLASH-based device
netJACK	0x15CF	0x0020	0x15CF	0x0000	RAM-based device
				0x0001	FLASH-based device
CIFX4000	0x15CF	0x4000	0x0000	0x0000	FLASH-based device

Table 18: Currently Defined PCI Devices

This definition and the recognition of FLASH or RAM-based is important, because the start-up handling differs for these devices.

Definition of RAM and FLASH-based devices:

- RAM-based device
  A RAM-based device does not store the bootloader, Firmware and configuration files in the device. On each power-up of such a device, all files must be downloaded to the device. A running firmware cannot be updated while the firmware is running. The device needs a hardware reset and a complete re-start to change the firmware. The user application is responsible to download the necessary files.
- FLASH-based device
  Flash-based devices use a Flash memory to store firmware and configuration files (in the device). A bootloader must be already on the hardware and must offer a standard Hilscher DPM to be able to download further files.

The toolkit offers a header file containing the necessary definitions:
Hilscher Vendor/Device ID definition **HilPCIDefs.h.**

7.2.2 BAR (Base Address Register) definition

PCI based devices are offering their hardware resources via the so called PCI Configuration space. The dual ported memory (DPM) physical address of a PCI based netX device can be determined by the PCI Base Address Registers (BARs).

<table>
<tr><th>31</th><th>16</th><th>15</th><th>0</th><th></th></tr>
<tr><td colspan="2">Device ID</td><td colspan="2">Vendor ID</td><td>00h</td></tr>
<tr><td colspan="2">Status</td><td colspan="2">Command</td><td>04h</td></tr>
<tr><td colspan="3">Class Code</td><td>Revision ID</td><td>08h</td></tr>
<tr><td>BIST</td><td>Header Type</td><td>Lat. Timer</td><td>Cache Line S.</td><td>0Ch</td></tr>
<tr><td colspan="4" rowspan="6">Base Address Registers</td><td>10h</td></tr>
<tr><td>14h</td></tr>
<tr><td>18h</td></tr>
<tr><td>1Ch</td></tr>
<tr><td>20h</td></tr>
<tr><td>24h</td></tr>
<tr><td colspan="4">Cardbus CIS Pointer</td><td>28h</td></tr>
<tr><td colspan="2">Subsystem ID</td><td colspan="2">Subsystem Vendor ID</td><td>2Ch</td></tr>
<tr><td colspan="4">Expansion ROM Base Address</td><td>30h</td></tr>
<tr><td colspan="3">Reserved</td><td>Cap. Pointer</td><td>34h</td></tr>
<tr><td colspan="4">Reserved</td><td>38h</td></tr>
<tr><td>Max Lat.</td><td>Min Gnt.</td><td>Interrupt Pin</td><td>Interrupt Line</td><td>3Ch</td></tr>
</table>

The dual ported memory (DPM) of netX500/100 PCI devices is provided via BAR 0 (Base Address Register 0, Offset 0x10).

Name	Offset	Definition Name	Description
BAR 0	0x10	DPM_BASE_ADDRESS	Dual Port Memory
BAR 1	0x14	TARGET_BASE_ADDRESS	MRAM area, if supported by the hardware
BAR 2	0x18	I/O_BASE_ADDRESS	Special netX feature, currently not implemented
BAR 3	0x1C	-/-	unused
BAR 4	0x20	-/-	unused
BAR 5	0x24	-/-	unused

Table 19:BAR - Base Address Register Overview

Note: The PCI configuration space is a standard PCI functionality and described in the PCI specification

7.2.3 Determine the size of PCI memory resources

Plug and Play aware operating systems are offering a driver PCI resource information by default.

Using an none Plug and Play aware operating system, the information can be determined by using the following procedure:

1. save the current value of the "Base Address Register" (this is the physical memory address)
2. write a 0xFFFFFFFF pattern to the "Base Address Register"
3. read back the content of the Base Address Register (this contains the size information)
4. restore the original value of the Base Address Register
5. compute the size of the memory region by using the previous read size information. This is done by masking out the lowest 4 bit (for a memory BAR) and building the 2 complement of the value (invert the value and add 1).

```
if (val & 1)
  size = (~val | 0x3) + 1;/* I/O space */
else
  size = (~val | 0xF) + 1;/* memory space */
```

   The resulting value is the memory size in bytes.

Note: The lowest bit in an memory size information defines the type of the resource (1 = I/O space, 1 = memory space).
The lowest 2 Bits in an I/O space and the 4 lowest Bits in a memory space are having special meanings and should be set to 0, when calculation the size.

Note: Determining the size of a PCI memory resource region is a standard PCI functionality and described in the PCI specification

7.2.4 Enable interrupt on PCI-based hardware

By default, a PCI device should only generate an interrupt if the user application (e.g. device driver) has already registered an interrupt service routine for the specific interrupt.

Because of this definition, the interrupt of a netX based PCI device is disabled by default. To enable the interrupt, a corresponding interrupt mask must be written to the netX "*Global Register Block*".

This register block is located at the last 512 bytes of the netX dual ported memory and the structure of the netX "*Global Register Block*" is defined in **NetX_RegDefs.h.**

The interrupt control registers (*ulIRQEnable_0* and *ulIRQEnable_1)* can be found in the netX "*Host Control Block*", which is a part of the netX "*Global Register Block*".

```
/*****************************************************************************/
/*! netX Host Register Block, always located at Offset DPMSize - 0x200       */
/*****************************************************************************/
typedef struct NETX_GLOBAL_REG_BLOCKtag
{
  /* 0xFE00, start of the DMA channel data (8Channels * 8DWords * 4Bytes/DWord = 0x100
Bytes) */
  NETX_DMA_CHANNEL_CONFIG atDmaCtrl[NETX_MAX_DMA_CHANNELS]; /*!< Configuration Register
for all 8 DMA Channels */

  /* 0xFF00, start of the netX Host control block */
  volatile uint32_t reserved[47];           /*!< unused/reserved */

  /* 0xFFBC, start of the defined registers */
  volatile uint32_t ulPCIBaseAddress;       /*!< PCI Base address of 2nd Memory Window */
  volatile uint32_t ulWatchDogTimeoutHost;  /*!< Host Watchdog Timeout value          */
  volatile uint32_t ulWatchDogTrigger;      /*!< Host Watchdog triggering cell        */
  volatile uint32_t ulWatchDogTimeoutNetx;  /*!< NetX Watchdog Timeout value          */
  volatile uint32_t reserved2;              /*!< unused/reserved                      */
  volatile uint32_t ulCyclicTimerControl;   /*!< Control of cyclic timer (repeat/single,
                                                 timer resolution, up/down)           */
  volatile uint32_t ulCyclicTimerStart;     /*!< Timer start value                    */
  volatile uint32_t ulSystemState;          /*!< System state register                */
  volatile uint32_t ulHostReset;            /*!< Host reset for initiating a hard reset
                                                 of the netX chip                     */
  volatile uint32_t ulIRQState_0;           /*!< IRQ State 0                          */
  volatile uint32_t ulIRQState_1;           /*!< IRQ State 1                          */
  volatile uint32_t reserved3;              /*!< unused/reserved                      */
  volatile uint32_t reserved4;              /*!< unused/reserved                      */
  volatile uint32_t ulIRQEnable_0;          /*!< IRQ enable register 0                */
  volatile uint32_t ulIRQEnable_1;          /*!< IRQ enable register 1                */
  volatile uint32_t reserved5;              /*!< unused/reserved                      */
  volatile uint32_t reserved6;              /*!< unused/reserved                      */
} NETX_GLOBAL_REG_BLOCK,*PNETX_GLOBAL_REG_BLOCK;
```

The Toolkit offers the functions *cifXTKitEnableHWInterrupt() / cifXTKitDisableHWInterrupt()* which implementing the enable / disable procedure.

8 Toolkit low-level hardware access functions

The toolkit is layered into the hardware functions (DPM functions) and the managing functions above the hardware layer. For very small systems like 8 bit microcontrollers, without an operating system, it is also possibly to only use the hardware functions module.

Note: These functions are intended to use with FLASH based netX hardware (comX) and can not be used for RAM based PCI hardware (cifX)! Because of the complexity of starting such a PCI hardware!

The following figure shows access to the DPM only with the toolkit's hardware functions.

The *Generic Interrupt Handler* provides access in interrupt mode. The *OS Specific Functions Module* abstracts the target specific functions, which makes it easier to port. Together these three modules build the *Low Level Interface*.

Overview:

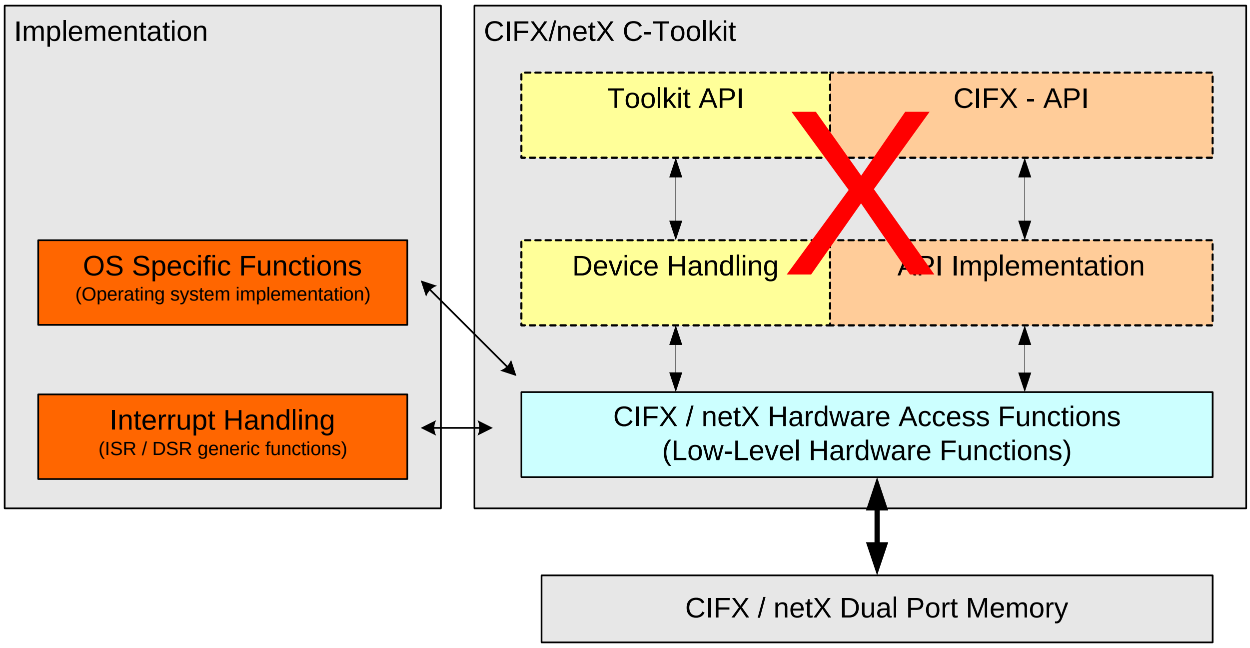

Figure 12: Hardware Function Layer

The following sections explain which files are necessary to build the *Low Level Interface*, how to initialize and use the *Toolkit Hardware Function Module*.

8.1 Function overview

The following table shows important *Toolkit Hardware Functions*. For information about the unlisted functions or more detailed information are available in the corresponding source and header files (cifXHWFunctions.c / cifXHWFunctions.h).

Hardware functions	Descriptions
Status Functions-	
DEV_IsReady()	Read COS flags and checks if channel is ready.
DEV_IsRunning()	Read COS flags and checks if channel is ready.
DEV_IsCommunicating()	Checks if channel is communicating.
DEV_GetHostState()	Returns the channel's application COS flags.
DEV_SetHostState()	Sets the channel's application COS flags.
DEV_BusState()	Set the channels COS bus flags and returns the resulting state.
DEV_CheckCOSFlags()	Checks and updates COS flags over all channels.
DEV_GetHandshakeBitState()	Reads handshake cells (->DEV_ReadHandshakeFlags()).
...	...
Initialization Functions-	
DEV_DoChannelInit()	Performs a channel init and checks after given timeout expected state.
DEV_DoSystemStart()	Performs a system restart and checks after given timeout expected state.
...	...
Communication Functions	
DEV_GetMBXState()	Returns state of device mailbox.
DEV_TransferPacket()	Transfer packet over given channel and returns received packets (->DEV_GetPacket()/->DEV_PutPacket()).
...	...

Table 20: Toolkit hardware functions

8.2 Using the Toolkit hardware functions

This chapter shows how to use the *Toolkit's Hardware Functions*.

The *Hardware Functions* are located in the cifXHWFuntions.c and cifXHWFuntions.h file and these low level functions expecting just a filled DEVICEINSTANCE and CHANNELINSTANCE structure to be usable.

The required toolkit files, needed to use the hardware functions are listed below:

- cifXHWFunctions.c
- cifXInterrupt.c
- cifXEndianess.h
- cifXErrors.h
- cifXHWFunctions.h
- cifXUser.h
- NetX_RegDefs.h
- OS_Dependent.h
- rcX_User.h
- TLR_Types.h

As the Hardware Function module uses some functionality which are potentially operating system or compiler depending, the OS Abstraction Layer must be implemented (see section *OS Abstraction* on page 60).

The only user environment specific function which is used by the hardware functions module is USER_Trace(), and thus must be implemented by the user (see section *USER_Trace* on page 90).

```
void USER_Trace (PDEVICEINSTANCE ptDevInstance, uint32_t ulTraceLevel,
                 const char* szFormat, ...)
```

As the trace level is external referenced by the *Hardware Function Module*, the trace level variable must be globally defined by the user.

```
uint32_t g_ulTraceLevel = 0;
```


8.3 Simple C application

The simple C-Source example shows how to identify a mapped DPM area (dual port memory) and retrieve the system and communication channel states.

The following table demonstrates the flow of the example program. The direction to read is from the top to the bottom. According to that the first line in the table shows the first command line out of the example source. In case of developing a user application the table shows the right order of the command flow.

The left row, the so called *User Implemented Functions,* need to be implemented by the user, because of its target dependency. The *Toolkit Hardware Functions* are a set of functions which are available in the *Toolkit Hardware Function Module* (see ***Function overview*** on page 100).

Example Program Structure		
User Implemented Functions	**Toolkit Hardware Functions**	**Description**
cifXTkHWFunctions_GetDPMPointer()		Retrieve pointer to DPM area.
	OS_Memcmp()	Validate DPM signature.
	DEV_Initialize()	Initialize DEVICEINSTANCE and CHANNELINSTANCE structures.
	DEV_ReadHostFlags()	Read the host flags of system and communication channel to synchronize internal states.
	DEV_IsReady()	Check if system and communication channel are ***Ready***.
	DEV_IsRunning()	Check if communication channel is running. If device is not ***Running*** the device needs to be configured. A configuration can be send through a *Warmstart* packet.
	DEV_IsCommunicating()	Check if device is ***Communicating***. If the communication channel is ***Communicating*** the configured IOs (see CHANNELINSTANCE) of the specified channel are available.
...	...	do anything
...	...	...
cifXTkHWFunctions_FreeDPMPointer()		At the end of the program release the mapped DPM area or other initialized resources.

Table 21: Example Program Structure

C-source example:

```
DEVICEINSTANCE* ptDevInstance;
BYTE*           pbDPM;

/* get the DPM pointer */
if (cifXTkHWFunctions_GetDPMPointer (&pbDPM, &ulDPMSize))
{
  /* setup initialize structure */

  /* Initialize device instance */
  if (CIFX_NO_ERROR != (cifXTkHWFunctions_InitializeDataStructures(
                               pbDPM, ulDPMSize, ptDevInstance, 10000)))
  {
    return DEV_ERROR;
  } else
  {
    CHANNELINSTANCE* ptChan = ptDevInstance->pptCommChannels[COM_CH];
    DEV_ReadHostFlags(&ptDevInstance->tSystemDevice, 0);
    DEV_ReadHostFlags(ptChan, 0);

    /* check if system device is ready... */
    if (!DEV_IsReady(&ptDevInstance->tSystemDevice))
    {
      return DEV_ERROR;
    /* Check if communication channel is ready... */
    } else if (!DEV_IsReady(ptChan))
    {
      return DEV_ERROR;
    } else /* device is ready */
    {
      if (!DEV_IsRunning(ptChan))
      {  /* configure device */
        IdentifyWarmstartPacket(ptChan,&tSndPack);
        DEV_TransferPacket(ptChan,&tSndPack,&tRecPack,PACKSIZE,TIMEOUT,0,0);

        DEV_DoChannelInit(ptDevInstance->pptCommChannels[COM_CH], TIMEOUT);
        /* Waiting for netX warmstarting */
        do
        {
          lRet = DEV_SetHostState(pChannel, CIFX_HOST_STATE_READY, 1000);
        } while (CIFX_DEV_NOT_RUNNING == lRet);
      }
      /* check if device is communicating */
      if(!DEV_IsCommunicating(ptDevInstance->pptCommChannels[COM_CH],&lRet))
      {
        /*... do anything */
        ...
```


- First of all the DPM pointer needs to be retrieved. In the example the function *cifXTkHWFunctions_GetDPMPointer()* returns a pointer to the DPM and the size of the mapped area. This function needs to be customized. The pointer can be validated by checking the DPM signature.

Note:	Retrieving the DPM pointer is completely target dependant (platform, OS, ...) and thus *cifXTkHWFunctions_GetDPMPointer()* is not a standard *Toolkit Hardware Function* and needs to be implemented!

- After retrieving the DPM pointer the *DEVICEINSTANCE* and *CHANNELINSTANCE* structure needs to be filled. *cifXTkHWFunctions_InitializeDataStructures()* sets up the *DEVICEINSTANCE* structure. Information about the structure can be found in section *DEVICEINSTANCE structure* on page 46.

Note:	*cifXTkHWFunctions_InitializeDataStructures()* is not a standard *Toolkit Hardware Function*. An example implementation for the *Standard DPM Layout* is delivered with the *cifX Toolkit* source. For custom layouts the function needs to be adapted.

- Before retrieving one of the various system and channel flags, synchronize the internal states. This can be done by reading the host flags over *DEV_ReadHostFlags*().

Note:	First, synchronize the internal states over *DEV_ReadHostFlags*(). It is not possible to retrieve flags from none existing channels (channel must be at least ***Ready***).

- To send a packet (e.g. via *DEV_TransferPacket()*) to a specified channel, the state of corresponding channel must be ***Ready***. A channel state request can be performed by *DEV_IsReady()*.
- In case *DEV_IsRunning()* returns ***False***, the configuration is missing. Now it is possible to send a configuration via *DEV_TransferPacket()*. *IdentifyWarmstartPacket()* identifies the running FW on channel *COM_CH* and configures the packet *tSndPack*. After sending the configuration the channel needs to be initialized, by calling *DEV_DoChannelInit()*.

Note:	The Warmstart configuration packet is FW specific and therefore IdentifyWarmstartPacket*()* is not a standard *Toolkit Hardware Function*, and thus needs to be implemented!

- If *DEV_IsCommunicating()* returns ***True***, the input and output data are available. Assumed the device's IO areas are configured (see section *CHANNELINSTANCE structure* on page 49).

Note:	General information over state changes, status flags or transferring packets can be found in the Hilscher Gesellschaft für Systemautomation mbH: Dual-Port Memory Interface Manual, Revision 15, english, 201 ([2]).

8.4 The Toolkit C example application

The *Toolkit C Example* is a complete application covering the device startup and configuration. It is located on the toolkit CD and can be used as a starting point and basis for an own implementation.

```
/*****************************************************************************/
/*! Hardware function example
 * \return 0 on success                                                       */
/*****************************************************************************/
int32_t cifXHWSample( void)
{
  int32_t  lDemoRet  = DEV_NO_ERROR;
  int32_t  lRet      = CIFX_NO_ERROR;

  uint8_t*          pbDPM         = NULL;       /* This pointer must be loaded to the DPM address */
  uint32_t          ulDPMSize     = 0;          /* Size of the DPM in bytes */
  DEVICEINSTANCE    tDevInstance;               /* Global deveice data structure used by all DEV_xxx functions */

  /* Get pointer to the hardware dual-port memory and check if it is available */
  if ( FALSE == cifXTkHWFunctions_GetDPMPointer( &pbDPM, &ulDPMSize))
    /* Failed to get the hardware DPM pointer and size */
    return -1;

#ifdef CIFX_TOOLKIT_HWIF
  tDevInstance.pfnHwIfRead  = cifXHwFnRead;    /* relizes read access to the system dependant DPM interface  */
  tDevInstance.pfnHwIfWrite = cifXHwFnWrite;   /* relizes write access to the system dependant DPM interface */
#endif

  /* Initialize the necessary data structures */
  if ( DEV_NO_ERROR == cifXTkHWFunctions_InitializeDataStructures( pbDPM, ulDPMSize, &tDevInstance, 10000))
  {
    /*---------------------------------------------------------------------*/
    /* Read actual device states                                           */
    /*---------------------------------------------------------------------*/
    PCHANNELINSTANCE ptSystemDevice = &tDevInstance.tSystemDevice;
    PCHANNELINSTANCE ptChannel      = tDevInstance.pptCommChannels[COM_CHANNEL];

    /* Wait for State acknowlede by the firmware */
    OS_Sleep(100);              /* Wait a bit */

    /* read the host flags of the system device, first time to synchronize our internal status */
    DEV_ReadHostFlags( ptSystemDevice, 0);

    /* read the host flags of the communication channel, first time to synchronise our internal status */
    DEV_ReadHostFlags( ptChannel, 0);

    /* check if "system device" is ready... */
    if (!DEV_IsReady( ptSystemDevice))
    {
      /* System device is not ready! */
      lDemoRet = ERR_DEV_SYS_READY;

    /* check if "communication channel" is ready... */
    } else if ( !DEV_IsReady(ptChannel))
    {
      /* Communication channel is not ready! */
      lDemoRet = ERR_DEV_COM_READY;

    } else
    {
      /*---------------------------------------------------------------------*/
      /* At this point we should have a running device and a configured         */
      /* communication channel.                                                 */
      /* Procced with "NORMAL Stack Handling!                                   */
      /*---------------------------------------------------------------------*/
      /* Signal Host application is available */
      lRet = DEV_SetHostState( ptChannel, CIFX_HOST_STATE_READY, 1000);

      /* Configure the device */
      lDemoRet = cifXTkHWFunctions_ConfigureDevice( ptChannel, ptSystemDevice);
      //if( DEV_NO_ERROR != lDemoRet)
      //  printf("Error");

      /* Initialize and activate interrupt if configured */
      DEV_InitializeInterrupt ( &tDevInstance);

      if (DEV_NO_ERROR == lDemoRet)
      {
        /*---------------------------------------------------------------------*/
        /* At this point we should have a running device and a configured         */
        /* communication channel if no error is shown                             */
        /*---------------------------------------------------------------------*/
        uint32_t ulState = 0;

        /* Signal Host application is available */
        lRet = DEV_SetHostState( ptChannel, CIFX_HOST_STATE_READY, 1000);

        /* Switch ON the BUS communication */
        lRet = DEV_BusState( ptChannel, CIFX_BUS_STATE_ON, &ulState, 3000);

        /* TODO: Decide to wait until communication is available or just go to */
        /*       to the cyclic data handling and check the state there         */
        /* Wait for communication is available or do this during the cyclic program handling*/
        lDemoRet = cifXTkHWFunctions_WaitUntilCommunicating( ptChannel);

        /*---------------------------------------------------------------------*/
        if (lDemoRet == DEV_NO_ERROR)
        {
```

```
          /* device is "READY", "RUNNING" and "COMMUNICATING" */
          /* Start cyclic demo with I/O Data-Transfer and packet data transfer */
          unsigned long ulCycCnt = 0;
          //uint32_t ulTriggerCount = 0;

          /* Cyclic I/O and packet handling for 'ulCycCnt'times */
          while( ulCycCnt < DEMO_CYCLES)
          {
            /* Start and trigger watchdog function, if necessary */
            //DEV_TriggerWatchdog(ptChannel, CIFX_WATCHDOG_START, &ulTriggerCount);

            /* Handle I/O data transfer */
            IODemo      (ptChannel);

            /* Handle rcX packet transfer */
            #ifdef FIELDBUS_INDICATION_HANDLING
              Fieldbus_HandleIndications( ptChannel);
            #else
              PacketDemo ( ptChannel);
            #endif

            ulCycCnt++;
          }

          /* Stop watchdog function, if it was previously started */
          //DEV_TriggerWatchdog(ptChannel, CIFX_WATCHDOG_STOP, &ulTriggerCount);
        }

        /* Switch OFF the BUS communication / dont't wait */
        lRet = DEV_BusState( ptChannel, CIFX_BUS_STATE_OFF, &ulState, 0);

        /* Signal Host application is not available anymore / don't wait */
        lRet = DEV_SetHostState( ptChannel, CIFX_HOST_STATE_NOT_READY, 0);
      }

      /* Uninitialize interrupt */
      DEV_UninitializeInterrupt ( &tDevInstance);
    }
  }

  /* Cleanup all used nenory areas and pointers */
  cifXTkHWFunctions_UninitializeDataStructures( &tDevInstance);

  /* cifXTkHWFunctions cleanup */
  cifXTkHWFunctions_FreeDPMPointer( pbDPM);

  return lDemoRet;
}
```


8.5 Toolkit hardware functions in interrupt mode

It is possible to use the *Toolkit Hardware Functions* either in *Polling Mode* or in *Interrupt Mode*. A *Generic Interrupt Handler* is integrated in the *Hardware Function Module* (see cifXTKitISRHandler() and cifXTKitDSRHandler()). The source is located in the *cifXInterrupt.c* file.

Information about the interrupt service routines can be found under section *Interrupt handling* on page 33 and the corresponding functions (*ISR* and *DSR Handler*) and section *Special interrupt handling* on page 92.

Use of the toolkit's hardware functions in interrupt mode requires initialization of all interrupt resources in the *DEVICEINSTANCE* and *CHANNELINSTANCE* structure.

DEVICEINSTANCE	
Variable	**Description**
fIrqEnabled	Set to true to signal irq mode enabled.
iIrqToDsrBuffer	Indicates which buffer to use in atIrqToDsrBuffer.
atIrqToDsrBuffer	Two synchronisation buffers (copy of handshake flags):
ulIrqCounter	Irq counter.

CHANNELINSTANCE	
Variable	**Description**
ahHandshakeBitEvents	Array of handles for signaling differant events (e.g. bus state...).
tSynch	Handles to synchronization objects.

Further it is necessary to implement additional OS functions such as locking functions or event signaling and its complements (e.g. *OS_Lock()*, *OS_SetEvent()*...). The use of the notification callback of IO areas is optional (see CHANNELINSTANCE). If it is not used it is necessary to implemet an alternative way to process the *IO Area*.

Of course to use the interrupt mode, the service routines must be installed according to the target system (platform, OS, ...).

For more detailed information about what is needed to be initialized see in *cifXInterrupt.c*.

Note: To use the interrupt service routines, the different handler need to be registered or installed. The ISR control mechanism depends on the target system and need to be implemented according to it!
For information of the resources, which need to be initialized to operate in interrupt mode, see section *DEVICEINSTANCE structure* on page 46 and the in ISR routine itself.

9 Error codes

Error code	Definition and description
0x00000000	CIFX_NO_ERROR No error

Error code	Definition and description
0x800A0001	CIFX_INVALID_POINTER Invalid pointer (e.g. NULL) passed to driver
0x800A0002	CIFX_INVALID_BOARD No board with the given name / index available
0x800A0003	CIFX_INVALID_CHANNEL No channel with the given index available
0x800A0004	CIFX_INVALID_HANDLE Invalid handle passed to driver
0x800A0005	CIFX_INVALID_PARAMETER Invalid parameter
0x800A0006	CIFX_INVALID_COMMAND Invalid command
0x800A0007	CIFX_INVALID_BUFFERSIZE Invalid buffer size
0x800A0008	CIFX_INVALID_ACCESS_SIZE Invalid access size
0x800A0009	CIFX_FUNCTION_FAILED Function failed
0x800A000A	CIFX_FILE_OPEN_FAILED File cannot not be opened
0x800A000B	CIFX_FILE_SIZE_ZERO File size is zero
0x800A000C	CIFX_FILE_LOAD_INSUFF_MEM Insufficient memory to load file
0x800A000D	CIFX_FILE_CHECKSUM_ERROR File checksum comparison failed
0x800A000E	CIFX_FILE_READ_ERROR Error while reading file
0x800A000F	CIFX_FILE_TYPE_INVALID Invalid file type
0x800A0010	CIFX_FILE_NAME_INVALID Invalid file name
0x800A0011	CIFX_FUNCTION_NOT_AVAILABLE Driver function not available
0x800A0012	CIFX_BUFFER_TOO_SHORT Given buffer too short
0x800A0013	CIFX_MEMORY_MAPPING_FAILED Memory mapping failed
0x800A0014	CIFX_NO_MORE_ENTRIES No more entries available

Error code	Definition and description
0x800A0015	CIFX_CALLBACK_MODE_UNKNOWN Unknown callback handling mode
0x800A0016	CIFX_CALLBACK_CREATE_EVENT_FAILED Creation of callback events failed
0x800A0017	CIFX_CALLBACK_CREATE_RECV_BUFFER Creation of callback receive buffer failed
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

Table 22: Error codes (0x800Axxxx)

Error code	Definition and description
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
0x800B0030	CIFX_DRV_DRIVER_NOT_LOADED CIFx driver is not running
0x800B0031	CIFX_DRV_INIT_ERROR Initialization of device failed
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
0x800B0040	CIFX_DRV_DOWNLOAD_STORAGE_UNKNOWN Unknown download storage type (RAM/FLASH based) found
0x800B0041	CIFX_DRV_DOWNLOAD_FW_WRONG_CHANNEL Channel number for a firmware download not supported
0x800B0042	CIFX_DRV_DOWNLOAD_MODULE_NO_BASEOS Modules are not allowed without a Base OS firmware

Table 23: Error codes (0x800Bxxxx)

Error code	Definition and description
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 too short
0x800C0020	CIFX_DEV_RESET_TIMEOUT Reset command timeout
0x800C0021	CIFX_DEV_NO_COM_FLAG COM-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

Error code	Definition and description
0x800C002D	CIFX_DEV_INITIALIZATION_TIMEOUT Timeout during channel initialization
0x800C002E	CIFX_DEV_BUS_STATE_ON_TIMEOUT 'Set Bus ON' Timeout
0x800C002F	CIFX_DEV_BUS_STATE_OFF_TIMEOUT 'Set Bus OFF' Timeout
0x800C0040	CIFX_DEV_MODULE_ALREADY_RUNNING Module already running
0x800C0041	CIFX_DEV_MODULE_ALREADY_EXISTS Module already exists
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 256 Byte)
0x800C0052	CIFX_DEV_DMA_BUFFER_TOO_BIG DMA buffers size too big (max. size 63.75 KByte)
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' Timeout

Table 24: Error codes (0x800Cxxxx)

10 Appendix

10.1 List of tables

10.2 List of figures

10.3 Legal notes

Copyright

Important notes

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

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

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

Liability disclaimer

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

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

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

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

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

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

Warranty

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

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

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

Costs of support, maintenance, customization and product care

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

Additional guarantees

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

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

Confidentiality

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

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

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

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

Export provisions

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

10.4 Contacts

Headquarters

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

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

Subsidiaries

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

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

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

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

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

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

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

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

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

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

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

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

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

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