Driver Manual

cifX Device Driver

Linux

V2.1.1.0

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC090201DRV14EN | Revision 14 | English | 2022-02 | Released | Public

Table of contents

1 Introduction

1.1 About this document

This manual describes the Hilscher cifX driver for Linux and its architecture.

The driver offers access to the Hilscher netX-based hardware (e.g. CIFX50) with the same functional API as the cifX Device Driver for Windows®.

1.2 List of revisions

Rev	Date	Name	Chapter	Revision
10	2019-02-13	SD	-	cifX Device Driver V1.2.0.0.
			1.4, 1.5, 1.7, 3.4.2, 3.7	Description updated according to V1.2.0.0 (build tool CMake).
			4.3.3	Section *The plugin interface of the driver* added.
			6.2.3	Section *Using the Device Tree* added.
			6.3.2	Section *Using the SPI plugin (Linux spidev framework)* added.
11	2019-09-11	SD	3.4.2.2	Added missing header files according to V2.0.0.0
			3.7	Section *Firmware and configuration file storage* updated; Section *Device identification via device name*: device name method added. Subsection reordered.
12	2019-12-03	SD	4	Comment about severity mapping added.
			4.1.1	New element ‘logfd’ in structure added.
13	2021-08-05	SD	3.4	Order of listed steps changed according to how they appear in the document.
			3.4.2.2 3.5.2	IDE description to use Makefile as project base updated.
			6.3.2	IRQ settings description added.
14	2022-02-04	SD	-	Version 2.1.1.0
			3.7	Removed limitation to one channel.
			4.6.2	Removed requirement of a cifX PCIe device.

Table 1: List of revisions

1.3 Overview

The cifX Linux driver runs as a library in userspace and accesses the card via a UIO kernel module (Userspace I/O).

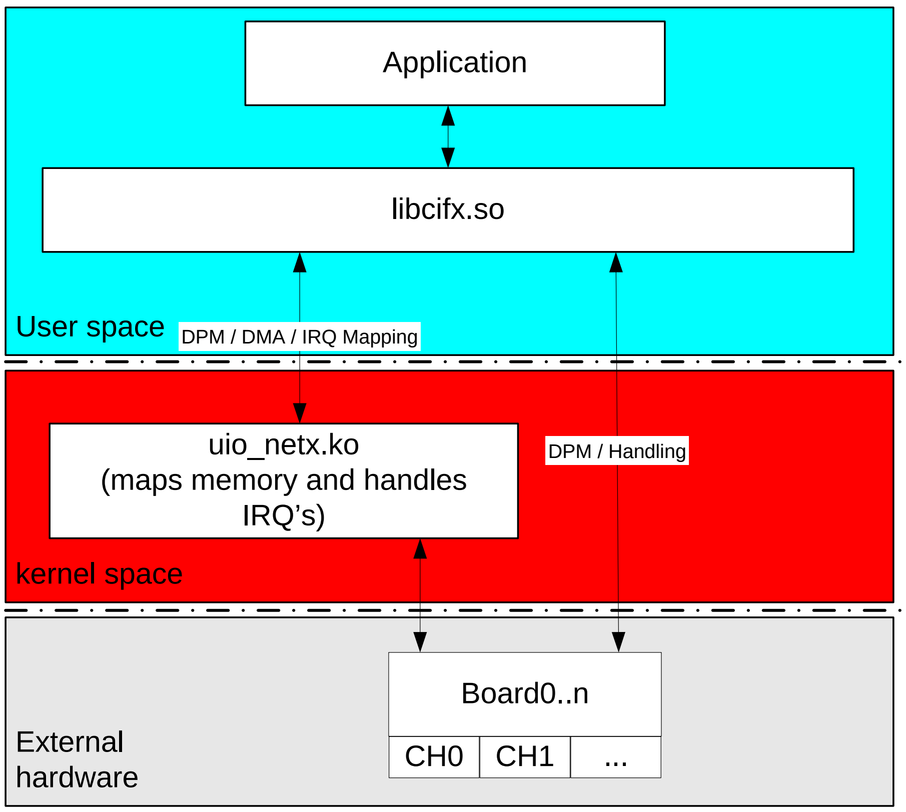


Figure 1: Linux cifX driver architecture

1.4 Requirements

Mandatory

- Linux Kernel Source (for PCI cards or other devices which rely on uio_netx)
- libpthread, librt
- CMake (min 2.8.9)

Optional

- Linux standard libraries *libpciaccess* (tested with V0.10.2 / V0.13.1-2)
  - always needed for cifX PCI cards, support can be disabled by defining *CIFX_TOOLKIT_DISABLEPCI*)
- *Optional: pkg-config* utility for automatic finding/configuring needed libraries

Note: The support of autotools build method is discontinued.
Starting with release V2.0.0.0 of the cifX Device Driver for Linux, CMake is supported only.

1.5 Features

- Unlimited number of cifX boards supported
- Support for NXSB-PCA or NX-PCA-PCI, netPLC, netJACK boards included
- Interrupt notification for applications
- Support of second Memory Window for PCI-based device (e.g. MRAM)
- Setting the device time during start-up if time handling is supported by the device
- DMA Support
- Support of a Virtual cifX Ethernet Interface (see section *netX-based virtual Ethernet interface* on page 54)
- uio_netx driver supports custom memory mapped devices
  (e.g. DPM, ISA, or other non PCI devices)
- Interrupt support for ISA devices (when using uio_netx with custom device)
- Simple integration of custom hardware interface
- Simple access of devices via SPI plugin (support for Linux spidev framework devices)
- uio-netX kernel module provides support for device tree initialization
- Building with *Eclipse*
- Example recipes for Yocto buildenvironment

1.6 Limitations

- No Interrupt support for NXSB-PCA and NX-PCA-PCI boards
- On big-endian machines the user is responsible for converting send/receive packets from/to little endian. This is **NOT** automatically done inside the driver / toolkit.
- One application can access a card simultaneously only. For multi-application access to a single card, a special application needs to be implemented by user
- Online diagnostics access via SYCON.net needs a TCP/IP Server functionality integrated into the user application. An example stand alone server is offered with the Linux driver.
- **libcifx (Toolkit) needs to run as 'root' or with a user that has the following rights:**
  - **read/write access to the PCI configuration registers (i.e. '/sys/class/uio/uio<n>/device/config')**
  - **Mapping of DPM to user space (see 'mmap' and 'ulimit -l')**
  - **read/write access to devices '/dev/uio<n>'**
  - **read/write access to /dev/mem (for user added devices)**

1.7 CD contents

Folder		Content
Documentation		Driver documentation
driver		
	libcifx	cifX Linux driver source (CMake project / Eclipse project)
	plugins	plugin source (for hw-read/write plugins & plugin for netX SPI access) (CMake project / Eclipse project)
	uio_netx	netx uio driver sources
	BSL	boot loader files
	scripts	installation scripts for the uio_netx kernel module
	templates	templates for several device configurations including device tree
examples		cifX example application
	cifxsample	Small example application, demonstrating driver initialization and toolkit usage (autoconf project / Eclipse project)
	cifXTCPServer	Example stand alone TCP server for SYCON.net diagnostic access (autoconf project / Eclipse project)
	cifXTestConsole	Demo application for testing toolkit functions (autoconf project / Eclipse project)
	LoadModules	Example application, demonstrating firmware module loading. (Eclipse project / Makefile)
	ISASample	Small application, demonstrating the initialization of an ISA device via User Space library libcifx
	SPISample	Small application, demonstrating the initialization of an SPI device via User Space library libcifx
Diagnostic and remote Access		Documentation, example and sources for the netX diagnostic and remote access

Table 2: CD contents

1.8 Terms, abbreviations and definitions

Term	Description
cifX	**C**ommunication **I**nter**f**ace based on netX
comX	**Co**mmunication **M**odule based on netX
PCI	**P**eripheral **C**omponent **I**nterconnect
UIO	**U**serspace **I/O**
API	**A**pplication **P**rogramming **I**nterface
DPM	**D**ual-**P**ort **M**emory Physical interface to all communication board **Note:** DPM is also sometimes used for PROFIBUS-**DP M**aster

Table 3: Terms, abbreviations and definitions

1.9 References to documents

This document refers to the following documents:

[1]	Hilscher Gesellschaft für Systemautomation mbH: Programming reference guide, CIFX API, DOC121201PR09EN, Revision 9, English, 2020.
[2]	Hilscher Gesellschaft für Systemautomation mbH: Driver Manual, cifX Device Driver, Windows XP/Vista/7/8/10/11, DOC060701DRV31EN, Revision 31, English, 2022.
[3]	Hilscher Gesellschaft für Systemautomation mbH: Protocol API, Ethernet, Packet interface, DOC060901API11EN, Revision 11, English, 2020.
[4]	Hilscher Gesellschaft für Systemautomation mbH: Toolkit Manual, cifX/netX Toolkit, DPM, V2.1, DOC090203TK11EN, Revision 11, English, 2019.
[5]	Hilscher Gesellschaft für Systemautomation mbH: Packet API, netX Dual-Port Memory, Packet-based services (netX 10/50/51/52/100/500), DOC161001API05EN, Revision 5, English, 2021.
[6]	Hilscher Gesellschaft für Systemautomation mbH: Packet API, netX Dual-Port Memory, Packet-based services (netX 90), DOC190301API06EN, Revision 6, English, 2021.

Table 4: References to documents

2 Licensing terms

The Hilscher cifX Linux driver consists of several modules.

- **uio_netx** Offered by Hilscher Gesellschaft für Systemautomation mbH

  The latest version of the *uio_netx* kernel module is located on the CD.

  This module is licensed under GPL V2 and can be used under these terms.

- **libcifx** Offered by Hilscher Gesellschaft für Systemautomation mbH

  This library is a userspace library and an intellectual property of the
  Hilscher Gesellschaft für Systemautomation mbH.

  The source code and library can be used for internal development, modification and debugging purpose.

  Distribution of the original **libcifx** source code, parts of the **libcifx** source code or modifications based on it is prohibited.

  Binary distribution for use in products is allowed.

This software program is protected under German and international copyright law as well as under international laws and regulations governing commerce and the protection of intellectual property.

Licensee is not permitted to convert the Software, and particularly the program code, back into another code form (decompilation / disassembly) or otherwise trace the various stages the Software has undergone during production thereof (reverse engineering), or to make any changes, additions, or modifications to the program except

- as permitted within the scope absolutely granted under copyright law (Sections 69 d and 69 e of the Copyright Act of the Federal Republic of Germany). Licensee is permitted to remedy errors or to change the program in order to adapt it to new provisions of applicable law or regulations or to adjust it for different hardware.
- as this is necessary for the exercise of rights resulting from the licenses of possibly used open source components. Modification of the Software and reverse engineering for debugging such modifications are permitted, insofar required for dynamically linking with LGPL licensed glibc.

3 Installation

This chapter describes the installation procedure consisting of the compile and installation process of the user space library *libcifx* and the kernel module *uio_netx* including the cifX example programs.

The cifX driver can be installed in two ways

- Using the installation script located on the CD, building automatically all required components and installing all required files
- Building and installing all components separately

For the standard use case the automatic installation should be sufficient (see section *Installation of the driver in one step* on page 13). In case of custom needs (e.g. update of only a single component, building the driver for another target system or any installation trouble) the single step installation is the preferred way (see section *Single step installation process* on page 13).

The following steps are required to run a demo application

- Plug in the cifX hardware and start the system
- Extract the driver sources (see section *Preparation* on page 12)
- Install all required driver components (see sections *Installation of the driver in one step* / *Single step installation process* on page 13)
- Load the kernel driver (optional - depends on the chosen installation method, see section *Loading netX UIO driver module* on page 25)
- Build the demo application (see section *Compiling the example programs* on page 23)

In case of any installation trouble please first refer to the chapter *Question and answers* on page 64.

3.1 Prerequisites

- In case building the libcifX library including PCI support (***default!***) the kernel headers are required to build the kernel module

  Kernel headers (version of the kernel, the modules should be build for)
- GCC

For PCI card support

- Library and development package of **libpciaccess** (tested with V0.10.3 / V0.13.1-2)

3.2 Preparation

The following steps explain how to copy the driver source from the CD and extract them in a working directory.

Note: Some files of the driver package provide special functions, e.g. scripts are marked as executable. Not to lose such attributes and permissions, it is required to unpack the driver archive under Linux operating systems. Unpacking the archive under another operating systems (e.g. Microsoft Windows) may clear all attributes. In this case it is not possible to run the scripts without manually restoring of all attributes and permissions.

- Change to your working directory (e.g. /home/project/)

  cd /home/project/

Note: Do not use any whitespaces within your project path since the provided scripts do not handle these.

- Extract/copy the sources from the CD (choose the archive because of the file attributes, see note above)

  tar xf /mnt/cdrom/driver.tar.bz
- Change into the extracted folder

  cd ./driver
- Most of the work, explained in this document will start from this point. If not especially noted, '***project folder***' refers to this folder.

Note: Since several installation instructions rely on the '***project folder***', in the following the document estimates the folder as extracted.

If not especially noted, '***project folder***' refers to the folder of the extracted driver source.

3.3 Installation of the driver in one step

Build process

Note: The next steps require the accomplishment of the preparation noted under section *Preparation* on page 12.

- Change to the driver directory

  cd driver/
- Run the installation script (root rights requested during installation). Optionally a build folder can be passed, otherwise the built binaries reside in ./tmp_build/

  optional step: **mkdir my_build_folder**

  ./build_install_driver [optional pass the build folder]
- Follow the installation instructions
- In case of successful installation the driver is ready to use. For any restrictions see the following note.

Note: In case of a successful installation, note the following restrictions

Running an example program, every accessible device will appear with only the boot loader being flashed. For device specific configuration (e.g. download of device specific firmware) see section *Firmware and configuration file storage* on page 26.

In case of a system reboot the kernel driver needs to be reloaded (for an automated load see section *Loading netX UIO driver module* on page 25).

3.4 Single step installation process

The single step installation process comprises the installation of the following components

- **Kernel Module (optional: required for PCI devices)**

  Build the kernel module netanalyzer.ko and install it (see section *Compiling the netX UIO kernel module* on page 14).
- **User Space library (mandatory: user space driver)**

  Build the libcifx user space library and install it (see section *Compiling the cifX userspace library* on page 18).
- **Boot loader and Firmware (mandatory: required for firmware and configuration)**

  Install the firmware and the boot loader (see section *Firmware and configuration file storage* on page 26 and *Creating the directory tree of the configuration file storage* on page 32).

3.4.1 Compiling the netX UIO kernel module

Building the uio_netx kernel module can be done in two ways

- Building the uio_netx kernel module during the kernel build procedure
- Building the uio_netx kernel module only

The way, which should be chosen, depends on, if the kernel of the target system is already built.

In case, the kernel is already built, there is no need to rebuild the whole kernel. It is possible to build the uio_netx module as an external module and install it afterwards.

3.4.1.1 Compiling the UIO kernel module during kernel build process

The following steps describe how to build the whole kernel including the uio_netx module. This generic kernel build procedure may differ from your kernel build mechanism.

Note: If the kernel is already built, it is not necessary to recompile the whole kernel. In this case, skip this step and continue with section *Compiling the UIO kernel module* on page 16.

- Change to your working directory (e.g. /usr/src)

  cd /usr/src
- Extract the kernel sources

  tar xjf linux-source-x.x.x.tar.bz2
- Change into the uio driver folder within the extracted kernel source

  cd linux-source-x.x.x/drivers/uio
- If necessary, make a copy of the original uio netX source code and then update the uio netx kernel source

  cp uio_netx.c uio_netx.c.[linux-version]

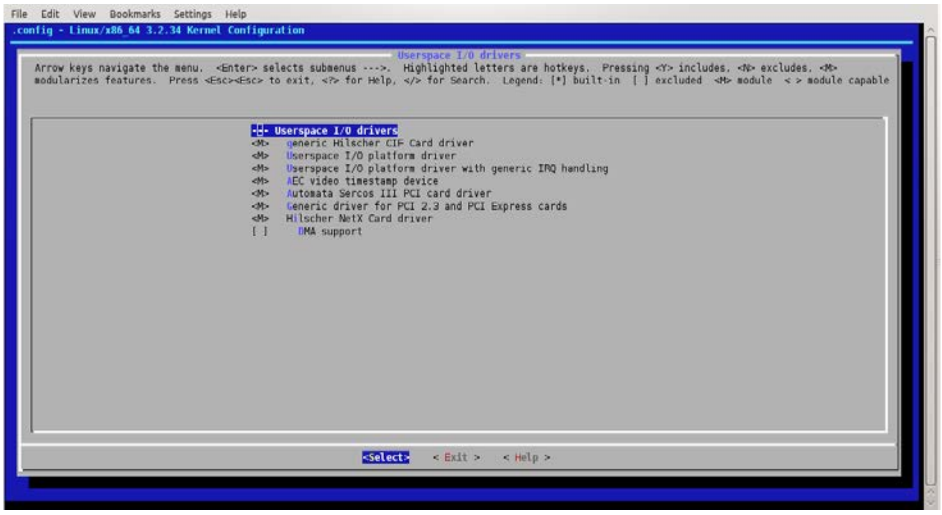
  cp [path to project folder]/driver/uio/uio_netx.c .
- Load your old kernel configuration via command line or inside 'make menuconfig'

  make oldconfig
- Configure your kernel to include UIO ('Userpace I/O drivers') and uio_netx ('Hilscher NetX Card driver')

  make menuconfig

  Enable 'Device Drivers / Userspace I/O Drivers / Hilscher netX Card Driver'

  On demand enable DMA support



```
.config - Linux/x86_64 3.2.34 Kernel Configuration
                         Userspace I/O drivers
Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M>
modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

                    --- Userspace I/O drivers
                    <M>   generic Hilscher CIF Card driver
                    <M>   Userspace I/O platform driver
                    <M>   Userspace I/O platform driver with generic IRQ handling
                    <M>   AEC video timestamp device
                    <M>   Automata Sercos III PCI card driver
                    <M>   Generic driver for PCI 2.3 and PCI Express cards
                    <M>   Hilscher NetX Card driver
                    [ ]     DMA support

                    <Select>    < Exit >    < Help >
```

- Optional: Rebuild the kernel (necessary only if Hilscher netX Card driver should be a built-in driver, not a module)

  make all install

- Build and install the modules

  make modules modules_install

3.4.1.2 Compiling the UIO kernel module only

To build the kernel module *out of tree* for a kernel that is already built, the build and installation process can be done manually or by script. If the target machine is the same machine as the build machine and the module should be built for the current running kernel the automatic installation process is the preferable way because of its easy usage. In contrast, the manual way is more flexible. In case of building the modules for another system choose the manual method.

Any further step depends on the preferred installation method, script-based or manually.

Automatic Installation Process using the Script

Note: The next steps require the accomplishment of the preparation noted under section *Preparation* on page 12.

- Change into the *project driver folder* (see section *Preparation* on page 12)

  (e.g.) cd /home/projects/driver
- Change into 'scripts'

  cd ./scripts
- Build the kernel module (during the build process it is possible to enable or disable DMA support)

  ./install_uio_netx build
- Install the module to the current kernel installation path (see /lib/modules/$(uname -r)/)

  ./install_uio_netx install
- Update the kernel's module dependencies

  ./install_uio_netx update

At this point the module is installed only. Module loading is described in chapter *Loading netX UIO driver module* on page 25.

Manual installation process

Note: The next steps require the accomplishment of the preparation noted under section *Preparation* on page 12.

- Change to your *project driver folder* (see section *Preparation* on page 12)

  (e.g.) cd /home/project/driver
- Change into 'uio_netx'

  cd uio_netx
- Run the makefile

  Note: By default the makefile will generate a module for the active kernel (-> see ***uname -r***) and DMA support enabled.

  To generate a module for a specific kernel set the argument 'KDIR' to the kernel header directory the module should be build for. To disable DMA set the argument 'DMA_DISABLE' to '1'.

  Example: Disabled DMA support and kernel header files located under /home/project/my_kernel/:

  make DMA_DISABLE=1 KDIR=/home/project/my_kernel/
- Copy the uio_netx module in the target directory of the system the module is built for

  cp uio_netx.ko /lib/modules/[kernel-version]/kernel/drivers/uio/

  (Example: cp uio_netx.ko /lib/modules/$(uname -r)/kernel/drivers/uio/)
- Update the list of the module dependencies

  depmod

At this point the module is installed only. Module loading is described in chapter *Loading netX UIO driver module* on page 25.

3.4.2 Compiling the cifX userspace library

The userspace library contains the cifX Toolkit with all necessary Linux adaptations. This library needs to be build for the system the library should run on. The library can be built via the console or the Eclipse IDE.

3.4.2.1 Using the console (CMake)

Installation procedure

Note: The next steps require the accomplishment of the preparation noted under section *Preparation* on page 12.

- Change to your *project driver folder* (see section *Preparation* on page 12)
  (e.g.) cd /home/project/driver
- Create and change into the libcifx build directory
  mkdir libcifxbuild
  cd libcifxbuild
- Configure CMake environment
  cmake [project driver folder] [options]
  e.g.: cmake /home/project/driver/libcifx –DDMA=ON –DTIME=OFF

Option	Parameter	Description
CMAKE_INSTALL_PREFIX	Installation path	Sets the path where the library (subdirectory lib) and include files (subdirectory include/cifx) will be installed. Default: /usr/local
DEBUG	On/off	Enables debug symbols for the generated library
DISABLE_PCI	On/off	Disable PCI support. This will remove all links to libpciaccess. **Note:** When compiling without PCI support, the driver cannot handle cifX PCI cards any more
DMA	On/off	Enables DMA support
HWIF	On/off	Enables support for custom hardware interface (for more information see section *Support for non-PCI device* on page 42). For SPI see also ***SPM_PLUGIN***
NO_MINSLEEP	On/off	Disables minimum sleep time. If “on” the driver may “wait active” (no call to `pthread_yield())`
SPM_PLUGIN	On/off	Enables support for SPI devices under Linux/spidev framework, see section *Using the SPI plugin (Linux spidev framework*) on page 63)
TIME	On/off	Enables toolkit function, setting the device time during device start-up.
VIRTETH	On/off	Enables support for the netX based virtual Ethernet interface **Note:** This feature requires dedicated hardware and firmware (for more information see section *netX-based virtual Ethernet interface* on page 54).
CFLAGS	Compiler flags	Custom compiler flags (e.g. 32-bit on 64-bit platform CFLAGS=-m32)

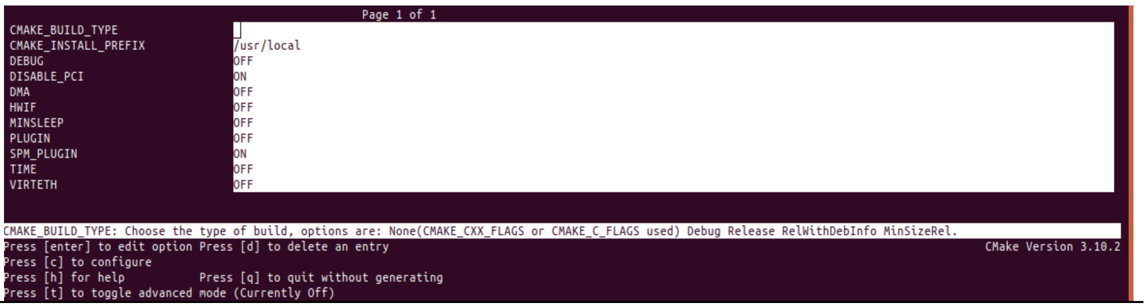
Table 5: Additional libcifx configuration options

- Build all source modules

  make all
- Install the library and include files (root required)

  make install

Note: For more comfortable configuration see ccmake. It's a "*console based graphical version*" of CMake.



```
                                                    Page 1 of 1
CMAKE_BUILD_TYPE                 
CMAKE_INSTALL_PREFIX            /usr/local
DEBUG                           OFF
DISABLE_PCI                     ON
DMA                             OFF
HWIF                            OFF
MINSLEEP                        OFF
PLUGIN                          OFF
SPM_PLUGIN                      ON
TIME                            OFF
VIRTETH                         OFF


CMAKE_BUILD_TYPE: Choose the type of build, options are: None(CMAKE_CXX_FLAGS or CMAKE_C_FLAGS used) Debug Release RelWithDebInfo MinSizeRel.
Press [enter] to edit option Press [d] to delete an entry                                                      CMake Version 3.10.2
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

3.4.2.2 Using the Eclipse IDE

Note: The procedure requires to initialize the build environment via the console. This will create a Makefile which then can be imported by Eclipse.

Get the Eclipse C/C++ environment from https://www.eclipse.org/downloads/. Depending on the download, you will need additionally the CDT-plugins (https://www.eclipse.org/cdt/downloads.php). They may be required to build and debug C/C++ projects. For more information see https://www.eclipse.org/cdt/. There is also information about how to start and develop under the Eclipse environment.

When Eclipse is installed and the workspace path is set, you can load the predefined cifX library project as follows:

- Create the Makefile via the console as described in section *Using the console (CMake)*. (page 18). The call 'make' is not necessary and will be executed by eclipse.
- Start Eclipse.
- Select **File > Import** and choose in the folder **Exisiting Code as Makefile Project**.
- Select the path to the extracted sources ([*'project folder'*]*/libcifx,* see section *Preparation* on page 12) and load the shown pre-selected project.

After importing the project, right click on libcifx, which is shown in the project explorer, to open the extended settings.

- Right click on the project **libcifx**.
- Select **Properties > C/C++ Build > Settings**.
- Under the tab **Tool Settings > Symbols** you can define or undefine special compiler flags.

The default setting is a debug version (g3) without any optimization. The following compiler flags can be set additionally.

Option	Parameter	Description
DEBUG	compiler parameters	Enables debug symbols for the generated library
CIFX_TOOLKIT_ DISABLEPCI	compiler parameters	Disable PCI support. This will remove all links to libpciaccess. **Note:** When compiling without PCI support, the driver cannot handle cifX PCI cards any more
VERBOSE	compiler parameters	Enable verbose outputs to console
CIFX_TOOLKIT_TIME	compiler parameters	Enables toolkit function, setting the device time during device start-up.
CIFX_TOOLKIT_DMA	compiler parameters	Enables DMA support
NO_MIN_SLEEP	compiler parameters	Disables minimum sleep time. If “on” the driver may “wait active” (no call to `pthread_yield())`
CIFXETHERNET	compiler parameters	Enables support for the netX based virtual Ethernet interface **Note:** This feature requires dedicated hardware and firmware (for more information see section *netX-based virtual Ethernet interface* on page 54).
CIFX_DRV_HWIF	compiler parameters	Enables support for custom hardware interface (e.g. SPI) (for more information see section *Support for non-PCI device* on page 42)
CIFX_PLUGIN_SUPPORT	compiler parameters	Enables support for plugins (see *The plugin interface of* the driver on page 48)

Table 6: Additional libcifx configuration options

Build the project

Use either the menu entry **Project->Build All** or right click the libcifx project entry in the 'Project Explorer' view and chose **Build Configurations->Build->All**.

Install the library

Now the library (located under '~/libcifx/Release/' or '~/libcifx/Debug/') needs to be copied to the installation path (/usr/local/lib/). The correct library file is in the format libcifx.so.[Major].[Minor].[Release].

Note: The name of the library depends on the version (e.g. library libcifx.so.1.0.4)

Finally run the next three steps:

- Change into the installation directory (cd /usr/local/lib/).
- Run ldconfig to register library and create a link

  ldconfig
- Create a symbolic link '*libcifx.so*' to the cifx library libcifx.so.[Maj].[Min].[Rel]

  Example for libcifx library V1.0.4

  ln –s libcifx.so.1.0.4 libcifx.so

Note: The required include files (see following list) must also be copied to the installation path (/usr/local/include/cifx/).

cifXEndianess.h	Hil_Logbook.h
cifXErrors.h	Hil_ModuleLoader.h
cifx_io_server.h	Hil_Packet.h
cifxlinux.h	Hil_Results.h
cifx_plc_server.h	Hil_SharedDefines.h
cifXUser.h	Hil_SystemCmd.h
Hil_ApplicationCmd.h	Hil_Taglist.h
Hil_BootParameter.h	Hil_Types.h
Hil_CommandRange.h	io_client.h
Hil_Compiler.h	netXAPI.h
Hil_ComponentID.h	netx_tap.h
Hil_DeviceProductionData.h	rcX_Public.h
Hil_DualPortMemory.h	rcX_User.h
Hil_FileHeaderV3.h	TLR_Types.h
Hil_FirmwareIdent.h	User_Compiler.h
Hil_GenericCommunicationInterface.h	

3.5 Compiling the example programs

All example applications listed in section *CD contents* on page 8, rely on the same two ways to be build.

- **Via Console (CMake / Makefile)**
- **Via IDE (Eclispe)**

The following chapter explains how to build an application using the *cifxsample* test program.

Note: Before using the test applications make sure you have compiled and installed the cifX library which is described in section *Compiling the cifX userspace library* on page 18.

3.5.1 Compiling the cifX example program via console

Installation procedure

Note: The next steps require the accomplishment of the preparation noted under section *Preparation* on page 12.

- Change into the *project examples folder* (see section *Preparation* on page 12)

  (e.g.) cd /home/projects/examples
- Configure the CMake environment

  cmake [options] [build folder]

Option	Parameter	Description
CMAKE_INSTALL_PREFIX	Installation path	Sets the path where the program will be installed. Default: /usr/local
CFLAGS / CXXFLAGS	Compiler flags	Custom compiler flags (e.g. 32bit on 64bit platform CFLAGS=-m32)

Table 7: Additional cifxsample configuration options

- Build all source modules

  make all
- Optional: Install the programs

  make install

3.5.2 Compiling the cifX example program via IDE

Note: The procedure requires to initialize the build environment via the console. This will create a Makefile which then can be imported by Eclipse.

As mentioned before, you must copy the entire *example* folder in your local workspace to open the example project.

- Start Eclipse and import the project as noted in section *Using the Eclipse IDE* on page 20.

  Note: Before compiling the example, the library libcifx must be installed (see section *Compiling the cifX userspace library* on page 18).

The default search path for the header is '/usr/local/include/cifx'. If another path is used, set the include path to the specified one.

- Right click the project **cifxsample**
- Select **Properties > C/C++ Build > Settings**.
- Under the tab **Tool Settings > Directories** you can set a new or additional include path.

Debug information output from the example program can be activated by defining the compiler flag *DEBUG* (set compiler flags, see section *Using the Eclipse IDE* on page 20).

Option	Parameter	Description
DEBUG	compiler parameters	Enables debug information output. (Disabled by default)

Table 8: Additional cifxsample configuration options

Build the project

- Use either the menu entry **Project->Build All** or a right click to the example project entry in the ‘Project Explorer' view and chose **Build Configurations->Build->All**.

The Eclipse debug environment can be used after compiling the project. When the library libcifx is built in debug version, it is also possible to step into the driver functions.

3.6 Loading netX UIO driver module

Module loading / unloading

To load the UIO driver module, you need to run the following command (requires root):

modprobe uio_netx

Note: To automatically load the UIO driver module at system startup, check the manual of your Linux distribution. Usually kernel modules loaded at startup are placed in /etc/modules.

To unload the module run

modprobe –r uio_netx

Optional arguments

The uio_netx driver provides mapping of non-PCI devices by passing the appropriate arguments (custom_dpm_addr, custom_dpm_len and custom_irq). For more information refer to section *Support for non-PCI devices* on page 42. The module arguments are arrays, so it is possible to pass a comma separated list of parameters.

Option	Parameter	Description
custom_dpm_addr	ULONG Array	Physical start address of the DPM (system dependent)
custom_dpm_len	ULONG Array	Length of the DPM (depends on the device)
custom_irq	int Array	Number of the interrupt line (0 = not connected)

Table 9: uio-netx optional arguments

Example parameter usage

The following command loads the kernel module and maps two cards, with a DPM memory location at 0xD0000 (16kB, IRQ5) and 0xFECC0000 (64kB, no IRQ).

modprobe uio_netx custom_dpm_start=0xD0000,0xFECC0000 custom_dpm_len=0x4000,0x10000 custom_irq=5,0

Using netX UIO Driver as user (non-root)

If you want to access the UIO driver with user privileges you will need to make sure the user has read / write access to the following device nodes and files:

- /dev/uio<n>
- /sys/class/uio/uio<n>/device/config

  This can automatically be done by writing an *udev* rule (see example below):

```
/etc/udev/netx.rules
SUBSYSTEMS=="pci",ATTRS{vendor}=="0x15cf",ATTRS{device}=="0x0000",MODE="0666",PROGRAM="/b
in/bash –c ’chmod 0666 /sys/class/uio/uio%n/device/config’"
```

An example of an udev rule (*80-udev-netx.rules*) is located on the CD under /driver/templates/udev/ (see section *CD contents* on page 8). For standard use case (the rule file will match all Hilscher cards) copy the rule file ‘*80-udev-netx.rules*’ to *‘/etc/udev/rules.d/’*. To make the changes take effect restart the udev system via ‘***sudo udevadm trigger***’ or unload and then reload the kernel module *uio_netx*.

3.7 Firmware and configuration file storage

3.7.1 Configuration file storage methods overview

Note: For a cifX device, firmware and configuration files that are not stored on the hardware, must be loaded into the hardware each time the card is powered-up.

cifX cards do not use any Flash memory to store a firmware or configuration in the card. Each time the card is powered-up all files (boot loader, firmware and configuration) must be loaded from the host system into the card. This chapter describes where and how these files have to be stored.

To allow device-specific configuration the card need to be identified and the firmware need to be stored on the host in a specific folder structure to create a unique relation between card and configuration.

These folders reside under a global base folder. By default it is '/opt/cifx' (can be changed during driver initialization).

The driver supports four different types of configuration of a card, each with its specific folder structure:

- slotnumber (depends on the hardware, requires slotnumber switch on the hardware)
- device and serial number
- card name
- single directory

During initialization, for each device, the driver

1. first checks for a folder structure matching the *Slotnumber*,
2. then checks for a folder structure matching the device and serial number,
3. then checks for a folder structure matching the device name
4. finally checks for a configuration in the global single directory.

The first matching variant will provide the configuration and firmware. Features of the respective methods are explained in the following.

(1) *Slotnumber* (requires a slotnumber switch on the hardware)

Advantage: less configuration update required in case of exchange of HW (maintenance)

Disadvantage: limited configuration variants (9 configuration variants)

The Slotnumber serves to distinguish between 9 cifX cards installed in one PC. The Slotnumber must be set at the cifX card using the "slotnumber switch". While Slotnumber 0 means, that the slotnumber identification is ignored and the next identification method is used. Slotnumbers from 1 to 9 corresponds to the Slotnumber 1 to 9. The firmware and configuration file must reside in the subdirectory /Slot_<1..9>/. For detailed information of the folder structure layout see section *Device identification via slotnumber (Slotnumber switch)* on page 28.

(2) device and serial number

Advantage: unlimited number of configuration variants

Disadvantage: exchange of HW requires configuration update

The firmware and configuration file must reside in the subdirectory */<Device Number>/<Serial Number>/*. For detailed information of the folder structure layout see section *Device identification via device and serial number* on page 29.

(3) device name

Advantage: unlimited number of configuration variants

Disadvantage: The name depends on the driver's enumeration. Depending on the connection type (e.g. PCI) the operating system may report the devices in different order. Only recommended if device enumeration order will not change.

The firmware and configuration file must reside in the subdirectory */<Device Name>/*. For detailed information of the folder structure layout see section *Device identification via device name* on page 30.

(4) single directory

Advantage: one configuration fits all, no configuration update necessary in case of HW exchange

Disadvantage: only one configuration possible

If only **ONE** cifX device needs to be supported by the system at the same time, a predefined global directory can be used. The firmware and configuration file must reside in the subdirectory named FW. For detailed information of the folder structure layout, see section *Device identification via single directory* on page 31.

Note: How to setup the basic directory tree of the configuration file storage is described in section *Creating the directory tree of the configuration file storage* on page 32. When creating directories or files remember Linux is case sensitive.

3.7.2 Device identification via slotnumber (Slotnumber switch)

The following table describes the different subdirectory levels, if the device provides a "*Slotnumber switch*" which is used for the Slotnumber identification (typically a rotary switch).

Subdirectory	Description
<BASEDIR>	Base directory **Default: '/opt/cifx'** Can be changed during userspace library initialization. This directory must contain the second stage PCI boot loader (e.g. NETX100-BSL.BIN)
deviceconfig	Device specific configuration files
Slot_<1..9>	If device provides a slotnumber switch, the files will be stored under: **Slot_<slotnumber switch set>**. (Only if the slotnumber switch is not 0) Contains the *device.conf* which holds the device specific settings **Note:** This directory must contain the rcX base firmware if loadable modules are used.
channel<#>	Channel specific files - firmware file (*.nxf - e.g. cifxdpm.nxf) - fieldbus configuration file (*.nxd - e.g. config.nxd) - firmware loadable module file (*.nxo)

Table 10: Firmware and configuration file storage - Slotnumber switch

Sample directory structure for a cifX device identified by *Slotnumber* 2

```
+ <BASEDIR>/
|
|-- NETX100-BSL.BIN (bootloader)
|
|--+ deviceconfig
   |
   |--+ Slot_1
...|
   |--+ Slot_2
   |  |
   |  |-- device.conf (configuration file)
   |  |
   |  |--+ channel0
   |  |  |
   |  |  |-- cifxdpm.nxf
   |  |  |-- config.nxd (fieldbus database or warmstart.dat)
   |  |
   |  |--+ channel1
   |  |--+ channel2
   |  |--+ channel3
   |  |--+ channel4
   |  |--+ channel5
...|
   |--+ Slot_3
   |--+ Slot_4
   |--+ Slot_5
   |--+ Slot_6
   |--+ Slot_7
   |--+ Slot_8
   |--+ Slot_9
```

Note: The base directory structure (including the second stage boot loader) can easily be created using the provided script, see section *Creating the directory tree of the configuration file storage* on page 32.

3.7.3 Device identification via device and serial number

Device identification via the device and serial number are the default way to distinguish between multiple cifX devices in one PC.

Note: *<Device Number>/<Serial Number>* are shown on the device hardware label.
Example:

Hardware Label Entry:	**1250.100 / 20217**
Directory Entry:	**'/1250100/20217'**

The following table describes the different subdirectory levels, without using the slotnumber switch:

Subdirectory	Description
<BASEDIR>	Base directory **Default:** '**/opt/cifx**' Can be changed during userspace library initialization. This directory must contain the second stage PCI boot loader (e.g. NETX100-BSL.BIN)
deviceconfig	Device specific configuration files
<Device Number>	Device number of the device (e.g. 1250100)
<Serial Number>	Serial number of the device (e.g. 20217) Contains *device.conf* storing device specific settings **NOTE:** This directory must contain the rcX base firmware if loadable modules are used.
channel<#>	Channel specific files - firmware file (*.nxf - e.g. cifxdpm.nxf) - fieldbus configuration file (*.nxd - e.g. config.nxd) - firmware loadable module file (*.nxo)

Table 11: Firmware and configuration file storage - Device and serial number

Sample directory structure for a cifX device with device number 1250100 and serial number 20217

```
+ <BASEDIR>/
|
|-- NETX100-BSL.BIN (bootloader)
|
|--+ deviceconfig
   |
   |--+ 1250100
      |
      |--+ 20217
      |  |
      |  |-- device.conf (configuration file)
      |  |
      |  |--+ channel0
      |  |  |
      |  |  |-- cifXdps.nxf (firmware)
      |  |  |-- config.nxd (fieldbus database or warmstart.dat)
      |  |
      |  |--+ channel1
      |  |--+ channel2
      |  |--+ channel3
      |  |--+ channel4
      |  |--+ channel5
      |
```

Note: The base directory structure (including the second stage boot loader) can easily be created using the provided script, see section *Creating the directory tree of the configuration file storage* on page 32.

3.7.4 Device identification via device name

Note: *Device name depends on the driver's enumeration (cifX0, cifX1,…). Depending on the connection (e.g. PCI) the operating system may change the order of reported cards after system restart and so the driver does.* Only recommended if device enumeration order will not change.

The following table describes the different subdirectory levels, using device name:

Subdirectory	Description
<BASEDIR>	Base directory **Default:** '**/opt/cifx**' Can be changed during userspace library initialization. This directory must contain the second stage PCI boot loader (e.g. NETX100-BSL.BIN)
deviceconfig	Device specific configuration files
<device name>	Device name e.g. cifX0 Contains the *device.conf* which holds the device specific settings **Note:** This directory must contain the rcX base firmware if loadable modules are used.
channel<#>	Channel specific files - firmware file (*.nxf - e.g. cifxdpm.nxf) - fieldbus configuration file (*.nxd - e.g. config.nxd) - firmware loadable module file (*.nxo)

Table 12: Firmware and configuration file storage - device name

Sample directory structure for a cifX1 device

```
+ <BASEDIR>/
|
|-- NETX100-BSL.BIN (bootloader)
|
|--+ deviceconfig
   |
   |--+ cifX0
...|
   |--+ cifX1
   |  |
   |  |-- device.conf (configuration file)
   |  |
   |  |--+ channel0
   |  |  |
   |  |  |-- cifxdpm.nxf
   |  |  |-- config.nxd (fieldbus database or warmstart.dat)
   |  |
   |  |--+ channel1
   |  |--+ channel2
   |  |--+ channel3
   |  |--+ channel4
   |  |--+ channel5
...|
   |--+ cifX2
   |--+ cifX3
```

Note: The base directory structure (including the second stage boot loader) can easily be created using the provided script, see section *Creating the directory tree of the configuration file storage* on page 32.

3.7.5 Device identification via single directory

The following table describes the different subdirectory levels, using a s*ingle directory* which has to contain the firmware and configuration files.

Subdirectory				Description
<BASEDIR>				Base directory **Default:** '**/opt/cifx**' Can be changed during userspace library initialization. This directory must contain the second stage PCI boot loader (e.g. NETX100-BSL.BIN).
	deviceconfig			Device specific configuration files
		FW		If s*ingle directory* is used, the search path is set to <**BASEDIR**>/**deviceconfig/FW** Contains the *device.conf* which holds the device specific settings **Note:** This directory must contain the rcX base firmware if loadable modules are used.
			channel<#>	Channel specific files ▪ firmware file (*.nxf - e.g. cifxdpm.nxf) ▪ fieldbus configuration file (*.nxd - e.g. config.nxd) ▪ firmware loadable module file (*.nxo)

Table 13: Firmware and configuration file storage - Single directory

Sample directory structure for *single directory* usage

```
+ <BASEDIR>/
|
|-- NETX100-BSL.BIN (bootloader)
|
|--+ deviceconfig
   |
   |--+ FW
      |
      |-- device.conf (configuration file)
      |
      |--+ channel0
      |  |
      |  |-- cifXdps.nxf (firmware)
      |  |-- config.nxd (fieldbus database or warmstart.dat)
      |
      |--+ channel1
      |--+ channel2
      |--+ channel3
      |--+ channel4
      |--+ channel5
```

Note: The base directory structure (including the second stage boot loader) can easily be created using the provided script, see section *Creating the directory tree of the configuration file storage* on page 32.

3.7.6 Creating the directory tree of the configuration file storage

An easy way to setup the configuration file storage is to use the provided installation script 'install_firmware' located on the CD under **/driver/scripts/**.

The following steps show how to create the directory tree needed by the different configuration file storage methods documented in the sections above starting on page 29.

Note: The next steps require the accomplishment of the preparation noted under section *Preparation* on page 12.

- Change to your *project folder* (see section *Preparation* on page 12)

  (e.g.) cd /usr/src/driver.
- Change into 'script'

  cd ./script
- First install the second stage boot loader by calling (**root** privileges are required)

  ./install_firmware install

  This creates the folder '**/opt/cifx/deviceconfig'** and copies the second stage boot loader to '**/opt/cifx/'**
- Depending on the chosen configuration file storage method, execute one of the following commands (**root** privileges are required)

  <u>Device identification via device and serial number:</u>

   ./install_firmware add_device [device no] [serial no]

  <u>Device identification via slotnumber:</u>

   ./install_firmware add_slot_dir [slot no]

  <u>Device identification via single directory:</u>

   ./install_firmware create_single_dir

Note: This installation procedure only creates the directory structure, installs the boot loader and adds a default configuration file.

To install an application specific firmware refer to section *Configuration file storage methods overview* on page 26.

For further device configuration see section *Device configuration (device.conf)* on page 52.

Remember to adapt the permissions, in case of normal users should be able to access files located in the configuration storage.

---

4 Linux driver-specific information

The Linux driver needs some special initialization comparing to the standard Windows driver, because it is not executed by the kernel at system startup.

The driver (libcifx) is linked to an application and needs to be configured correctly to work.

To enable the use of the cifX driver by an application, some special functions are provided. These functions are described in the following chapters. Further the Linux driver supports also access to devices via different hardware interfaces (e.g. SPI, ISA), for more information refer to *Support for non-PCI device* on page 42.

Chapter *Startup procedure of driver*/library on page 50 describes the correct usage and sequence of the functions.

Features

The user space driver libcifX provides debug output feature. The tracing can be enabled during the driver's initialization (see *Trace Level*, *Structure CIFX_LINUX_INIT* on page 34).

Depending on the trace level the following messages will be logged:

- Trace Level = 0x00 – Tracing disabled
- Trace Level = 0x01 – Debug messages will be logged
- Trace Level = 0x02 – Information messages will be logged
- Trace Level = 0x04 – Warning messages will be logged
- Trace Level = 0x08 – Errors messages will be logged
- Trace Level = 0xFF – All messages will be logged

For debugging purposes it is sometimes useful to enable all debug messages.

By default the driver creates a log file in the driver's ***'base directory'*** (see *Firmware and configuration file storage* on page 26). If the log file creation fails (e.g. no permissions to create or write to a file in the configuration directory) the debug messages will be printed to the console output. Since driver version V2.0.1.0 the log messages can be interpreted from journald or syslogd for example. To map the severities correctly change the severity_mapping array in USER_Linux.c to your needs.

Note: By default root can create and write to a log file only. To be able to log debug messages created by an application started by a normal user, remember to change the permissions of the driver's configuration base directory (see section *Firmware and configuration file storage* on page 26).

Restrictions

By default only root can access a cifX device

Note: libcifx (netX/cifX Toolkit) needs to be run as 'root' or with a user that has the following rights:
=> read/write access to the PCI configuration registers (i.e.'/sys/class/uio/uio<n>/device/config')
=> read/write access to devices '/dev/uio<n>'
=> Mapping of DPM to user space (see 'mmap' and 'ulimit -l')
=> read/write access to /dev/mem (for user added devices)

To be able to access a device as 'normal user' see section *Loading netX UIO driver module* on page 25.

4.1 Additional structures

Some of the Linux specific functions need parameters provided through structures. The structures and the meaning of the internal data are described in the following chapter.

4.1.1 Structure CIFX_LINUX_INIT

This structure is used to initialize the cifX driver.

Element	Data type	Description
init_options	Int	Driver Initialization options: **0 = CIFX_DRIVER_INIT_NOSCAN** Driver does not scan for available cards detected by the UIO driver (driver handles only the user defined cards, see element *user_cards*) **1 = CIFX_DRIVER_INIT_AUTOSCAN** Driver scans for all available cards, which are detected by the UIO driver initializes and adds them to the application. **2 = CIFX_DRIVER_INIT_CARDNUMBER** Driver scans for only one card (UIO device) specified by iCardNumber. Independently of the number the Device name is set to 'cifX0'.
iCardNumber	Int	Index of card to initialize when init_option is set to **CIFX_DRIVER_INIT_CARDNUMBER**
fEnableCardLocking	Int	Ensures exclusive access to one specific cifX card for multiple applications. **NOTE: It does not synchronize concurrent access between multiple applications. Synchronization for multiple applications need to be implemented by the user (see Limitations on page 7).** **fEnableCardLocking = 0** User application has to guarantee not to grant any other applications access. **fEnableCardLocking <>0** Cards which are already accessed by other applications will be ignored and therefore not enumerated. (Useful option in mode **CIFX_DRIVER_INIT_CARDNUMBER)**
base_dir	const char*	Set the base directory of the driver, Set to NULL to use the default directory (**'/opt/cifx'**)
poll_interval	unsigned long	Polling interval in milliseconds [ms] for non-interrupt cards. Used for *Change of State* (COS) detection **0 = default of 500ms** **CIFX_POLLINTERVAL_DISABLETHREAD** can be used to completely disable COS polling
poll_priority	Int	Priority of the polling thread (for possible values see man page of pthread_setschedparam) **0 = default (priority of the calling thread)**
poll_schedpolicy	Int	Scheduling policy, need to be set when poll_priority is set 0 = SCHED_NORMAL (poll_priority 0) 1 = SCHED_FIFO (poll_priority 1..99) 2 = SCHED_RR (poll_priority 1..99)
poll_StackSize	Int	Stack size of the polling thread. *poll_StackSize* specifies the number additional bytes to add to PTHREAD_STACK_MIN (= 0x4000Bytes). If *poll_StackSize* is set to 0 the default size +0x1000 byte is used. **Default Stack-Size: PTHREAD_STACK_MIN + 0x1000**
trace_level	unsigned long	Set the trace level of the driver. **0x0000 = no trace information is created** **0xFFFF = maximum trace information is created**

Element	Data type	Description
user_card_cnt	int	Number of user cards to be manually added to the driver. Devices are specified by the CIFX_DEVICE_T structure.
user_cards	struct CIFX_DEVICE_T*	Pointer to an optional array of additional user card structures. Number of card structures in the array must be given in ***user_card_cnt***. For more information see section *Structure CIFX_DEVICE_T* on page 35.
logfd	FILE*	Pointer to an open file used as driver log file. Can be used as well to log to journal or syslogd for example.

Table 14: Structure CIFX_LINUX_INIT definition

4.1.2 Structure CIFX_DEVICE_T

This structure contains all information describing a cifX device. The structure needs to be filled in the following cases:

- **Handling non-UIO devices**

  In case of a netX device, which is not detectable by the UIO driver, should be added to the driver's control (for more information see section *Support for non-PCI devices* on page 42).
- **Controlling more than one UIO device, but not all that exist in the system**

  In this case neither the *CIFX_DRIVER_INIT_CARDNUMBER* nor the *CIFX_DRIVER_INIT_AUTOSCAN* option can be used. Instead an array of the required cards needs to be passed to the driver.

Thereby the requested cards, so called '*User Cards*', are differentiated by the following two groups

- ***UIO-Devices***

  Detected by the UIO driver (cifX PCI cards)
- ***Non-UIO devices***

  Not detectable by the UIO driver

In case of a ***UIO-Device*** the information for the CIFX_DEVICE_T structure can be easily retrieved by calling *cifXFindDevice(). cifXFindDevice()* fills the CIFX_DEVICE_T structure for the requested device and returns. In case of a ***non-UIO device*** the structure needs to be filled by the user and passed to the driver. In this case UIO-specific fields need to be invalidated by setting the values to '-1'.

CIFX_DEVICE_T data content

Element	Data type	Description	
		UIO device	None UIO device
dpm	unsigned char*	**Virtual pointer to card DPM**	
		Filled by *cifXFindDevice()*	Must be provided by the user (e.g. via mmap()).For more information refer to Support for non-PCI devices on page 42
dpmaddr	unsigned long	**Virtual pointer to card DPM**	
		Filled by *cifXFindDevice()*	Must be provided by the user. For more information refer to Support for non-PCI devices on page 42
dpmlen	unsigned long	**Size of the DPM in bytes**	

Element	Data type	Description	
		UIO device	None UIO device
		Filled by *cifXFindDevice()*	Must be provided by the user. For more information refer to Support for non-PCI devices on page 42
uio_num	int	**UIO number of the device**	
		Filled by *cifXFindDevice()*	Not used set to '-1'
uio_fd	int	**File handle to UIO device**	
		Filled by *cifXFindDevice()*	Not used set to '-1'
pci_card	int	**PCI card handling** 0 = Card is a non-PCI card with firmware in FLASH memory (no reset during start-up required) 1 = Card is a PCI card, needs to be reset on every start	
		- Filled by *cifXFindDevice()* *- Can be overwritten by user*	Not used set to '-1'
force_ram	int	**Force card storage behavior** 0 = Auto-detect card storage (PCI = RAM, DPM = Flash) 1 = Force usage of RAM only on this card. (This will execute a HW reset and download boot loader / Firmware on every start of the card)	
		- Filled by *cifXFindDevice()* *- Can be overwritten by user*	Must be provided by the user
notify	PFN_CIFX_NOTIFY_EVENT	**Optional user initialization function** Callback that is made at several stages when initializing a device. This allows the user to setup DPM and timings if they are different from the netX ROM Loader settings. **NULL = suppress callback**	
		User provided	User provided
userparam	void*	**User definable information per device**	
		User provided	User provided
Optional (requires the compiler flag CIFX_DRV_HWIF set, when compiling the user space library libcifx, see section Compiling the cifX userspace library on page 18**)**			
hwif_init		**Optional: User definable function.** Initializes the hardware interface (may be NULL). For more information refer to *Hardware initialization via hwif_init* on page 45.	
		Not used set to NULL	User provided
hwif_deinit		**Optional: User definable function.** De-initializes the hardware interface (may be NULL). For more information refer to *Hardware de-initialization via hwif_deinit* on page 45.	
		Not used set to NULL	User provided
hwif_read		**User definable function. Implements the read access to the netX device.** Function implementation highly depends on the hardware interface. For more information refer to *Hardware read access via hwif_read* on page 46.	
		Not used set to NULL	User provided
hwif_write		**User definable function. Implements the write access to the netX device.** Function implementation highly depends on the hardware interface. For more information, see *Hardware write access via hwif_write* (page 47).	
		Not used set to NULL	User provided

Table 15: CIFX_DEVICE_T data content

4.2 Additional functions

This chapter describes functions which are available for the Linux version of the driver only. These functions need to be used to initialize the driver to be usable inside an application.

Specific functions of the Linux driver:

Function	**Description**
cifXDriverInit()	Driver initialization function, see *cifXDriverInit()*) on page 38.
cifXDriverDeinit()	De-initialization of the driver, see *cifXDriverDeinit()* on page 39,
xDriverRestartDevice()	Restarts the specified device, see *xDriverRestartDevice()* on page 39,
cifXGetDriverVersion()	Returns the driver and toolkit version, see *cifXGetDriverVersion()* on page 40,
cifXGetDeviceCount()	Returns the number of the detected UIO devices, see *cifXGetDeviceCount()* on page 40,
cifXFindDevice()	Returns the information structure (CIFX_DEVICE_T) of the requested UIO device, see *cifXFindDevice()* on page 41.
cifXDeleteDevice()	Deletes a previously via cifXFindDevice() acquired device. see *cifXDeleteDevice()* on page 41.

Table 16: Linux cifX Driver: Specific functions of the Linux driver

4.2.1 cifXDriverInit()

This function must be called before accessing any driver function. It initializes the driver and adds the needed devices to the control of the libcifx shared library.

Function call

```
int32_t cifXDriverInit(struct CIFX_LINUX_INIT* init_params)
```

Arguments

Argument	Data type	Description
init_params	struct CIFX_LINUX_INIT_T*	Initialization parameters (see section *Structure CIFX_LINUX_INIT* on page 34 for details)

Return Values

CIFX_NO_ERROR (0) if the driver was successfully initialized.

Remarks

The driver initialization provides three different types, see element '***init_options***' in *Structure CIFX_LINUX_INIT* on page 34.

Note: The given initialization option belongs only to UIO devices. In general user defined Non-UIO devices (see *Structure CIFX_DEVICE_T* on page 35) given in '*user_cards*' are not effected and will be always added to the driver's control.

- **CIFX_DRIVER_INIT_NOSCAN**

  The driver ignores all devices which are detected by the UIO driver.

  The driver handles only the given ***User Cards*** (see element '***user_cards***' in section *Structure CIFX_LINUX_INIT* on page 34).

  Use case: The application should not acquire every device found, instead specified ones only.
- **CIFX_DRIVER_INIT_AUTOSCAN**

  The driver scans for all devices, which are detected by the UIO driver and adds them to the driver's control.

  Use case: The application should have access to all cards, found in the PC.
- **CIFX_DRIVER_INIT_CARDNUMBER**

  The driver scans for the requested device (UIO device) and adds it to the driver's control.

  Use case: The application should have access to only one specific card (UIO device).

4.2.2 cifXDriverDeinit()

Un-initialize the driver and remove all devices from the control of the *libcifx* shared library. After calling this function the application must not access any cifX Driver API function any more.

Function call

```
void cifXDriverDeinit(void)
```

Arguments

None

4.2.3 xDriverRestartDevice()

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

A restart is necessary on PCI-based-netX boards, if a running firmware should be updated or changed. Because on such boards the firmware is not stored in a FLASH file system and updating the firmware while it is running in RAM is not possible.

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

Function call

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

Arguments

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

Return Values

CIFX_NO_ERROR if the function succeeds.

4.2.4 cifXGetDriverVersion()

This function returns the version of the cifX driver for Linux.

Function call

```
int32_t cifXGetDriverVersion ( uint32_t ulSize, char* szVersion);
```

Arguments

Argument	Data type	Description
ulSize	unsigned long	Size of buffer referenced by parameter *szVersion*
szVersion	char*	Buffer to return driver version string

Return values

Return values	
CIFX_NO_ERROR	Memory mapping successful
CIFX_INVALID_BUFFERSIZE	Size of supplied buffer is too small

4.2.5 cifXGetDeviceCount()

Query the number of available UIO devices. Device detection only works through the netX UIO driver.

Function call

```
int cifXGetDeviceCount(void)
```

Arguments

None

Return values

Number of detected devices.

4.2.6 cifXFindDevice()

Build a CIFX_DEVICE_T structure for a given device.

The structure can be used by an application if only some specific cards should be used. Therefore the application has to add them manually to the driver as a user card, see section *Structure CIFX_LINUX_INIT* on page 34.

This can be done by calling the function *cifXDriverInit()* with the '***CIFX_DRIVER_INIT_NOSCAN***' option and passing the card information in the *user_cards* parameter.

Function call

```
struct CIFX_DEVICE_T* cifXFindDevice(int num, int fCheckAccess)
```

Arguments

Argument	Data type	Description
num	int	Device number of the chosen device. Range: 0..cifXGetDeviceCount()
int	fCheckAccess	Check if device is already used by another application fCheckAccess = 0, do not check if already accessed fCheckAccess = 1, check if device is already accessed

Return values

Pointer to the device information structure of the given device.

NULL, if the device number is invalid or not available or if *fCheckAccess* = 1 and the device is already used by another application.

4.2.7 cifXDeleteDevice()

Delete a CIFX_DEVICE_T structure that was returned by *cifXFindDevice()*. This needs to be done **after** the driver un-initialization to clean up all internally used administration data and allocated memory areas.

Function call

```
void cifXDeleteDevice(stuct CIFX_DEVICE_T* device)
```

Arguments

Argument	Data type	Description
device	struct CIFX_DEVICE_T*	Pointer to a device returned by *cifXFindDevice()*

4.3 Support for non-PCI devices

The *Linux cifX Driver* provides the ability to access devices connected via several hardware interfaces. In general, the supported devices can be grouped into so called memory-mapped devices and non-memory-mapped devices. Since the driver is capable to detect PCI devices autonomously only, other devices need to be published by the customer.

Note: The driver provides a plugin for netX devices that are connected via SPI under Linux (spidev framework). For more information see *Using the SPI plugin (Linux* spidev framework) on page 63.

Depending on the type of the device (memory-mapped or non-memory-mapped) the driver provides the following integration possibilities:

Device type	Integration interface	Features / limitations / description
Memory-mapped DPM	Kernel Mode Driver uio-netx	Features ▪ No difference between custom and standard uio device ▪ Interrupt support ▪ No additional initialization in user space (skips adding user defined card) Limitations ▪ Parameter need to be passed during driver startup Memory mapped devices can easily passed to the driver without any driver preparation. For more information see section *ISA or other memory-mapped devices* on page 43.
	User Space Driver libcifx	Features ▪ Independent of the uio_netx kernel module Limitations ▪ No interrupt support Memory mapped devices can easily passed to the driver without any driver preparation. For more information see section *ISA or other memory-mapped devices* on page 43.
Non-Memory Mapped DPM	User Space Driver libcifx	Features ▪ Depends on the customer's implementation Limitations ▪ Depends on the customer's implementation This method requires the implementation of dedicated hardware read/write functions. For more information see section *Custom-specific hardware interface* on page 44. **Note: To be able to handle a non memory mapped device the driver need to be build with the compiler flag CIFX_DRV_HWIF set (see section *Compiling the cifX userspace library* on page 18).**

Table 17: Overview supported device types

4.3.1 ISA or other memory-mapped devices (DPM)

The netx-uio kernel mode driver is capable to detect PCI devices autonomously only. Other memory-mapped devices like for example ISA devices do not provide any detection methods and need to be published by the customer. The device can easily be integrated by passing the device-specific-memory parameter to the driver. The parameter can be passed to the uio-netx kernel module or the *User Space Driver* libcifx. For the differences of both methods see table *Overview supported device types* on page 42. On ARM platforms the device tree can be used. For more information how to parametrize the device tree see Using the Device Tree on page 61.

- ***Memory-mapped device via kernel module uio_netx***

  The device-specific information can be passed via command line parameter during module loading:

  modprobe custom_dpm_addr=0xD0000 custom_dpm_len=0x4000 custom_irq=4

  The above example adds a device with the DPM located at the physical address 0xD0000, DPM length of 16 KB and interrupt connected to IRQ line 4. For more information of the parameter see section *Loading netX UIO driver module* on page 25.

  In case the mapping succeeds, the driver creates a new uio_netx device which is accessible via the user space library libcifx **as common uio device**.

  For an example refer to Using UIO driver on page 60.

- ***Memory-mapped device via user space library libcifx***

  Integration of a memory mapped device via a user space library requires the device specification via *Structure CIFX_DEVICE_T* (page 35). The filled structure need to be passed to the drivers initialization routine *cifXDriverInit()* via *Structure CIFX_LINUX_INIT* (page 34).

  The following table shows the important parameter of the CIFX_DEVICE_T structure. For other parameter or general information refer to *Structure CIFX_DEVICE_T* on page 35.

Name	Type	Description
Dpm	unsigned char*	Virtual Pointer to the card’s DPM. The driver provides a helper function (*cifx_ISA_map_dpm()*), mapping the physical address to the application’s specific virtual memory. For more information refer to the Example: Driver initialization for ISA device on page 59.
dpmaddr	unsigned long	Physical address to the card’s DPM (this parameter depends on the system and the hardware configuration, for more information refer to the appropriate hardware documentation).
Dpmlen	unsigned long	Size of the DPM in bytes (depends on the device, for more information refer to the appropriate hardware documentation).

Table 18: Initialization parameter: Custom memory mapped device

For an example refer to *Not using UIO driver* on page 59.

4.3.2 Custom-specific hardware interface

Note: To enable the following feature, the user space library needs to be built with the compiler flag CIFX_DRV_HWIF (set), see section *Compiling the cifX userspace library* on page 18.

Additional to *Memory-mapped devices*, the cifX Toolkit is capable to access netX-based hardware via the *cifXToolkit Hardware Function Interface.* For more information refer to **Fehler! Verweisquelle konnte nicht gefunden werden.**.

Similar to a '*non-UIO-memory-mapped Device*', the custom device needs to be specified via the *Structure CIFX_DEVICE_T* (on page 35). Though, the custom hardware interface feature requires an additional implementation of interface-specific read/write functions.

The hardware-specific read/write functions need to be specified per device, during driver initialization (see *hwif_read, hwif_write*, *Structure CIFX_DEVICE_T*).

The following table shows the important parameter of the CIFX_DEVICE_T structure. For other parameter or general information refer to *Structure CIFX_DEVICE_T* on page *Structure CIFX_DEVICE_T* on page 35.

Name	**Type**	**Description**
dpm	unsigned char*	set to 0 (since no physical address exists)
dpmaddr	unsigned long	set to 0 (since the driver cannot access the device's DPM directly)
dpmlen	unsigned long	Size of DPM in Bytes (depends on the device, refer to the hardware documentation)
userparam	void*	Optional: User parameter (may point to information required for the hardware interface). If not used set to NULL
hwif_init	PFN_DRV_HWIF_INIT	Optional: Initializes the custom hardware interface **Note: Need to be implemented by customer** (see section *Hardware initialization via* hwif_init on page 45). If not used set to NULL
hwif_deinit	PFN_DRV_HWIF_DEINIT	Optional: De-initializes the custom hardware interface **Note: Need to be implemented by customer** (see section *Hardware de-initialization via hwif_deinit* on page 45). If not used set to NULL
hwif_read	PFN_DRV_HWIF_MEMCPY	Reads a given number of bytes from the netX DPM via the custom hardware interface. **Note: Need to be implemented by customer** (see section *Hardware read access via hwif_read* on page 46).
hwif_write	PFN_DRV_HWIF_MEMCPY	Writes a given number of bytes to the netX DPM via the custom hardware interface. **Note: Need to be implemented by customer** (see section *Hardware write access via hwif_write* on page 47).

Table 19: Initialization parameter: Custom hardware interface

In case of registered hardware functions (*hwif_read*, *hwif_write*), the toolkit replaces the common memory access (e.g. via memcpy()) by the appropriate access function.

The initialized structure need to be passed to the drivers initialization routine *cifXDriverInit()* via *Structure CIFX_LINUX_INIT*. For an SPI example application see SPISample, *CD contents* on page 8).

4.3.2.1 Hardware initialization via hwif_init

This function needs to be implemented by the customer. The function needs to provide a complete initialization of the specific hardware interface. If the function returns success (*CIFX_NO_ERROR*) it must be guaranteed that is is possible to read from and write to the interface. Further, the function implementation must be aware of its initialization state, since it may be called during application runtime (e.g. in case of a reset via *xDriverRestartDevice()*). The passed device-specific structure *Structure CIFX_DEVICE_T*, provides a tag called *userparam*, which enables passing of interface-specific information and states.

This function is optional. In case initialization is not required set the *hwif_init* in *Structure CIFX_DEVICE_T* to NULL.

Function call

```
int32_t hwif_init (stuct CIFX_DEVICE_T* device)
```

Arguments

Argument	Data type	Description
device	struct CIFX_DEVICE_T*	Pointer to the device

Return values

CIFX_NO_ERROR on success

For all possible error values refer to the header file *cifXErrors.h* located in the *cifX Driver Toolkit*, see Toolkit, *CD contents* on page 8.

4.3.2.2 Hardware de-initialization via hwif_deinit

This function needs to be implemented by the customer. The function needs to provide a complete de-initialization of the specific-hardware interface. Further, the function implementation must be aware of its initialization state, since it may be called during application runtime (e.g. in case of a reset *xDriverRestartDevice()*). The passed device specific structure *Structure CIFX_DEVICE_T* provides a tag called *userparam*, which enables passing of interface-specific information and states.

This function is optional. In case de-initialization is not required, set the *hwif_deinit* in the Structure CIFX_DEVICE_T to NULL.

Function call

```
void hwif_deinit (stuct CIFX_DEVICE_T* device)
```

Arguments

Argument	Data type	Description
device	struct CIFX_DEVICE_T*	Pointer to the device

4.3.2.3 Hardware read access via hwif_read

This functions need to be implemented by the customer. The function needs to provide reading the number of bytes given by ulLen from the DPM. pvAddr contains the offset where to start reading from the DPM.

The passed device specific structure *Structure CIFX_DEVICE_T* provides a tag called *userparam*, which enables passing of interface-specific information and states.

Function call

```
void* hwif_read ( stuct CIFX_DEVICE_T* device,
                  void* pvAddr,
                  void* pvData,
                  uint32_t ulLen)
```

Arguments

Argument	Data type	Description
device	struct CIFX_DEVICE_T*	Pointer to the device
pvAddr	void*	Offset in DPM where to read from **Note: This is a pointer to the DPM location where to read from. This can be handled as an offset (unsigned long) from the beginning of the DPM, if the parameter *dpmaddr* is set to NULL (see *Structure CIFX_DEVICE_T* on page 35). Otherwise *dpmaddr* need to subtracted to get the offset.**
pvData	void*	Pointer to memory where to store read data
ulLen	uint32_t	Length of data to read

Return values

Pointer to the read buffer passed into the function call (pvData).

4.3.2.4 Hardware write access via hwif_write

This function needs to be implemented by the customer and has to write the number of bytes, given by ulLen to the DPM start offset contained in pvAddr.

The passed device specific structure *Structure CIFX_DEVICE_T* provides a tag called *userparam*, which enables passing of interface specific information and states.

Function call

```
void* hwif_write ( stuct CIFX_DEVICE_T* device,
                   void* pvAddr,
                   void* pvData,
                   uint32_t ulLen)
```

Arguments

Argument	Data type	Description
device	struct CIFX_DEVICE_T*	Pointer to the device
pvAddr	void*	Offset in DPM where to write to **Note: This is a pointer to the DPM location where to write to and it can be handled as an offset (unsigned long) from the beginning of the DPM, if the parameter *dpmaddr* is set to NULL (see *Structure CIFX_DEVICE_T* on page 35). Otherwise *dpmaddr* need to subtracted to get the offset.**
pvData	void*	Pointer to a buffer containing the write data
ulLen	uint32_t	Length of data to write

Return values

Pointer holding the write address passed into the function (pvAddr).

4.3.3 The plugin interface of the driver

The plugin interface of the driver enables easy integration of different hardware interfaces. It is build on the technique of the hardware access functions of the toolkit, see *Custom-specific hardware interface* on page 44. Since the configuration becomes abstracted, the result is an easy and generic way to handle various types of hardware and their interfaces. The driver is able to automatically load the plugin and enumerate transparently the devices via the cifX API. The default search path for plugins is *(/opt/cifx/plugins/)*. A plugin is a dynamically loadable library and has to provide the functions listed in the following table.

The driver is delivered with an example plugin for the spidev interface of the Linux Kernel, see SPM_PLUGIN in the drivers build options (*Compiling the cifX userspace library* on page 18). For configuration see *Using the SPI plugin (Linux* spidev framework) on page 63. In case of incompatible frameworks see *Custom-specific hardware interface*.

Overview of the functions required to implement a plugin.

Function	**Description**
cifx_device_count()	Returns the total number devices available by this plugin
cifx_alloc_device()	Returns a pointer to the filled device information (see *Structure CIFX_DEVICE_T*)
cifx_free_device()	Frees the resources of the device allocated by cifx_alloc_device()

4.3.3.1 cifx_device_count()

The function returns the total number of devices which are available by this plugin.

Function call

```
uint32_t cifx_device_count(void)
```

Arguments

Argument	**Data type**	**Description**
- / -	- / -	- / -

Return value

Returns to total number of devices available by this plugin.

4.3.3.2 cifx_alloc_device()

The functions returns the information structure CIFX_DEVICE_T of the device identified by the parameter ***num***. The structure contains all required information how to access the device, see *Custom-specific hardware interface* on page 44.

Function call

```
struct CIFX_DEVICE_T* cifx_alloc_device (uint32_t num)
```

Arguments

Argument	Data type	Description
num	uint32_t	Number of the device which resources should be returned in the device information structure. The returned pointer should be valid for all num while ((return value of cifx_device_count() – 1) – num) >= 0

Return value

Pointer to the device information structure of the device indexd **[*num*]**.

NULL, if the device number is invalid or the allocation failed.

4.3.3.3 cifx_free_device()

The function frees all the device resources and the buffer that has been previously allocated by cifx_alloc_device().

Function call

```
void cifx_free_device( struct CIFX_DEVICE_T* ptDev)
```

Arguments

Argument	Data type	Description
ptDev	struct CIFX_DEVICE_T*	Pointer to Device structure previously allocated by cifx_alloc_device()

Return value

No

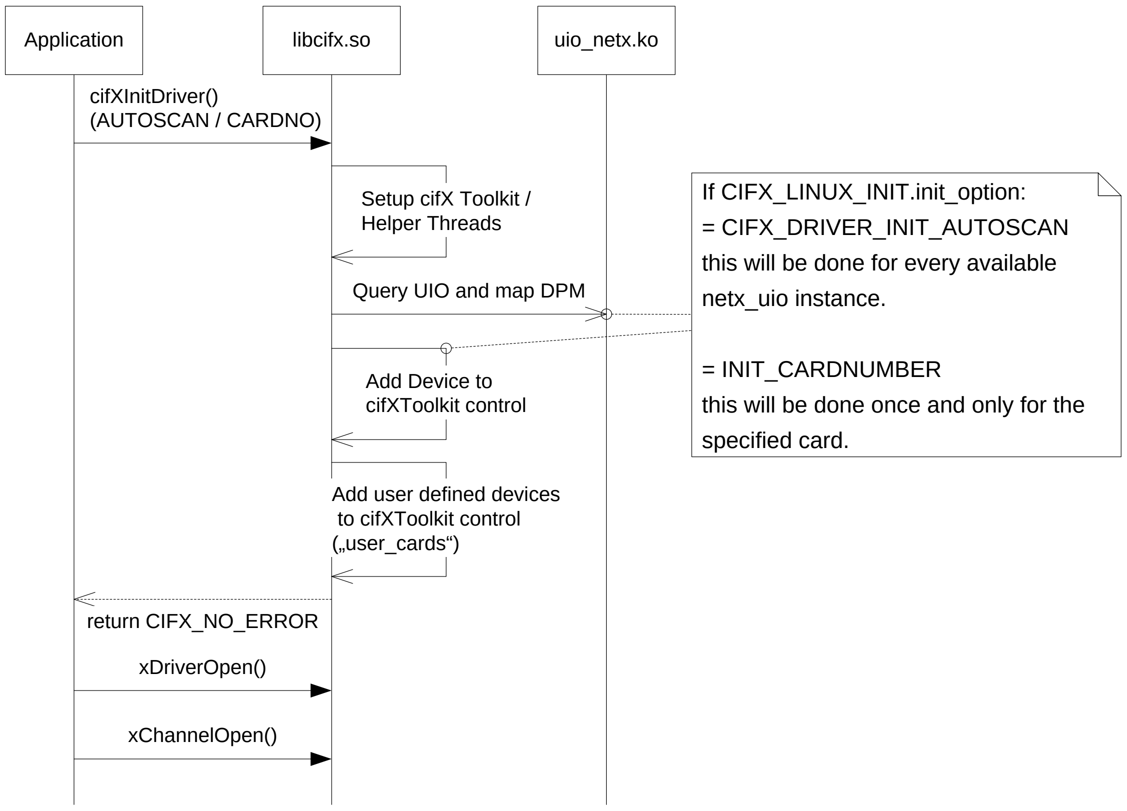
4.4 Startup procedure of driver/library

The driver start-up procedure can be controlled by the user, setting the appropriate initialization flag (*Structure CIFX_LINUX_INIT (init_options)* on page 34).

The following three use cases are available:

- **CIFX_DRIVER_INIT_AUTOSCAN**
  Automatically add all found uio_netx-based devices and add user specified devices.
- **CIFX_DRIVER_INIT_CARDNUMBER**
  Add only one specific uio_netx-based device and add user specified devices.
- **CIFX_DRIVER_INIT_NOSCAN**
  Skip uio_netx device scan and add only user specified devices.

4.4.1 Startup via AUTOSCAN or CARD number



If CIFX_LINUX_INIT.init_option:
= CIFX_DRIVER_INIT_AUTOSCAN
this will be done for every available netx_uio instance.

= INIT_CARDNUMBER
this will be done once and only for the specified card.

Figure 2: Initialization of libcifx using CIFX_DRIVER_INIT_AUTOSCAN / CIFX_DRIVER_INIT_CARDNUMBER

4.4.2 Startup via CIFX_DRIVER_INIT_NOSCAN

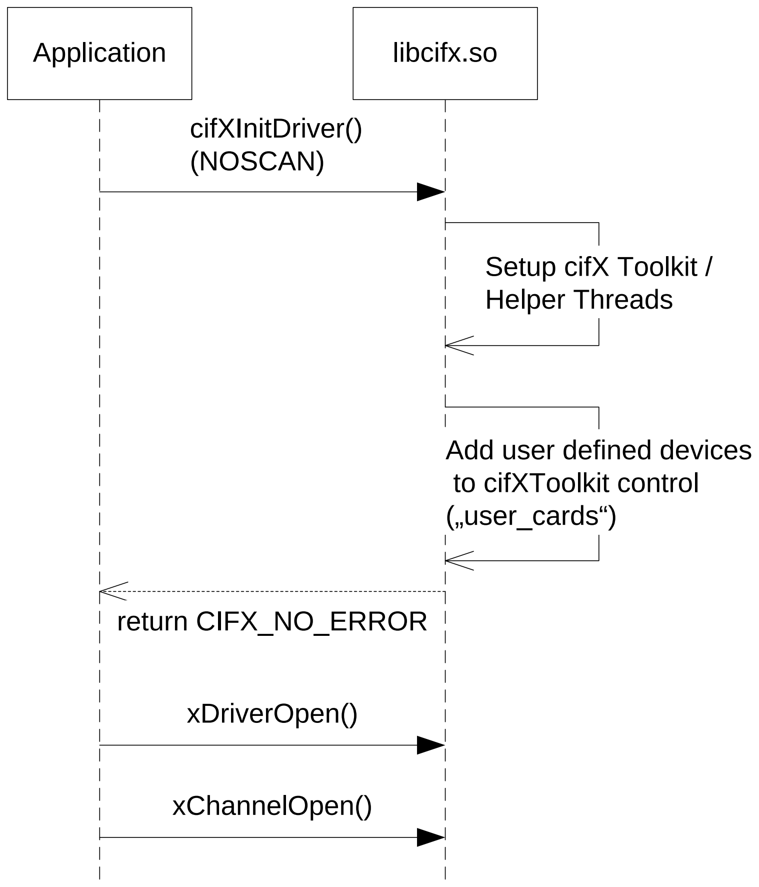


Figure 3: Initialization of libcifx using CIFX_DRIVER_INIT_NOSCAN

4.5 Device configuration (device.conf)

Parameters like a unique alias name and interrupt support can be configured per device. The configuration file must be named *'device.conf'.* Where to place the configuration file, depends on the chosen configuration file storage method.

Device identification via slotnumber (page 28)

- '*/opt/cifx/deviceconfig/Slot_[no]/device.conf*'
  e.g. '/opt/cifx/deviceconfig/Slot_1/*device.conf* '

Device identification via device and serial number (page 29)

- *'/opt/cifx/deviceconfig/[device no]/[serial no]/device.conf'*
  e.g. '/opt/cifx/deviceconfig/1250100/20217/*device.conf*'

Device identification via device name (page 30)

- *'/opt/cifx/deviceconfig/[device name]/device.conf'*
  e.g. '/opt/cifx/deviceconfig/cifX0/*device.conf*'

Device identification via single directory (page 31)

- '*/opt/cifx/deviceconfig/FW/device.conf*'

The file may contain the following keys:

Key	Datatype	Description
Alias	char[16]	Alias name for the device. Must be less than 16 characters
Irq	String	Enable/Disable IRQ on the device **'no' = IRQ disabled** **'yes' = IRQ enabled**
Irqprio	Int	Priority of the ISR handler thread (0 = default (priority of the calling thread) see Linux man pages pthread_attr_setschedparam
irqsched	String	Setup alternate ISR scheduling algorithm See Linux man pages pthread_attr_setschedpolicy **'fifo' = FIFO scheduling (see SCHED_FIFO -> irqprio 1..99)** **'rr' = Real-Time Scheduling (see SCHED_RR -> irqprio 1..99)**
Dma	String	Enable/Disable DMA support of the device **'no' = DMA disabled** **'yes' = DMA enabled** **Note:** DMA support needs also to be enabled in the *uio_netx* kernel module, for more information see sections *Compiling the UIO kernel module during kernel build* (page 14) and Compiling the UIO kernel module (page 16).
Eth	String	Enable/Disable Virtual Ethernet Interface support of the device **'no' = Ethernet Interface disabled** **'yes' = Ethernet Interface enabled** **Note:** This feature requires a firmware running on the PC card cifX that provides an extra channel supporting a dedicated stack to transport Raw-Ethernet data (for more information see section *netX-based virtual Ethernet interface* on page 54)

Table 20: device.conf parameters

Sample device.conf

```
#Sample device configuration file
alias=PROFIBUS
irq=no
irqprio=1
irqsched=fifo
dma=no
```

4.6 netX-based virtual Ethernet interface

Note: This feature requires a firmware running on the PC card cifX that provides an extra channel supporting a dedicated stack to transport Raw-Ethernet data.

The libcifx user space library provides an extension to create and serve a virtual Ethernet device for common network application usage.

The virtual network adapter is based on the TUN/TAP driver.

4.6.1 Features

- Polling Mode
- Simultaneous access of the PC card cifX from cifX driver and the corresponding Ethernet device

4.6.2 Requirements

- cifX Device Driver V1.0.3.0 or later
- Firmware with appropriate Ethernet packet API as specified in section “Ethernet Protocol API in Ethernet (NDIS) Mode“ in reference [3].

4.6.3 Limitations

- **Performance:**

  Max. TCP/IP throughput (send/receive): 42-49 MBit/s / 11-17 MBit/s.

  Note: The throughput highly depends on the running firmware and the fieldbus configuration.
- **Network packets:**

  Network packet type indication is not configurable. Since the libcifx driver does no packet filtering (Multicast, Broadcast, …) the types of delivered packets depends on the firmware. For detailed information about the set of provided network packets refer to the documentation of the firmware which will be installed.
- **MAC Address:**

  The device MAC address is not configurable and therefore bind to a fixed MAC address. For more information refer to documentation of the firmware which will be installed. The fieldbus stack running on the netX will hold its own MAC address.
- The application/user must have CAP_NET_ADMIN privileges
- The Ethernet device lifetime is bind to applications lifetime, which initializes the driver
- Ethernet device will disappear if a device reset is executed
- Application must not access the communication channel used for raw Ethernet access

4.6.4 Overview

The following figure shows an example application and all the required components and how they are layered and interact.

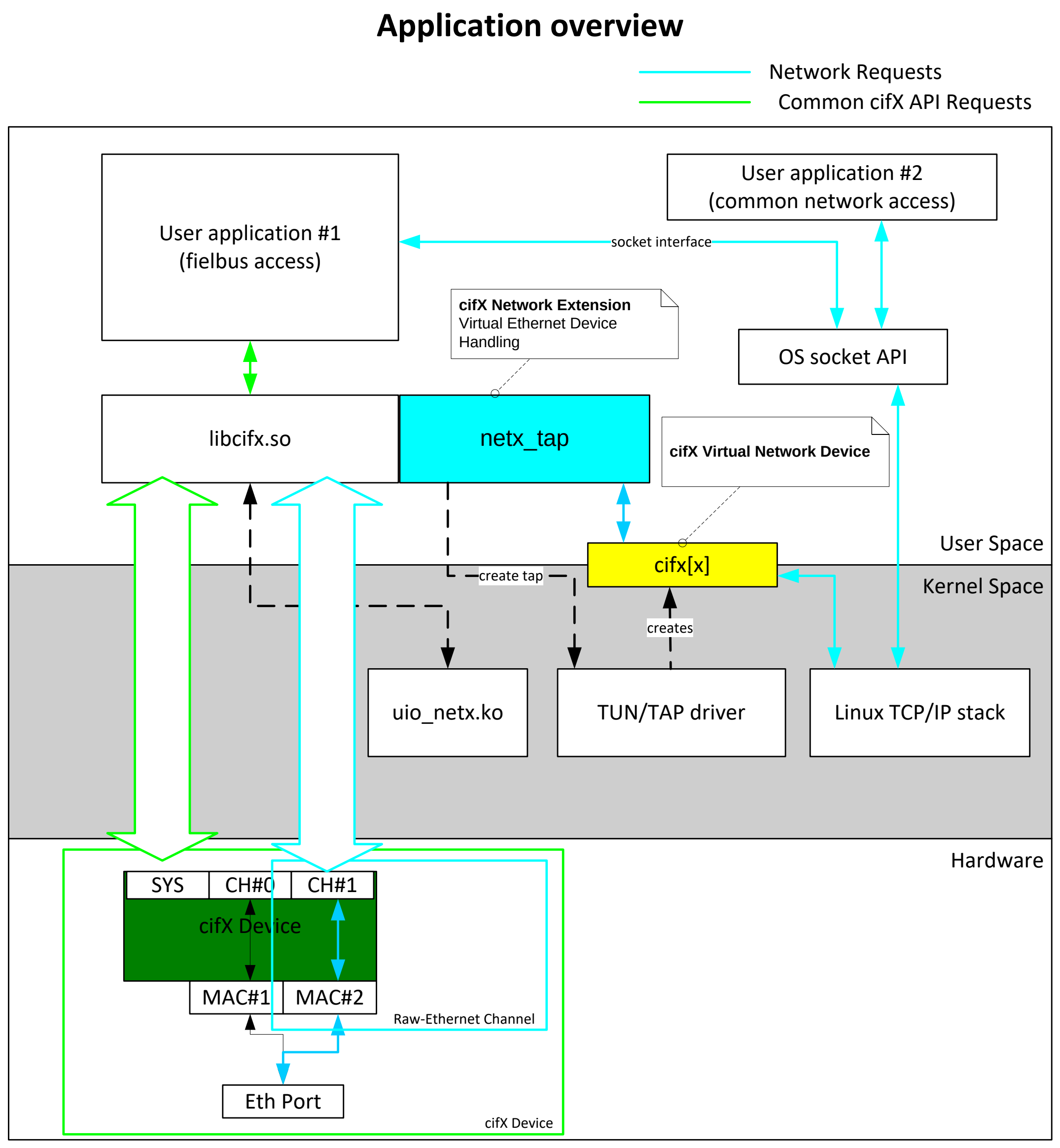


Figure 4: Virtual cifX network interface – Application overview

The *netx_tap* module is an extension of the user space library *libcifx* and manages the virtual Ethernet iterface handling. In case of cifX-Ethernet support is enabled, see section *Virtual cifX Ethernet interface setup* on page 56, the driver searches for an appropriate channel providing Raw-Ethernet support. If a channel is detected the *netx_tap* extension attaches via the cifX API (libcifx) and creates a virtual network interface *'netx_tap device'*.

The creation of the virtual network interface and all of its required initialization is done by the TUN/TAP driver. During runtime the *netx_tap* module transfers the network data from the cifx device to the netx_tap device and vice versa.

From the application point of view network requests are routed through the Linux network API through the TUN/TAP driver over the libcifX to the device.

It is also possible to access the device via the common cifX API in parallel.

4.6.5 Virtual cifX Ethernet interface setup

Prerequisites

Make sure to configure at least one card to run a firmware providing the Raw-Ethernet support, see section *Firmware and configuration file storage* on page 26.

Setup

- **Build the user space library libcifx providing the cifX Ethernet extension**
  - Build and install the user space library libcifx with Ethernet support enabled, see section *Compiling the cifX userspace library* on page 18.

    ./configure –enable-cifxethernet
  - Enable Ethernet support for the device providing the firmware with an extra communication channel for Raw-Ethernet Support, see section *Device configuration (device.conf)* on page 52.

    eth=yes
- **Start an application which initializes the driver**

  Note: The initialization options (see *cifXDriverInit()* on page 38) must not skip the device providing the Raw-Ethernet interface.

- **Start network application accessing the cifx Ethernet interface**

  After driver initialization the virtual cifX Ethernet device should be present (see *ifconfig -a*). The device is named as its parent device (e.g. parent device cifx3 -> Ethernet interface cifx3)

Optional

- **Allow non-root users to start the application**

  Note: By default root can create a virtual ethernet interface only.

  - By default root can create a cifX Ethernet interface only. To be able to run the application as non-root user, add the CAP_NET_ADMIN capability to your application

    ***setcap cap_net_admin+pe* [name of the application] (root required)**
- **Automatic interface startup and configuration**

  Note: By default the network interface will not appear until it is configured and enabled by the administrator (root) e.g. via *ifconfig*. This interface setup can be skipped by adding an *udev* rule which automatically configures the interface.

  - Add *udev* rule, which automatically configures the Ethernet interface. A template is located on the CD (/driver/templates/udev/80-udev-cifxeth.rules, see section *CD contents* on page 8).

    cp 80-udev-cifxeth.rules /etc/udev/rules.d/
  - The previously installed rule file refers to a script named ***cifxeth***, which provides the device start and configuration. The template *udev* rule estimates the configuration script to be located under /etc/init.d/.

    cp cifxeth /etc/init.d
  - Customize the start and configuration script to your own needs. The provided template (cifxeth) will enable DHCP for every cifX Ethernet interface.

5 Using SYCON.net to configure the fieldbus system

The Hilscher fieldbus hardware has to be configured by a Windows application called SYCON.net. SYCON.net is based on the FDT/DTM concept and generates the configuration files for the hardware. It is also able to update the firmware for a specific card.

Please use the following steps to create a configuration:

- Install SYCON.net
- Open SYCON.net and create a configuration
- Store the SYCON.net configuration project and export the configuration from SYCON.net into a so called database file (NXD).
- Copy the database and the firmware files to the device configuration directory (see section *Firmware and configuration file storage* on page 26).
- Now start/restart the cifX Linux driver. This will load the firmware and configuration into the cifX card.

5.1 Remote access via TCP/IP-Server

SYCON.net is also able to connect to a remote device supporting the Hilscher 'cifX Diagnostics and Remote Access' functions.

The driver CD also includes a standalone TCP/IP server example (cifXTCPServer), offering access to a remote system with an installed CIFX hardware.

The example can be found in the examples directory of the Linux driver CD.

Note: The TCP/IP server example exclusively accesses the remote CIFX hardware without a running user application on the Linux (remote) system.
It can be used to test fieldbus configurations and running fieldbus diagnostics from SYCON.net.

6 Programming with the cifX Linux Driver

6.1 Example: Generic driver initialization

The cifX Linux driver offers the same interface described in the CIFX API. Therefore the *CIFX API - Application Programming Interface* manual (see reference [1]) can be used. This manual describes the driver functions, error codes and shows some program examples.

Note: As the driver is contained in the library linked to your application, you will need to initialize the driver by a calling the function '*cifXDriverInit*' and '*cifXDriverDeInit*'.

Initialization example

```
struct CIFX_LINUX_INIT init =
{
  .init_options        = CIFX_DRIVER_INIT_AUTOSCAN, // Find all UIO devices automatically
  .iCardNumber         = 0,       // not used when init_options set to AUTOSCAN
  .fEnableCardLocking  = 0,       // do not lock card
  .base_dir            = NULL,    // use default (/opt/cifx/)
  .poll_interval       = 0,       // use default poll interval (500ms)
  .poll_StackSize      = 0,       // used default size (0x5000 Byte)
  .trace_level         = 255,     // Enable all debugging outputs to log file
  .user_card_cnt       = 0,       // no user defined cards
  .user_cards          = NULL,    // not used
};
  /* First of all initialize toolkit */
  long lRet = cifXDriverInit(&init);

  /* TODO: Insert your application here */

  cifXDriverDeinit();
```

The installation CD includes an 'Example' directory with Linux-specific examples.

6.2 Example: Driver initialization for ISA device

6.2.1 Not using UIO driver

The following example shows how to initialize the driver to map an ISA device passed via user space liberary libcifx.

```
struct CIFX_DEVICE_T   tISADevice  = {0};
struct CIFX_LINUX_INIT tDriverInit = {0};

tISADevice.dpmaddr = 0xD0000; /* physical address to DPM of the ISA device, need to  */
                              /* be set according to the jumper settings             */
                              /* NOTE: for more information of the address setup     */
                              /* (jumper settings) refer to hardware’s documentation */
tISADevice.dpmlen  = 0x4000;  /*!< length of DPM in bytes, depends on the device     */

/* since device is not a uio device and no pci card invalidate the following parameter */
tISADevice.uio_num   = -1; /*!< uio number, -1 for non-uio devices      */
tISADevice.uio_fd    = -1; /*!< uio file handle, -1 for non-uio devices */

/* Open the system memory file (/dev/mem)        */
/* Required to map the memory of the ISA device. */
if ((iISAfd = cifx_ISA_open())<0) {
  printf("Error opening the system memory (%s)
  return -1;
}
printf("Try to map the physical dpm address to a virtual memory\n");
if ((fSucess = cifx_ISA_map_dpm( iISAfd,
                                 (void**)&tISADevice.dpm,
                                 tISADevice.dpmaddr,
                                 tISADevice.dpmlen))<0) {
  printf("Error mapping dpm (%s)!\n", strerror(errno));
  cifx_ISA_close( iISAfd);
  return -1;
} else {
{
  /* setup the standard driver initializaion structure */
  tDriverInit.init_options  = CIFX_DRIVER_INIT_NOSCAN; /* NOSCAN since we are not   */
                                                       /* interested in other cards */
  tDriverInit.user_card_cnt = 1; /* set user card count to 1 since we pass one */
                                 /* user card */
  tDriverInit.user_cards    = &tISADevice; /* the previously prepared ISA device */

  /* initialize driver */
  lRet = cifXDriverInit(&tDriverInit);

  if (CIFX_NO_ERROR == lRet) {

    /* TODO: Insert your application here */

    cifXDriverDeinit();
  }
  cifx_ISA_unmap_dpm(tISADevice.dpm, tISADevice.dpmlen);
}
cifx_ISA_close(fd_isa);
```

For an ISA example application see ISASample, CD contents on page 8.

Note: Using this method does not allow using interrupt mode on ISA devices.

6.2.2 Using UIO driver

The following example shows how to initialize the driver to map an ISA device passed via the uio_netx kernel mode driver.

1. Pass ISA card to uio driver:

```
modprobe uio_netx user_dpm_start=0xD0000 user_dpm_len=0x4000 user_irq=5
```

2. Initialize driver as required (e.g. AUTOSCAN for devices)

```
struct CIFX_LINUX_INIT init =
{
  .init_options        = CIFX_DRIVER_INIT_AUTOSCAN, // Find all UIO devices automatically
  .iCardNumber         = 0,       // not used when init_options set to AUTOSCAN
  .fEnableCardLocking  = 0,       // do not lock card
  .base_dir            = NULL,    // use default (/opt/cifx/)
  .poll_interval       = 0,       // use default poll interval (500ms)
  .poll_StackSize      = 0,       // used default size (0x5000 Byte)
  .trace_level         = 255,     // Enable all debugging outputs to log file
  .user_card_cnt       = 0,       // no user defined cards
  .user_cards          = NULL,    // not used
};
  /* First of all initialize toolkit */
  long lRet = cifXDriverInit(&init);

  /* TODO: Insert your application here */

  cifXDriverDeinit();
```

Note: This method does allow using interrupt mode on ISA devices.

6.2.3 Using the Device Tree

Note: This method requires appropriate kernel configuration. Make sure that your kernel configuration has set CONFIG_OF=y.

For Linux operating system on embedded ARM-based architecture, the device tree is a common way to describe the hardware components and its configuration. The uio-netx kernel module provides support for device tree initialization. A sample configuration is located under "/driver/templates/netx.dtsi".

The following table show the available parameter.

Name	Values	Description
compatible	"hilscher,uio-netx"	Do not change! Name of the driver to be load, always set to "hilscher,uio-netx"
reg	<[DPM address] [size]>	Physical address to the DPM of the card and size of the card (this parameter depends on the system and the hardware configuration). reg = <0xF8034000 0x10000>;
interrupt-names	"card"	Do not change! Name of the interrupt resources if IRQ pin is provided, e.g. interrupt-names = "card";
interrupts	<[number] [flags]>	Only valid if interrupt-names is set to "card" Interrupt number and flags e.g. interrupts = <168 IRQ_TYPE_LEVEL_HIGH>;
dma	0,1	Enable (1) / disables (0) DMA e.g. dma = <1>;
startuptype	"flash","ram","auto","donttouch"	Specifies the startup behavior: flash = treat as flash based device ram = treat as RAM based device auto = auto detection (RAM or Flash based) donttouch = no special initialization treatment at startup e.g. startuptype = "auto";
alias	variable	An alias how the device can be identified via the driver API Alias = "mycard";

6.3 Example: Driver initialization for custom hardware interface

6.3.1 Using the hardware read/write abstraction

The following example shows how to initialize the driver to be able to communicate via custom hardware interface. For an SPI example application see SPISample, CD contents on page 8.

```
void* CustomHwIFRead(struct CIFX_DEVICE_T* ptDev, void* pvaddr,
                      void* pvdata, uint32_t ulLen)
{
  //TODO: Need to be implemented by the customer
  /* read the given number of bytes from the DPM */

  return pvdata; /* return destination address */
}

void* CustomHwIFWrite(struct CIFX_DEVICE_T* ptDev, void* pvaddr,
                      void* pvdata, uint32_t ulLen)
{
  //TODO: Need to be implemented by the customer
  /* write the given number of bytes to the DPM */

  return pvaddr; /* return destination address */
}


int main()
{
  struct CIFX_DEVICE_T   tCustomDev  = {0};
  struct CIFX_LINUX_INIT tDriverInit = {0};

  tCustomDev.dpmaddr = 0x00;    /* not used since address is not memory mapped */
  tCustomDev.dpmlen  = 0x10000; /*!< length of DPM in bytes, depends on the device */

  /* since device is no uio device and no pci card invalidate the following parameter */
  tCustomDev.uio_num   = -1; /*!< uio number, -1 for non-uio devices      */
  tCustomDev.uio_fd    = -1; /*!< uio file handle, -1 for non-uio devices */

  /* custom hardware interface initialization */
  tCustomDev.hwif_init     = NULL; /* we need no initialization of the interface */
  tCustomDev.hwif_deinit = NULL; /* we need no initialization of the interface */
  tCustomDev.hwif_read   = CustomHwIFRead; /* custom read function */
  tCustomDev.hwif_write  = CustomHwIFWrite; /* custom read function */

  /* setup the standard driver initializaion structure */
  tDriverInit.init_options  = CIFX_DRIVER_INIT_NOSCAN; /* NOSCAN since we are not   */
                                                       /* interested in other cards */
  tDriverInit.user_card_cnt = 1; /* set user card count to 1 since we pass 1 user card */
  tDriverInit.user_cards    = & tCustomDev; /* the previously prepared device */

  /* initialize driver */
  lRet = cifXDriverInit(&tDriverInit);
  if (CIFX_NO_ERROR == lRet) {
    /* TODO: Insert your application here */

    cifXDriverDeinit();
  }
  return 0;
}
```

6.3.2 Using the SPI plugin (Linux spidev framework)

To build and install the plugin, enable the ***SPM_PLUGIN*** option if compiling the libcifx library, see *Compiling the cifX userspace library* on page 18.

By default the spidev plugin installation path will be "***/opt/cifx/plugins/***". Therefore the configuration path results in "***/opt/cifx/plugins/spm-plugin/***".

For each device, a configuration file needs to be created, named ***config*** followed by its unique index ***config[index]*** in the plugin specific configuration directory (e.g. ***/opt/cifx/plugins/spm-plugin/config0***).

The following options are available.

Parameter	**Description**
Device	Name of the SPI device to open -> see Linux spidev framework e.g. /dev/spidev0.0 Device=spidev0.0
Speed	Maximum speed to configure the driver e.g. 25Mhz Speed=25000000
Mode	Mode to be setup (0 to 3) Mode=3
ChunkSize	Chunk size (maximum size of transfer after which a new transfer will be automatically setup in bytes). If set to 0, no transfer splitting will be executed. e.g. Split transfers in case it is larger than 250 byte ChunkSize=250
Irq	Path to irq file e.g. /sys/class/gpio/gpio1/value

Sample

config0 for one device accessible in polling mode via the spidev0.0 interface. The speed is set to 25Mhz, Mode=3 and no transfer splitting is executed.

```
Device=spidev0.0
Speed=25000000
Mode=3
ChunkSize=0
```

IRQ

If using GPIO-based IRQ under Linux, only edge will available while the netX works level oriented. Therefore, the driver internal compensates this but requires the following GPIO settings:

```
direction=in
edge=rising
active_low=1
```

7 Question and answers

Troubleshooting Instruction

Try to solve the problem in the order of the noted solutions following below.

7.1 cifX Device Driver

7.1.1 Failed to install driver via build script

Make sure that the path to your project folder does not contain any whitespaces.

In case the path does not contain any whitespaces refer to the console output and analyze the given error message. In case of an imprecise error message try to install the driver manually. This might give a more detailed description.

- How to build the user space library manually - *Compiling the cifX userspace library* on page 18
- How to build the kernel modul uio_netx - *Compiling the netX UIO kernel module* on page 14

7.1.2 It is not possible to run any script located on the CD

Some files of the driver package provide special functions. E.g. the scripts are marked as executable. Extracting the sources under another operating system than Linux may clear such attributes and permissions. Therefore make sure to choose the ‘.tar.bz’ archive of the driver, located on the CD and extract it under Linux, see section *Preparation* on page 12.

7.1.3 Failed to load the uio_netx kernel module

Note: To be able to load the kernel module root privileges are required.

- Refer to the error message returned when loading the module.
- Make sure the required uio module is already loaded (dump the list of the currently loaded modules)

  Run the ***lsmod*** command.
- Refer to information kept in the kernel log.

  Print the kernel log message (e.g. via ***dmesg***).

7.1.4 Unable to access or find a device

- Refer to the log file of the driver

  How to enable the drivers log file – see section *Linux driver-specific information* on page 33.
- Verify to have the correct permissions to access a device.

  Refer to the restrictions listed in section *Linux driver-specific information* on page 33.
- Failed to map the DPM

  Go to section *Failed to map the DPM of a device* on page 65.
- Make sure the no other application is running and already accessing the device.

7.1.5 Failed to map the DPM of a device

To allow mapping of the DPM to a user application, make sure the application is allowed to *mmap* enough memory (at least 64 Kbyte). You can check the current memory lock limit using the following command, which returns the maximum possible mapped memory in kB: ***ulimit -l***

7.1.6 cifX device is not correctly configured

The device appears without or with the wrong firmware/configuration being flashed

- Make sure the device configuration is correctly setup.

  Refer to the cifX log (cifX[x].log) file located in the driver's configuration directory.
- Refer to the driver's log file and make sure according to the chosen configuration method, the appropriate folder structure is created. For more information see section *Firmware and configuration file storage* on page 26.
- If no driver log file can be found – see section *No log file of the user space driver is created* on page 65.

7.1.7 No log file of the user space driver is created

If the driver's tracing feature is enabled, by default the driver tries to create a log file in the driver's configuration directory. If this fails the driver will print the debug messages to the console. Error messages, which appear before log file creation, will be printed to '*stderr*'.

- How to enable the drivers log file – see section *Linux driver-specific* information on page 33.
- Make sure to have to correct access rights to the driver's configuration directory (read+write!)

7.1.8 Failed request DMA state or to exchange IO-data via DMA

DMA support needs to be enabled during build of both driver components

- Make sure to enable DMA support during built of the kernel module uio_netx

  Compiling the netX UIO kernel module on page 14
- Make sure to enable DMA support during built user space driver libcifx

  Compiling the cifX userspace library on page 18

7.2 netX-based virtual Ethernet interface

7.2.1 Failed to create a virtual Ethernet interface

Note: A virtual Ethernet interface will be created during the driver's initialization and its lifetime is bind to the applications lifetime, which initializes the driver.

- Refer to the error message printed to *stderr* or the cifX log file.
- Make sure to enable the Ethernet extension when building the user space library libcifx.

  How to build the user space library manually - *Compiling the cifX userspace library* on page 18.
- Ethernet support needs to be enabled per device. Make sure to enable Ethernet support on the device with the firmware providing the Raw-Ethernet channel.

  Refer to Device configuration (device.conf) on page 52.
- Make sure to have the correct permissions to be able to create a Virtual Ethernet interface.

  see CAP_NET_ADMIN – section *Virtual cifX Ethernet interface setup* on page 56.
- If the previous steps does not solve the problem, go on with section *No cifX Ethernet device appears* on page 66.

7.2.2 No cifX Ethernet device appears

The device may already be created but still not active. An Ethernet interface still needs to be enabled by the administrator.

- Make sure the application which initializes the driver is running without any errors.

  Go to section *Failed to create a virtual Ethernet interface* on page 66.
- Verify if the interface is already created by running the command ***ifconfig –a***
  - If device is not present go on with *Failed to create a virtual Ethernet interface* on page 66.
  - If device is present verify the automated setup and configuration - *Virtual cifX Ethernet interface setup* on page 56.

7.2.3 No network access although device successfully created

- On some distributions, configuring more than one network adapter to the very same subnet may lead into communication errors

  Make sure to configure only one adapter per subnet

7.2.4 Network adapter disappears during device reset

When resetting a device or the system channel all of its channels will be re-initialized. Therefore a reset of a device, offering a virtual *cifX Ethernet Interface,* as a consequence, also restarts the *cifX Ethernet interface* and all connections using the Ethernet interface get interrupted.

8 Appendix

8.1 List of tables

8.2 List of figures

8.3 Legal Notes

Copyright

Important notes

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

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

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

Liability disclaimer

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

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

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

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

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

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

Warranty

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

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

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

Costs of support, maintenance, customization and product care

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

Additional guarantees

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

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

Confidentiality

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

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

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

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

Export provisions

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

8.4 Contacts

Headquarters

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

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

Subsidiaries

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

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

France
Hilscher France S.a.r.l.
69800 Saint Priest
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