User Manual NanoLib

C++

Valid with product version 1.3.0

User Manual Version: 1.4.2

Contents

1 Document aim and conventions 4

2 Before you start 5
2.1 System and hardware requirements 5
2.2 Intended use and audience 5
2.3 Scope of delivery and warranty 5

3 The *NanoLib* architecture 7
3.1 User interface 7
3.2 *NanoLib* core 7
3.3 Communication libraries 7

4 Getting started 8
4.1 Prepare your system 8
4.2 Install the *Ixxat* adapter driver for Windows 8
4.3 Install the *Peak* adapter driver for Windows 8
4.4 Install the *Ixxat* adapter driver for Linux 8
4.5 Install the *Peak* adapter driver for Linux 9
4.6 Connect your hardware 9
4.7 Load *NanoLib* 9

5 Starting the example project 10

6 Creating your own Windows project 12
6.1 Import *NanoLib* 12
6.2 Configure your project 12
6.3 Build your project 13

7 Creating your own Linux project 14
7.1 Install the shared objects with Makefile 14
7.2 Install the shared objects by hand 14
7.3 Create your project 14
7.4 Compile and test your project 15

8 Classes / functions reference 16
8.1 NanoLibAccessor 16
8.2 BusHardwareId 27
8.3 BusHardwareOptions 28
8.4 BusHwOptionsDefault 29
8.5 CanBaudRate 30
8.6 CanBus 30
8.7 CanOpenNmtService 30
8.8 CanOpenNmtState 30
8.9 EtherCATBus struct 30
8.10 EtherCATState struct 31

8.11 Ixxat....32
8.12 IxxatAdapterBusNumber....32
8.13 Peak....32
8.14 PeakAdapterBusNumber....32
8.15 DeviceHandle....32
8.16 DeviceId....33
8.17 LogLevelConverter....35
8.18 LogModuleConverter....35
8.19 ObjectDictionary....35
8.20 ObjectEntry....36
8.21 ObjectSubEntry....37
8.22 OdIndex....39
8.23 OdLibrary....40
8.24 OdTypesHelper....40
8.25 RESTfulBus struct....42
8.26 ProfinetDCP....42
8.27 ProfinetDevice struct....44
8.28 Result classes....44
8.28.1 ResultVoid....45
8.28.2 ResultInt....45
8.28.3 ResultString....46
8.28.4 ResultArrayByte....46
8.28.5 ResultArrayInt....47
8.28.6 ResultBusHwIds....47
8.28.7 ResultDeviceId....48
8.28.8 ResultDeviceIds....48
8.28.9 ResultDeviceHandle....49
8.28.10 ResultObjectDictionary....49
8.28.11 ResultConnectionState....50
8.28.12 ResultObjectEntry....50
8.28.13 ResultObjectSubEntry....51
8.28.14 ResultProfinetDevices....51
8.28.15 ResultSampleDataArray....52
8.28.16 ResultSamplerState....52
8.29 NlcErrorCode....53
8.30 NlcCallback....54
8.31 NlcDataTransferCallback....54
8.32 NlcScanBusCallback....54
8.33 NlcLoggingCallback....54
8.34 SamplerInterface....54
8.35 SamplerConfiguration struct....56
8.36 SamplerNotify....56
8.37 SampleData struct....57
8.38 SampledValue struct....57
8.39 SamplerTrigger struct....57
8.40 Serial struct....57
8.41 SerialBaudRate struct....58
8.42 SerialParity struct....58

9 Licenses....59

10 Imprint, contact, versions....60

1 Document aim and conventions

This document describes the setup and use of the *NanoLib* library and contains a reference to all classes and functions for programming your own control software for Nanotec controllers. We use the following typefaces:

Underlined text marks a cross reference or hyperlink.

- Example 1: For exact instructions on the NanoLibAccessor, see Setup.
- Example 2: Install the Ixxat driver and connect the CAN-to-USB adapter.

Italic text means: This is a *named object*, a *menu path / item,* a *tab / file name* or (if necessary) a *foreign-language* expression.

- Example 1: Select *File* > *New* > *Blank Document*. Open the *Tool* tab and select *Comment*.
- Example 2: This document divides users (= *Nutzer; usuario; utente; utilisateur; utente* etc.) from:
  - Third-party user (= *Drittnutzer; tercero usuario; terceiro utente; tiers utilisateur; terzo utente* etc.).
  - End user (= *Endnutzer; usuario final; utente final; utilisateur final; utente finale* etc.).

`Courier` marks `code blocks` or `programming commands`.

- Example 1: Via Bash, call `sudo make install` to copy shared objects; then call `ldconfig`.
- Example 2: Use the following NanoLibAccessor function to change the logging level in NanoLib:

```
//
        ***** C++ variant *****
 void setLoggingLevel(LogLevel level);
```

Bold text emphasizes individual words of **critical** importance. Alternatively, bracketed exclamation marks emphasize the critical(!) importance.

- Example 1: Protect yourself, others and your equipment. Follow our **general** safety notes that are generally applicable to **all** Nanotec products.
- Example 2: For your own protection, also follow **specific** safety notes that apply to **this** specific product.

The verb *to co-click* means a click via secondary mouse key to open a context menu etc.

- Example 1: Co-click on the file, select *Rename*, and rename the file.
- Example 2: To check the properties, co-click on the file and select *Properties*.

2 Before you start

Before you start using *NanoLib*, do prepare your PC and inform yourself about the intended use and the library limitations.

2.1 System and hardware requirements

NOTICE

Malfunction from 32-bit operation or discontinued system!

►Use, and consistently maintain, a 64-bit system.

►Observe OEM discontinuations and ~instructions.

NanoLib ***1.3.0*** supports all Nanotec products with CANopen, Modbus RTU (also USB on virtual *com* port), Modbus TCP, EtherCat, and Profinet. For **older** NanoLibs: See changelog in the imprint. At **your** risk only: legacy-system use. **Note:** Follow valid OEM instructions to set the latency as low as possible if you face problems when using an FTDI-based USB adapter.

Requirements (64-bit system mandatory)

Windows 10 or 11 w/ *Visual Studio 2019 version 16.8 or later and Windows SDK 10.0.20348.0 (version 2104) or later*

- *C++ redistributables 2017* or higher
- CANopen: *Ixxat* VCI or PCAN basic driver (optional)
- EtherCat module / Profinet DCP: *Npcap* or *WinPcap*
- RESTful module: *Npcap, WinPcap,* or admin permission to communicate w/ Ethernet bootloaders

Linux w/ *Ubuntu* 20.04 LTS to *24* (all x64 and arm64)

- Kernel headers and *libpopt-dev* packet
- Profinet DCP: `CAP_NET_ADMIN` and `CAP_NET_RAW` abilities
- CANopen: *Ixxat* ECI driver or *Peak* PCAN-USB adapter
- EtherCat: `CAP_NET_ADMIN, CAP_NET_RAW` and `CAP_SYS_NICE` abilities
- RESTful: `CAP_NET_ADMIN` ability to communicate w/ Ethernet bootloaders (also recommended: `CAP_NET_RAW`)

Language, fieldbus adapters, cables

C++ *GCC 7 or higher (Linux)*

- EtherCAT: *Ethernet cable*
- VCP / USB hub: *now uniform USB*
- USB mass storage: *USB cable*
- REST: *Ethernet cable*
- CANopen: *Ixxat USB-to-CAN V2; Nanotec ZK-USB-CAN-1, Peak PCAN-USB adapter* **No** Ixxat support for *Ubuntu* on *arm64*
- Modbus RTU: *Nanotec ZK-USB-RS-485-1* or equivalent adapter; USB cable on virtual *com* port (VCP)
- Modbus TCP: *Ethernet cable as per product datasheet*

2.2 Intended use and audience

NanoLib is a program library and software component for the operation of, and communication with, Nanotec controllers in a wide range of industrial applications – and for duly skilled programmers only.

Due to real-time incapable hardware (PC) and operating system, NanoLib is not for use in applications that need synchronous multi-axis movement or are generally time-sensitive.

In no case may you integrate NanoLib as a safety component into a product or system. On delivery to end users, you must add corresponding warning notices and instructions for safe use and safe operation to each product with a Nanotec-manufactured component. You must pass all Nanotec-issued warning notices right to the end user.

2.3 Scope of delivery and warranty

NanoLib comes as a **.zip* folder from our download website for either EMEA / APAC or AMERICA. Duly store and unzip your download before setup. The NanoLib package contains:

- Interface headers as source code (API)
- Libraries that facilitate communication: *nanolibm_[yourfieldbus].dll* etc.
- Core functions as libraries in binary format: *nanolib.dll*
- Example project: *Example.sln* (Visual Studio project) and *example.cpp* (main file)

For scope of warranty, please observe a) our terms and conditions for either EMEA / APAC or AMERICA and b) all license terms. **Note:** Nanotec is not liable for faulty or undue quality, handling, installation, operation, use, and maintenance of third-party equipment! For due safety, always follow valid OEM instructions.

3 The *NanoLib* architecture

NanoLib's modular software structure lets you arrange freely customizable motor controller / fieldbus functions around a strictly pre-built core. NanoLib contains the following modules:

User interface (API)

Interface and helper classes which

- access you to your controller's OD (object dictionary)
- base on the NanoLib core functionalities.

NanoLib core

Libraries which

- implement the API functionality
- interact with bus libraries.

Communication libraries

Fieldbus-specific libraries which

- do interface between NanoLib core and bus hardware.

3.1 User interface

The user interface consists of header interface files you can use to access the controller parameters. The user interface classes as described in the Classes / functions reference allow you to:

- Connect to both the hardware (fieldbus adapter) and the controller device.
- Access the OD of the device, to read/write the controller parameters.

3.2 *NanoLib* core

The *NanoLib* core comes with the import library *nanolib.lib*. It implements the user interface functionality and is responsible for:

- Loading and managing the communication libraries.
- Providing the user interface functionalities in the NanoLibAccessor. This communication entry point defines a set of operations you can execute on the NanoLib core and communication libraries.

3.3 Communication libraries

In addition to *nanotec.services.nanolib.dll* (useful for your optional *Plug & Drive Studio*), *NanoLib* offers the following communication libraries:

- *nanolibm_canopen.dll*
- *nanolibm_ethercat.dll*
- *nanolibm_usbmmsc.dll*
- *nanolibm_modbus.dll*
- *nanolibm_restful-api.dll*
- *nanolibm_profinet.dll*

All libraries lay a hardware abstraction layer between core and controller. The core loads them at startup from the designated project folder and uses them to establish communication with the controller by corresponding protocol.

4 Getting started

Read how to set up *NanoLib* for your operating system duly and how to connect hardware as needed.

4.1 Prepare your system

Before installing the adapter drivers, do prepare your PC along the operating system first. To prepare the PC along your Windows OS, install *MS Visual Studio* with *C++* extensions. To install *make* and *gcc* by *Linux Bash,* call `sudo apt install build-essentials`. Do then enable `CAP_NET_ADMIN`, `CAP_NET_RAW`, and `CAP_SYS_NICE` capabilities for the application that uses NanoLib:

1. Call `sudo setcap 'cap_net_admin,cap_net_raw,cap_sys_nice+eip' <application_name>`.
2. Only then, install your adapter drivers.

4.2 Install the *Ixxat* adapter driver for Windows

Only after due driver installation, you may use Ixxat's *USB-to-CAN V2* adapter. Read the USB drives' product manual, to learn if / how to activate the virtual comport (VCP).

1. Download and install Ixxat's VCI 4 driver for Windows from www.ixxat.com.
2. Connect Ixxat's USB-to-CAN V2 compact adapter to the PC via USB.
3. By Device Manager: Check if both driver and adapter are duly installed/recognized.

4.3 Install the *Peak* adapter driver for Windows

Only after due driver installation, you may use Peak's *PCAN-USB* adapter. Read the USB drives' product manual, to learn if / how to activate the virtual comport (VCP).

1. Download and install the Windows device driver setup (= installation package w/ device drivers, tools, and APIs) from http://www.peak-system.com.
2. Connect Peak's PCAN-USB adapter to the PC via USB.
3. By Device Manager: Check if both driver and adapter are duly installed/recognized.

4.4 Install the *Ixxat* adapter driver for Linux

Only after due driver installation, you may use Ixxat's *USB-to-CAN V2* adapter. **Note:** Other supported adapters need your permissions by `sudo chmod +777/dev/ttyACM*` (* device number). Read the USB drives' product manual, to learn if / how to activate the virtual comport (VCP).

1. Install the software needed for the ECI driver and demo application:

```
sudo apt-get update
apt-get install libusb-1.0-0-dev libusb-0.1-4 libc6 libstdc++6 libgcc1 build-
essential
```

2. Download the ECI-for-Linux driver from www.ixxat.com. Unzip it via:

```
unzip eci_driver_linux_amd64.zip
```

3. Install the driver via:

```
cd /EciLinux_amd/src/KernelModule
sudo make install-usb
```

4. Check for successful driver installation by compiling and starting the demo application:

```
cd /EciLinux_amd/src/EciDemos/
sudo make
cd /EciLinux_amd/bin/release/
./LinuxEciDemo
```

4.5 Install the *Peak* adapter driver for Linux

Only after due driver installation, you may use Peak's *PCAN-USB* adapter. **Note:** Other supported adapters need your permissions by `sudo chmod +777/dev/ttyACM*` (* device number). Read the USB drives' product manual, to learn if / how to activate the virtual comport (VCP).

1. Check if your Linux has kernel headers: `ls /usr/src/linux-headers-`uname -r``. **If not,** install them:

```
sudo apt-get install linux-headers-`uname -r`
```

2. Only now, install the *libpopt-dev* packet:

```
sudo apt-get install libpopt-dev
```

3. Download the needed driver package (*peak-linux-driver-xxx.tar.gz*) from www.peak-system.com.
4. To unpack it, use:

```
tar xzf peak-linux-driver-xxx.tar.gz
```

5. In the unpacked folder: Compile and install the drivers, PCAN base library, etc.:

```
make all
```

```
sudo make install
```

6. To check the function, plug the PCAN-USB adapter in.
   a) Check the kernel module:

```
lsmod | grep pcan
```

   b) … and the shared library:

```
ls -l /usr/lib/libpcan*
```

Note: If USB3 problems occur, use a USB2 port.

4.6 Connect your hardware

To be able to run a NanoLib project, connect a compatible Nanotec controller to the PC using your adapter.

1. By a suitable cable, connect your adapter to the controller.
2. Connect the adapter to the PC according to the adapter data sheet.
3. Power on the controller using a suitable power supply.
4. If needed, change the Nanotec controller's communication settings as instructed in its product manual.

4.7 Load *NanoLib*

For a first start with quick-and-easy basics, you may (but must not) use our example project.

1. Depending on your region: Download NanoLib from our website for either EMEA / APAC or AMERICA.
2. Unzip the package's files / folders and do select one option:

- **For quick-and easy basics:** See Starting the example project.
- **For advanced customizing in Windows:** See Creating your own Windows project.
- **For advanced customizing in Linux:** See Creating your own Linux project.

5 Starting the example project

With *NanoLib* duly loaded, the example project shows you through NanoLib usage with a Nanotec controller. **Note:** For each step, comments in the provided example code explain the functions used. The example project consists of:

- the '**_functions_example.*'* files, which contain the implementations for the NanoLib interface functions
- the '**_callback_example.*'* files, which contain implementations for the various callbacks (scan, data and logging)
- the '*menu_*.**' file, which contains the menu logic and code
- the *Example.** file, which is the main program, creating the menu and initializing all used parameters
- the *Sampler_example.** file, which contains the example implementation for sampler usage.

You can find more examples, with some motion commands for various operation modes, in the *Knowledge Base* at nanotec.com. All are usable in Windows or Linux.

In Windows with Visual Studio

1. Open the *Example.sln* file.
2. Open the *example.cpp*.
3. Compile and run the example code.

In Linux via Bash

1. Unzip the source file, navigate to the folder with unzipped content. The main file for the example is *example.cpp*.
2. In the bash, call:
   a. `"sudo make install"` to copy the shared objects and call `ldconfig`.
   b. `"make all"` to build the test executable.
3. The *bin* folder contains an executable *example* file. By bash: Go to the output folder and type *./example*.
   → If no error occurs, your shared objects are now duly installed, and your library is ready for use.
   → If the error reads `./example: error while loading shared libraries: libnanolib.so: cannot open shared object file: No such file or directory`, the shared objects' installation failed. In this case, follow the next steps.
4. Create a new folder within /usr/local/lib (admin rights needed). Into the bash, thus type:

```
sudo mkdir /usr/local/lib/nanotec
```

5. Copy all shared objects from the zip file's *lib* folder:

```
install ./lib/*.so /usr/local/lib/nanotec/
```

6. Check the content of the target folder with:

```
ls -al /usr/local/lib/nanotec/
```

   → It should list the shared object files from the *lib* folder.
7. Run *ldconfig* on this folder:

```
sudo ldconfig /usr/local/lib/nanotec/
```

The example is implemented as a CLI application and provides a menu interface. The menu entries are context based and will be enabled or disabled, depending on the context state. They offer you the possibility to select and execute various library functions following the typical workflow for handling a controller:

1. Check the PC for connected hardware (adapters) and list them.
2. Establish connection to an adapter.
3. Scan the bus for connected controller devices.
4. Connect to a device.

5. Test one or more of the library functions: Read/write from/to the controller's object dictionary, update the firmware, upload and run a *NanoJ* program, get the motor running and tune it, configure and use the logging/sampler.
6. Close the connection, *first* to the device, *then* to the adapter.

6 Creating your own Windows project

Create, compile and run your own Windows project to use *NanoLib*.

6.1 Import *NanoLib*

Import the *NanoLib* header files and libraries via MS Visual Studio.

1. Open Visual Studio.
2. Via *Create new project* > *Console App C++* > *Next:* Select a project type.
3. Name your project (here: *NanolibTest*) to create a project folder in the Solution Explorer.
4. Select *Finish.*
5. Open the windows file explorer and navigate to the new created project folder.
6. Create two new folders, *inc* and *lib*.
7. Open the NanoLib package folder.
8. From there: Copy the header files from the *include* folder into your project folder *inc* and all *.lib* and *.dll* files to your new project folder *lib*.
9. Check your project folder for due structure, for example:

```
.
├── NanolibTest
│   ├── inc
│   │   ├── accessor_factory.hpp
│   │   ├── bus_hardware_id.hpp
│   │   ├── ...
│   │   ├── od_index.hpp
│   │   └── result_od_entry.hpp
│   ├── lib
│   │   ├── nanolibm_canopen.dll
│   │   ├── nanolib.dll
│   │   ├── ...
│   │   └── nanolib.lib
│   ├── NanolibTest.cpp
│   ├── NanolibTest.vcxproj
│   ├── NanolibTest.vcxproj.filters
│   └── NanolibTest.vcxproj.user
└── NanolibTest.sln
```

6.2 Configure your project

Use the Solution Explorer in MS Visual Studio to set up *NanoLib* projects. **Note:** For correct NanoLib operation, select the release (not debug!) configuration in Visual C++ project settings; then build and link the project with VC runtimes of C++ redistributables **[2022]**.

1. In the Solution Explorer: Go to your project folder (here: *NanolibTest*).
2. Co-click the folder to open the context menu.
3. Select *Properties.*
4. Activate *All configurations* and *All platforms.*
5. Select *C/C++* and go to *Additional Include Directories.*
6. Insert: `$(ProjectDir)Nanolib/includes;%(AdditionalIncludeDirectories)`
7. Select *Linker* and go to *Additional Library Directories.*
8. Insert: `$(ProjectDir)Nanolib;%(AdditionalLibraryDirectories)`
9. Extend *Linker* and select *Input.*
10. Go to *Additional Dependencies* and insert: `nanolib.lib;%(AdditionalDependencies)`
11. Confirm via *OK.*

12. Go to *Configuration* > *C++* > *Language* > *Language Standard* > *ISO C++17 Standard* and set the language standard to *C++17* (/std:c++17).

6.3 Build your project

Build your *NanoLib* project in MS Visual Studio.

1. Open the main **.cpp* file (here: *nanolib_example.cpp*) and edit the code, if needs be.
2. Select *Build* > *Configuration Manager*.
3. Change *Active solution platforms* to *x64*.
4. Confirm via *Close*.
5. Select *Build* > *Build solution*.
6. No `error`? Check if your compile output duly reports:

```
1>------ Clean started: Project: NanolibTest, Configuration: Debug x64 ------
========== Clean: 1 succeeded, 0 failed, 0 skipped ==========
```

7 Creating your own Linux project

Create, compile and run your own Linux project to use *NanoLib*.

1. In the unzipped NanoLib installation kit: Open *<root>/nanotec_nanolib*.
2. Find all shared objects in the *tar.gz* file.
3. Select one option: Install each *lib* either with a Makefile or by hand.

7.1 Install the shared objects with Makefile

Use Makefile with Linux Bash to auto-install all default **.so* files.

1. Via Bash: Go to the folder containing the *makefile*.
2. Copy the shared objects via:

```
sudo make install
```

3. Confirm via:

```
ldconfig
```

7.2 Install the shared objects by hand

Use a Bash to install all **.so* files of *NanoLib* manually.

1. Via Bash: Create a new folder within */usr/local/lib*.
2. Admin rights needed! Type:

```
sudo mkdir /usr/local/lib/nanotec
```

3. Change to the unzipped installation package folder.
4. Copy all shared objects from the lib folder via:

```
install ./nanotec_nanolib/lib/*.so /usr/local/lib/nanotec/
```

5. Check the content of the target folder via:

```
ls -al /usr/local/lib/nanotec/
```

6. Check if all shared objects from the *lib* folder are listed.
7. Run `ldconfig` on this folder via:

```
sudo ldconfig /usr/local/lib/nanotec/
```

7.3 Create your project

With your shared objects installed: Create a new project for your Linux *NanoLib*.

1. Via Bash: Create a new project folder (here: *NanoLibTest*) via:

```
mkdir NanoLibTest
 cd NanoLibTest
```

2. Copy the header files to an include folder (here: *inc*) via:

```
mkdir inc
 cp /<PLACE WHERE THE CONTENT OF THE ZIP FILE IS>/nanotec_nanolib/inc/*.hpp
 inc
```

3. Create a main file (*NanoLibTest.cpp*) via:

```
#include "accessor_factory.hpp"
 #include <iostream>
```

```
int main(){
       nlc::NanoLibAccessor *accessor = getNanoLibAccessor();
nlc::ResultBusHwIds result =
       accessor->listAvailableBusHardware();
if(result.hasError()) { std::cout <<
       result.getError() << std::endl; }
else{ std::cout << "Success" << std::endl;
       }
delete accessor;
return 0;
}
```

4. Check your project folder for due structure:

```
.
└── NanoLibTest
    ├── inc
    │   ├── accessor_factory.hpp
    │   ├── bus_hardware_id.hpp
    │   ├── ...
    │   ├── od_index.hpp
    │   └── result.hpp
    └── NanoLibTest.cpp
```

7.4 Compile and test your project

Make your Linux *NanoLib* ready for use via Bash.

1. Via Bash: Compile the main file via:

```
g++ -Wall -Wextra -pedantic -I./inc -c NanoLibTest.cpp -o
    NanoLibTest
```

2. Link the executable together via:

```
g++ -Wall -Wextra -pedantic -I./inc -o test NanoLibTest.o -
L/usr/local/lib/nanotec -lnanolib -ldl
```

3. Run the test program via:

```
./test
```

4. Check if your Bash duly reports:

```
success
```

8 Classes / functions reference

Find here a list of *NanoLib's* user interface classes and their member functions. The typical description of a function includes a short introduction, the function definition and a parameter / return list:

ExampleFunction ()

Tells you briefly what the function does.

```
virtual void nlc::NanoLibAccessor::ExampleFunction (Param_a const & param_a,
 Param_b const & param_B)
```

Parameters	*param_a*	Additional comment if needed.
	param_b	
Returns	*ResultVoid*	Additional comment if needed.

8.1 NanoLibAccessor

Interface class used as entry point to the *NanoLib*. A typical workflow looks like this:

1. Start by scanning for hardware with NanoLibAccessor.listAvailableBusHardware ().
2. Set the communication settings with BusHardwareOptions ().
3. Open the hardware connection with NanoLibAccessor.openBusHardwareWithProtocol ().
4. Scan the bus for connected devices with NanoLibAccessor.scanDevices ().
5. Add a device with NanoLibAccessor.addDevice ().
6. Connect to the device with NanoLibAccessor.connectDevice ().
7. After finishing the operation, disconnect the device with NanoLibAccessor.disconnectDevice ().
8. Remove the device with NanoLibAccessor.removeDevice ().
9. Close the hardware connection with NanoLibAccessor.closeBusHardware ().

NanoLibAccessor has the following public member functions:

listAvailableBusHardware ()

Use this function to list available fieldbus hardware.

```
virtual ResultBusHwIds nlc::NanoLibAccessor::listAvailableBusHardware ()
```

Returns	*ResultBusHwIds*	Delivers a fieldbus ID array.

openBusHardwareWithProtocol ()

Use this function to connect bus hardware.

```
virtual ResultVoid nlc::NanoLibAccessor::openBusHardwareWithProtocol
 (BusHardwareId const & busHwId, BusHardwareOptions const & busHwOpt)
```

Parameters	*busHwId*	Specifies the fieldbus to open.
	busHwOpt	Specifies fieldbus opening options.
Returns	*ResultVoid*	Confirms that a void function has run.

isBusHardwareOpen ()

Use this function to check if your fieldbus hardware connection is open.

```
virtual ResultVoid nlc::NanoLibAccessor::openBusHardwareWithProtocol (const
 BusHardwareId & busHwId, const BusHardwareOptions & busHwOpt)
```

Parameters	*BusHardwareId*	Specifies each fieldbus to open.
Returns	*true*	Hardware is open.
	false	Hardware is closed.

getProtocolSpecificAccessor ()

Use this function to get the protocol-specific accessor object.

```
virtual ResultVoid nlc::NanoLibAccessor::getProtocolSpecificAccessor
 (BusHardwareId const & busHwId)
```

Parameters	*busHwId*	Specifies the fieldbus to get the accessor for.
Returns	*ResultVoid*	Confirms that a void function has run.

getProfinetDCP ()

Use this function to return a reference to Profinet DCP interface.

```
virtual ProfinetDCP & getProfinetDCP ()
```

Returns	ProfinetDCP	

getSamplerInterface ()

Use this function to get a reference to the sampler interface.

```
virtual SamplerInterface & getSamplerInterface ()
```

Returns	*SamplerInterface*	Refers to the sampler interface class.

setBusState ()

Use this function to set the bus-protocol-specific state.

```
virtual ResultVoid nlc::NanoLibAccessor::setBusState (const BusHardwareId &
 busHwId, const std::string & state)
```

Parameters	*busHwId*	Specifies the fieldbus to open.
	state	Assigns a bus-specific state as a string value.
Returns	*ResultVoid*	Confirms that a void function has run.

scanDevices ()

Use this function to scan for devices in the network.

```
virtual ResultDeviceIds nlc::NanoLibAccessor::scanDevices (const BusHardwareId
 & busHwId, NlcScanBusCallback* callback)
```

Parameters	*busHwId*	Specifies the fieldbus to scan.
	callback	NlcScanBusCallback progress tracer.
Returns	*ResultDeviceIds*	Delivers a device ID array.
	IOError	Informs that a device is not found.

addDevice ()

Use this function to add a bus device described by *deviceId* to *NanoLib's* internal device list, and to return *deviceHandle* for it.

```
virtual ResultDeviceHandle nlc::NanoLibAccessor::addDevice (DeviceId const &
 deviceId)
```

Parameters	*deviceId*	Specifies the device to add to the list.
Returns	*ResultDeviceHandle*	Delivers a device handle.

connectDevice ()

Use this function to connect a device by *deviceHandle*.

```
virtual ResultVoid nlc::NanoLibAccessor::connectDevice (DeviceHandle const
 deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib connects to.
Returns	*ResultVoid*	Confirms that a void function has run.
	IOError	Informs that a device is not found.

getDeviceName ()

Use this function to get a device's name by *deviceHandle*.

```
virtual ResultString nlc::NanoLibAccessor::getDeviceName (DeviceHandle const
 deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib gets the name for.
Returns	*ResultString*	Delivers device names as a string.

getDeviceProductCode ()

Use this function to get a device's product code by *deviceHandle*.

```
virtual ResultInt nlc::NanoLibAccessor::getDeviceProductCode (DeviceHandle
 const deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib gets the product code for.
Returns	*ResultInt*	Delivers product codes as an integer.

getDeviceVendorId ()

Use this function to get the device vendor ID by *deviceHandle*.

```
virtual ResultInt nlc::NanoLibAccessor::getDeviceVendorId (DeviceHandle const
 deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib gets the vendor ID for.
Returns	*ResultInt*	Delivers vendor ID's as an integer.
	ResourceUnavailable	Informs that no data is found.

getDeviceId ()

Use this function to get a specific device's ID from the *NanoLib* internal list.

```
virtual ResultDeviceId nlc::NanoLibAccessor::getDeviceId (DeviceHandle const
 deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib gets the device ID for.
Returns	*ResultDeviceId*	Delivers a device ID.

getDeviceIds ()

Use this function to get all devices' ID from the *NanoLib* internal list.

```
virtual ResultDeviceIds nlc::NanoLibAccessor::getDeviceIds ()
```

Returns	*ResultDeviceIds*	Delivers a device ID list.

getDeviceUid ()

Use this function to get a device's unique ID (96 bit / 12 bytes) by *deviceHandle*.

```
virtual ResultArrayByte nlc::NanoLibAccessor::getDeviceUid (DeviceHandle const
 deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib gets the unique ID for.
Returns	*ResultArrayByte*	Delivers unique ID's as a byte array.
	ResourceUnavailable	Informs that no data is found.

getDeviceSerialNumber ()

Use this function to get a device's serial number by *deviceHandle*.

```
virtual ResultString NanolibAccessor::getDeviceSerialNumber (DeviceHandle
 const deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib gets the serial number for.
Returns	*ResultString*	Delivers serial numbers as a string.
	ResourceUnavailable	Informs that no data is found.

getDeviceHardwareGroup ()

Use this function to get a bus device's hardware group by *deviceHandle*.

```
virtual ResultDeviceId nlc::NanoLibAccessor::getDeviceHardwareGroup
 (DeviceHandle const deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib gets the hardware group for.
Returns	*ResultInt*	Delivers hardware groups as an integer.

getDeviceHardwareVersion ()

Use this function to get a bus device's hardware version by *deviceHandle*.

```
virtual ResultDeviceId nlc::NanoLibAccessor::getDeviceHardwareVersion
 (DeviceHandle const deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib gets the hardware version for.
Returns	*ResultString*	Delivers device names as a string.
	ResourceUnavailable	Informs that no data is found.

getDeviceFirmwareBuildId ()

Use this function to get a bus device's firmware build ID by *deviceHandle*.

```
virtual ResultDeviceId nlc::NanoLibAccessor::getDeviceFirmwareBuildId
 (DeviceHandle const deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib gets the firmware build ID for.
Returns	*ResultString*	Delivers device names as a string.

getDeviceBootloaderVersion ()

Use this function to get a bus device's bootloader version by *deviceHandle*.

```
virtual ResultInt nlc::NanoLibAccessor::getDeviceBootloaderVersion
 (DeviceHandle const deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib gets the bootloader version for.
Returns	*ResultInt*	Delivers bootloader versions as an integer.
	ResourceUnavailable	Informs that no data is found.

getDeviceBootloaderBuildId ()

Use this function to get a bus device's bootloader build ID by *deviceHandle*.

```
virtual ResultDeviceId nlc::NanoLibAccessor:: (DeviceHandle const
 deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib gets the bootloader build ID for.
Returns	*ResultString*	Delivers device names as a string.

rebootDevice ()

Use this function to reboot the device by *deviceHandle*.

```
virtual ResultVoid nlc::NanoLibAccessor::rebootDevice (const DeviceHandle
 deviceHandle)
```

Parameters	*deviceHandle*	Specifies the fieldbus to reboot.
Returns	*ResultVoid*	Confirms that a void function has run.

getDeviceState ()

Use this function to get the device-protocol-specific state.

```
virtual ResultString nlc::NanoLibAccessor::getDeviceState (DeviceHandle const
 deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib gets the state for.

Returns	*ResultString*	Delivers device names as a string.

setDeviceState ()

Use this function to set the device-protocol-specific state.

```
virtual ResultVoid nlc::NanoLibAccessor::setDeviceState (const DeviceHandle
 deviceHandle, const std::string & state)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib sets the state for.
	state	Assigns a bus-specific state as a string value.
Returns	*ResultVoid*	Confirms that a void function has run.

getConnectionState ()

Use this function to get a specific device's last known connection state by *deviceHandle* (= *Disconnected, Connected, ConnectedBootloader*)

```
virtual ResultConnectionState nlc::NanoLibAccessor::getConnectionState
 (DeviceHandle const deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib gets the connection state for.
Returns	*ResultConnectionState*	Delivers a connection state (= *Disconnected, Connected, ConnectedBootloader*).

checkConnectionState ()

Only if the last known state was not *Disconnected:* Use this function to check and possibly update a specific device's connection state by *deviceHandle* and by testing several mode-specific operations.

```
virtual ResultConnectionState nlc::NanoLibAccessor::checkConnectionState
 (DeviceHandle const deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib checks the connection state for.
Returns	*ResultConnectionState*	Delivers a connection state (= not *Disconnected*).

assignObjectDictionary ()

Use this **manual** function to assign an object dictionary (OD) to *deviceHandle* on your **own**.

```
virtual ResultObjectDictionary nlc::NanoLibAccessor::assignObjectDictionary
 (DeviceHandle const deviceHandle, ObjectDictionary const & objectDictionary)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib assigns the OD to.
	objectDictionary	
Returns	*ResultObjectDictionary*	Shows the properties of an object dictionary.

autoAssignObjectDictionary ()

Use this **automatism** to let ***NanoLib*** assign an object dictionary (OD) to *deviceHandle*. On finding and loading a suitable OD, NanoLib automatically assigns it to the device. **Note:** If a compatible OD is already loaded in the object library, NanoLib will automatically use it without scanning the submitted directory.

```
virtual ResultObjectDictionary
 nlc::NanoLibAccessor::autoAssignObjectDictionary (DeviceHandle const
 deviceHandle, const std::string & dictionariesLocationPath)
```

Parameters	*deviceHandle*	Specifies for which bus device NanoLib shall automatically scan for suitable OD's.
	dictionariesLocationPath	Specifies the path to the OD directory.
Returns	*ResultObjectDictionary*	Shows the properties of an object dictionary.

getAssignedObjectDictionary ()

Use this function to get the object dictionary assigned to a device by *deviceHandle*.

```
virtual ResultObjectDictionary
 nlc::NanoLibAccessor::getAssignedObjectDictionary (DeviceHandle const device
Handle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib gets the assigned OD for.
Returns	*ResultObjectDictionary*	Shows the properties of an object dictionary.

getObjectDictionaryLibrary ()

This function returns an OdLibrary reference.

```
virtual OdLibrary& nlc::NanoLibAccessor::getObjectDictionaryLibrary ()
```

Returns	*OdLibrary&*	Opens the entire OD library and its object dictionaries.

setLoggingLevel ()

Use this function to set the needed log detailing (and log file size). Default level is *Info*.

```
virtual void nlc::NanoLibAccessor::setLoggingLevel (LogLevel level)
```

Parameters	*level*	The following log detailings are possible:

0 = *Trace*	Lowest level (largest log file); logs any feasible detail, plus software start / stop.
1 = *Debug*	Logs debug information (= interim results, content sent or received, etc.)
2 = *Info*	Default level; logs informational messages.
3 = *Warn*	Logs problems that did occur but **won't** stop the current algorithm.
4 = *Error*	Logs just severe trouble that **did** stop the algorithm.
5 = *Critical*	Highest level (smallest log file); turns logging **off**; no further log at all.
6 = *Off*	No logging at all.

setLoggingCallback ()

Use this function to set a logging callback pointer and log module (= library) for that callback (not for the logger itself).

```
virtual void nlc::NanoLibAccessor::setLoggingCallback (NlcLoggingCallback*
 callback, const nlc::LogModule & logModule)
```

Parameters	**callback*	Sets a callback pointer.
	logModule	Tunes the callback (not logger!) to your library.

0 = *NanolibCore*	Activates a callback for NanoLib's core only.
1 = *NanolibCANopen*	Activates a CANopen-only callback.
2 = *NanolibModbus*	Activates a Modbus-only callback.
3 = *NanolibEtherCAT*	Activates an EtherCAT-only callback.

4 = *NanolibRest*	Activates a REST-only callback.
5 = *NanolibUSB*	Activates a USB-only callback.

unsetLoggingCallback ()

Use this function to cancel a logging callback pointer.

```
virtual void nlc::NanoLibAccessor::unsetLoggingCallback ()
```

readNumber ()

Use this function to read a numeric value from the object dictionary.

```
virtual ResultInt nlc::NanoLibAccessor::readNumber (const DeviceHandle
 deviceHandle, const OdIndex odIndex)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib reads from.
	odIndex	Specifies the (sub-) index to read from.
Returns	*ResultInt*	Delivers an uninterpreted numeric value (can be signed, unsigned, fix16.16 bit values).

readNumberArray ()

Use this function to read numeric arrays from the object dictionary.

```
virtual ResultArrayInt nlc::NanoLibAccessor::readNumberArray (const
 DeviceHandle deviceHandle, const uint16_t index)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib reads from.
	index	Array object index.
Returns	*ResultArrayInt*	Delivers an integer array.

readBytes ()

Use this function to read arbitrary bytes (domain object data) from the object dictionary.

```
virtual ResultArrayByte nlc::NanoLibAccessor::readBytes (const DeviceHandle
 deviceHandle, const OdIndex odIndex)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib reads from.
	odIndex	Specifies the (sub-) index to read from.
Returns	*ResultArrayByte*	Delivers a byte array.

readString ()

Use this function to read strings from the object directory.

```
virtual ResultString nlc::NanoLibAccessor::readString (const DeviceHandle
 deviceHandle, const OdIndex odIndex)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib reads from.
	odIndex	Specifies the (sub-) index to read from.
Returns	*ResultString*	Delivers device names as a string.

writeNumber ()

Use this function to write numeric values to the object directory.

```
virtual ResultVoid nlc::NanoLibAccessor::writeNumber (const DeviceHandle
 deviceHandle, int64_t value, const OdIndex odIndex, unsigned int bitLength)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib writes to.
	value	The uninterpreted value (can be signed, unsigned, fix 16.16).
	odIndex	Specifies the (sub-) index to read from.
	bitLength	Length in bit.
Returns	*ResultVoid*	Confirms that a void function has run.

writeBytes ()

Use this function to write arbitrary bytes (domain object data) to the object directory.

```
virtual ResultVoid nlc::NanoLibAccessor::writeBytes (const DeviceHandle
 deviceHandle, const std::vector <uint8_t> & data, const OdIndex odIndex)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib writes to.
	data	Byte vector / array.
	odIndex	Specifies the (sub-) index to read from.
Returns	*ResultVoid*	Confirms that a void function has run.

uploadFirmware ()

Use this function to update your controller firmware.

```
virtual ResultVoid nlc::NanoLibAccessor::uploadFirmware (const DeviceHandle
 deviceHandle, const std::vector <uint8_t> & fwData, NlcDataTransferCallback*
 callback)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib updates.
	fwData	Array containing firmware data.
	NlcDataTransferCallback	A data progress tracer.
Returns	*ResultVoid*	Confirms that a void function has run.

uploadFirmwareFromFile ()

Use this function to update your controller firmware by uploading its file.

```
virtual ResultVoid nlc::NanoLibAccessor::uploadFirmwareFromFile (const
 DeviceHandle deviceHandle, const std::string & absoluteFilePath,
 NlcDataTransferCallback* callback)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib updates.
	absoluteFilePath	Path to file containing firmware data (std::string).
	NlcDataTransferCallback	A data progress tracer.
Returns	*ResultVoid*	Confirms that a void function has run.

uploadBootloader ()

Use this function to update your controller bootloader.

```
virtual ResultVoid nlc::NanoLibAccessor::uploadBootloader (const DeviceHandle
 deviceHandle, const std::vector <uint8_t> & btData, NlcDataTransferCallback*
 callback)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib updates.
	btData	Array containing bootloader data.
	NlcDataTransferCallback	A data progress tracer.
Returns	*ResultVoid*	Confirms that a void function has run.

uploadBootloaderFromFile ()

Use this function to update your controller bootloader by uploading its file.

```
virtual ResultVoid nlc::NanoLibAccessor::uploadBootloaderFromFile (const
 DeviceHandle deviceHandle, const std::string & bootloaderAbsoluteFilePath,
 NlcDataTransferCallback* callback)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib updates.
	bootloaderAbsoluteFilePath	Path to file containing bootloader data (std::string).
	NlcDataTransferCallback	A data progress tracer.
Returns	*ResultVoid*	Confirms that a void function has run.

uploadBootloaderFirmware ()

Use this function to update your controller bootloader and firmware.

```
virtual ResultVoid nlc::NanoLibAccessor::uploadBootloaderFirmware (const
 DeviceHandle deviceHandle, const std::vector <uint8_t> & btData, const
 std::vector <uint8_t> & fwData, NlcDataTransferCallback* callback)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib updates.
	btData	Array containing bootloader data.
	fwData	Array containing firmware data.
	NlcDataTransferCallback	A data progress tracer.
Returns	*ResultVoid*	Confirms that a void function has run.

uploadBootloaderFirmwareFromFile ()

Use this function to update your controller bootloader and firmware by uploading the files.

```
virtual ResultVoid nlc::NanoLibAccessor::uploadBootloaderFirmwareFromFile
 (const DeviceHandle deviceHandle, const std::string &
 bootloaderAbsoluteFilePath, const std::string & absoluteFilePath,
 NlcDataTransferCallback* callback)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib updates.
	bootloaderAbsoluteFilePath	Path to file containing bootloader data (std::string).
	absoluteFilePath	Path to file containing firmware data (uint8_t).
	NlcDataTransferCallback	A data progress tracer.
Returns	*ResultVoid*	Confirms that a void function has run.

uploadNanoJ ()

Use this public function to updload the NanoJ program to your controller.

```
virtual ResultVoid nlc::NanoLibAccessor::uploadNanoJ (DeviceHandle const
 deviceHandle, std::vector <uint8_t> const & vmmData, NlcDataTransferCallback*
 callback)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib uploads to.
	vmmData	Array containing NanoJ data.
	NlcDataTransferCallback	A data progress tracer.
Returns	*ResultVoid*	Confirms that a void function has run.

uploadNanoJFromFile ()

Use this public function to updload the NanoJ program to your controller by uploading the file.

```
virtual ResultVoid nlc::NanoLibAccessor::uploadNanoJFromFile (const
 DeviceHandle deviceHandle, const std::string & absoluteFilePath,
 NlcDataTransferCallback* callback)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib uploads to.
	absoluteFilePath	Path to file containing NanoJ data (std::string).
	NlcDataTransferCallback	A data progress tracer.
Returns	*ResultVoid*	Confirms that a void function has run.

disconnectDevice ()

Use this function to disconnect your device by *deviceHandle*.

```
virtual ResultVoid nlc::NanoLibAccessor::disconnectDevice (DeviceHandle const
 deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib disconnects from.
Returns	*ResultVoid*	Confirms that a void function has run.

removeDevice ()

Use this function to remove your device from *NanoLib's* internal device list.

```
virtual ResultVoid nlc::NanoLibAccessor::removeDevice (const DeviceHandle
 deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib delists.
Returns	*ResultVoid*	Confirms that a void function has run.

closeBusHardware ()

Use this function to disconnect from your fieldbus hardware.

```
virtual ResultVoid nlc::NanoLibAccessor::closeBusHardware (BusHardwareId const
 & busHwId)
```

Parameters	*busHwId*	Specifies the fieldbus to disconnect from.
Returns	*ResultVoid*	Confirms that a void function has run.

8.2 BusHardwareId

Use this class to identify a bus hardware one-to-one or to distinguish different bus hardware from each other. This class (without setter functions to be immutable from creation on) also holds information on:

- Hardware (= adapter name, network adapter etc.)
- Bus hardware specifier (= serial port name, MAC address etc.)
- Protocol to use (= Modbus TCP, CANopen etc.)
- Friendly name

BusHardwareId () [1/3]

Constructor that creates a new bus hardware ID object.

```
nlc::BusHardwareId::BusHardwareId (std::string const & busHardware_,
 std::string const & protocol_, std::string const & hardwareSpecifier_,
 std::string const & name_)
```

Parameters	*busHardware_*	Hardware type (= ZK-USB-CAN-1 etc.).
	*protocol_*	Bus communication protocol (= CANopen etc.).
	*hardwareSpecifier_*	The specifier of a hardware (= COM3 etc.).
	*extraHardwareSpecifier_*	The extra specifier of the hardware (say, USB location info).
	*name_*	A friendly name (= *AdapterName (Port)* etc.).

BusHardwareId () [2/3]

Constructor that creates a new bus hardware ID object, with the option for an extra hardware speicifier.

```
nlc::BusHardwareId::BusHardwareId (std::string const & busHardware_,
 std::string const & protocol_, std::string const & hardwareSpecifier_,
 std::string const & extraHardwareSpecifier_, std::string const & name_)
```

Parameters	*busHardware_*	Hardware type (= ZK-USB-CAN-1 etc.).
	*protocol_*	Bus communication protocol (= CANopen etc.).
	*hardwareSpecifier_*	The specifier of a hardware (= COM3 etc.).
	*extraHardwareSpecifier_*	The extra specifier of the hardware (say, USB location info).
	*name_*	A friendly name (= *AdapterName (Port)* etc.).

BusHardwareId () [3/3]

Constructor that copies an exisiting *busHardwareId*.

```
nlc::BusHardwareId::BusHardwareId (BusHardwareId const &)
```

```
nlc::BusHardwareId::BusHardwareId (BusHardwareId const &)
```

Parameters	*busHardwareId*	Names the bus hardware ID to copy from.

equals ()

Compares a new bus hardware ID to existing ones.

```
bool nlc::BusHardwareId::equals (BusHardwareId const & other) const
```

Parameters	*other*	Another object of the same class.
Returns	*true*	If both are equal in all values.

false	If the values differ.

getBusHardware ()

Reads out the bus hardware string.

```
std::string nlc::BusHardwareId::getBusHardware () const
```

Returns *string*

getHardwareSpecifier ()

Reads out the bus hardware's specifier string (= network name etc.).

```
std::string nlc::BusHardwareId::getHardwareSpecifier () const
```

Returns *string*

getExtraHardwareSpecifier ()

Reads out the bus extra hardware's specifier string (= MAC address etc.).

```
std::string nlc::BusHardwareId::getExtraHardwareSpecifier () const
```

Returns *string*

getName ()

Reads out the bus hardware's friendly name.

```
std::string nlc::BusHardwareId::getName () const
```

Returns *string*

getProtocol ()

Reads out the bus protocol string.

```
std::string nlc::BusHardwareId::getProtocol () const
```

Returns *string*

toString ()

Returns the bus hardware ID as a string.

```
std::string nlc::BusHardwareId::toString () const
```

Returns *string*

8.3 BusHardwareOptions

Find in this class, in a key-value list of strings, all options needed to open a bus hardware.

BusHardwareOptions () [1/2]

Constructs a new bus hardware option object.

```
nlc::BusHardwareOptions::BusHardwareOptions ()
```

Use the function addOption () to add key-value pairs.

BusHardwareOptions () [2/2]

Constructs a new bus hardware options object with the key-value map already in place.

```
nlc::BusHardwareOptions::BusHardwareOptions (std::map <std::string,
 std::string> const & options)
```

Parameters	*options*	A map with options for the bus hardware to operate.

addOption ()

Creates additional keys and values.

```
void nlc::BusHardwareOptions::addOption (std::string const & key, std::string
 const & value)
```

Parameters	*key*	Example: `BAUD_RATE_OPTIONS_NAME`, see *bus_hw_options_defaults*
	value	Example: `BAUD_RATE_1000K`, see *bus_hw_options_defaults*

equals ()

Compares the BusHardwareOptions to existing ones.

```
bool nlc::BusHardwareOptions::equals (BusHardwareOptions const & other) const
```

Parameters	*other*	Another object of the same class.
Returns	*true*	If the other object has all of the exact same options.
	false	If the other object has different keys or values.

getOptions ()

Reads out all added key-value pairs.

```
std::map <std::string, std::string> nlc::BusHardwareOptions::getOptions ()
 const
```

Returns	*string map*	

toString ()

Returns all keys / values as a string.

```
std::string nlc::BusHardwareId::toString () const
```

Returns	*string*	

8.4 BusHwOptionsDefault

This default configuration options class has the following public attributes:

const CanBus	*canBus* = CanBus ()
const Serial	*serial* = Serial ()
const RESTfulBus	restfulBus = RESTfulBus()
const EtherCATBus	ethercatBus = EtherCATBus()

8.5 CanBaudRate

Struct that contains CAN bus baudrates in the following public attributes:

const std::string	`BAUD_RATE_1000K` = "1000k"
const std::string	`BAUD_RATE_800K` = "800k"
const std::string	`BAUD_RATE_500K` = "500k"
const std::string	`BAUD_RATE_250K` = "250k"
const std::string	`BAUD_RATE_125K` = "125k"
const std::string	`BAUD_RATE_100K` = "100k"
const std::string	`BAUD_RATE_50K` = "50k"
const std::string	`BAUD_RATE_20K` = "20k"
const std::string	`BAUD_RATE_10K` = "10k"
const std::string	`BAUD_RATE_5K` = "5k"

8.6 CanBus

Default configuration options class with the following public attributes:

const std::string	`BAUD_RATE_OPTIONS_NAME` = "can adapter baud rate"
const CanBaudRate	*baudRate* = CanBaudRate ()
const Ixxat	*ixxat* = Ixxat ()

8.7 CanOpenNmtService

For the NMT service, this struct contains the CANopen NMT states as string values in the following public attributes:

const std::string	`START` = "START"
const std::string	`STOP` = "STOP"
const std::string	`PRE_OPERATIONAL` = "PRE_OPERATIONAL"
const std::string	`RESET` = "RESET"
const std::string	`RESET_COMMUNICATION` = "RESET_COMMUNICATION"

8.8 CanOpenNmtState

This struct contains the CANopen NMT states as string values in the following public attributes:

const std::string	`STOPPED` = "STOPPED"
const std::string	`PRE_OPERATIONAL` = "PRE_OPERATIONAL"
const std::string	`OPERATIONAL` = "OPERATIONAL"
const std::string	`INITIALIZATION` = "INITIALIZATION"
const std::string	`UNKNOWN` = "UNKNOWN"

8.9 EtherCATBus struct

This struct contains the EtherCAT communication configuration options in the following public attributes:

const std::string NETWORK_FIRMWARE_STATE_OPTION_NAME = "Network Firmware State"	Network state treated as firmware mode. Acceptable values (default = PRE_OPERATIONAL): ■ EtherCATState::PRE_OPERATIONAL ■ EtherCATState::SAFE_OPERATIONAL ■ EtherCATState::OPERATIONAL
const std::string DEFAULT_NETWORK_FIRMWARE_STATE = "PRE_OPERATIONAL"	
const std::string EXCLUSIVE_LOCK_TIMEOUT_OPTION_NAME = "Shared Lock Timeout"	Timeout in milliseconds to acquire exclusive lock on the network (default = 500 ms).
const unsigned int DEFAULT_EXCLUSIVE_LOCK_TIMEOUT = "500"	
const std::string SHARED_LOCK_TIMEOUT_OPTION_NAME = "Shared Lock Timeout"	Timeout in milliseconds to acquire shared lock on the network (default = 250 ms).
const unsigned int DEFAULT_SHARED_LOCK_TIMEOUT = "250"	
const std::string READ_TIMEOUT_OPTION_NAME = "Read Timeout"	Timeout in milliseconds for a read operation (default = 700 ms).
const unsigned int DEFAULT_READ_TIMEOUT = "700"	
const std::string WRITE_TIMEOUT_OPTION_NAME = "Write Timeout"	Timeout in milliseconds for a write operation (default = 200 ms).
const unsigned int DEFAULT_WRITE_TIMEOUT = "200"	
const std::string READ_WRITE_ATTEMPTS_OPTION_NAME = "Read/Write Attempts"	Maximum read or write attempts (non-zero values only; default = 5).
const unsigned int DEFAULT_READ_WRITE_ATTEMPTS = "5"	
const std::string CHANGE_NETWORK_STATE_ATTEMPTS_OPTION_NAME = "Change Network State Attempts"	Maximum number of attempts to alter the network state (non-zero values only; default = 10).
const unsigned int DEFAULT_CHANGE_NETWORK_STATE_ATTEMPTS = "10"	
const std::string PDO_IO_ENABLED_OPTION_NAME = "PDO IO Enabled"	Enables or disables PDO processing for digital in- / outputs ("True" or "False" only; default = "True").
const std::string DEFAULT_PDO_IO_ENABLED = "True"	

8.10 EtherCATState struct

This struct contains the EtherCAT slave / network states as string values in the following public attributes.
Note: Default state at power on is PRE_OPERATIONAL; *NanoLib* can provide no reliable "OPERATIONAL" state in a non-realtime operating system:

const std::string	NONE = "NONE"
const std::string	INIT = "INIT"
const std::string	PRE_OPERATIONAL = "PRE_OPERATIONAL"
const std::string	BOOT = "BOOT"
const std::string	SAFE_OPERATIONAL = "SAFE_OPERATIONAL"
const std::string	OPERATIONAL = "OPERATIONAL"

8.11 Ixxat

This struct holds all information for the *Ixxat* usb-to-can in the following public attributes:

const std::string	`ADAPTER_BUS_NUMBER_OPTIONS_NAME` = "ixxat adapter bus number"
const IxxatAdapterBusNumber	*adapterBusNumber* = IxxatAdapterBusNumber ()

8.12 IxxatAdapterBusNumber

This struct holds the bus number for the *Ixxat* usb-to-can in the following public attributes:

const std::string	`BUS_NUMBER_0_DEFAULT` = "0"
const std::string	`BUS_NUMBER_1` = "1"
const std::string	`BUS_NUMBER_2` = "2"
const std::string	`BUS_NUMBER_3` = "3"

8.13 Peak

This struct holds all information for the *Peak* usb-to-can in the following public attributes:

const std::string	`ADAPTER_BUS_NUMBER_OPTIONS_NAME` = "peak adapter bus number"
const PeakAdapterBusNumber	*adapterBusNumber* = PeakAdapterBusNumber ()

8.14 PeakAdapterBusNumber

This struct holds the bus number for the *Peak* usb-to-can in the following public attributes:

const std::string	`BUS_NUMBER_1_DEFAULT` = std::to_string (PCAN_USBBUS1)
const std::string	`BUS_NUMBER_2` = std::to_string (PCAN_USBBUS2)
const std::string	`BUS_NUMBER_3` = std::to_string (PCAN_USBBUS3)
const std::string	`BUS_NUMBER_4` = std::to_string (PCAN_USBBUS4)
const std::string	`BUS_NUMBER_5` = std::to_string (PCAN_USBBUS5)
const std::string	`BUS_NUMBER_6` = std::to_string (PCAN_USBBUS6)
const std::string	`BUS_NUMBER_7` = std::to_string (PCAN_USBBUS7)
const std::string	`BUS_NUMBER_8` = std::to_string (PCAN_USBBUS8)
const std::string	`BUS_NUMBER_9` = std::to_string (PCAN_USBBUS9)
const std::string	`BUS_NUMBER_10` = std::to_string (PCAN_USBBUS10)
const std::string	`BUS_NUMBER_11` = std::to_string (PCAN_USBBUS11)
const std::string	`BUS_NUMBER_12` = std::to_string (PCAN_USBBUS12)
const std::string	`BUS_NUMBER_13` = std::to_string (PCAN_USBBUS13)
const std::string	`BUS_NUMBER_14` = std::to_string (PCAN_USBBUS14)
const std::string	`BUS_NUMBER_15` = std::to_string (PCAN_USBBUS15)
const std::string	`BUS_NUMBER_16` = std::to_string (PCAN_USBBUS16)

8.15 DeviceHandle

This class represents a handle for controlling a device on a bus and has the following public member functions.

DeviceHandle ()

```
DeviceHandle (uint32_t handle)
```

equals ()

Compares itself to a given device handle.

```
bool equals (DeviceHandle const other) const (uint32_t handle)
```

toString ()

Returns a string representation of the device handle.

```
std::string toString () const
```

get ()

Returns the device handle.

```
uint32_t get () const
```

8.16 DeviceId

Use this class (not immutable from creation on) to identify and distinguish devices on a bus:

- Hardware adapter identifier
- Device identifier
- Description

The meaning of device ID / description values depends on the bus. For example, a CAN bus may use the integer ID.

DeviceId () [1/3]

Constructs a new device ID object.

```
nlc::DeviceId::DeviceId (BusHardwareId const & busHardwareId_, unsigned int
 deviceId_, std::string const & description_)
```

Parameters	*busHardwareId_*	Identifier of the bus.
	*deviceId_*	An index; subject to bus (= CANopen node ID etc.).
	*description_*	A description (may be empty); subject to bus.

DeviceId () [2/3]

Constructs a new device ID object with extended ID options.

```
nlc::DeviceId::DeviceId (BusHardwareId const & busHardwareId, unsigned int
 deviceId_, std::string const & description_ std::vector <uint8_t> const &
 extraId_, std::string const & extraStringId_)
```

Parameters	*busHardwareId_*	Identifier of the bus.
	*deviceId_*	An index; subject to bus (= CANopen node ID etc.).
	*description_*	A description (may be empty); subject to bus.
	*extraId_*	An additional ID (may be empty); meaning depends on bus.
	*extraStringId_*	Additional string ID (may be empty); meaning depends on bus.

DeviceId () [3/3]

Constructs a copy of a device ID object.

```
nlc::DeviceId::DeviceId (DeviceId const &)
```

Parameters	*deviceId_*	Device ID to copy from.

equals ()

Compares new to existing objects.

```
bool nlc::DeviceId::equals (DeviceId const & other) const
```

Returns *boolean*

getBusHardwareId ()

Reads out the bus hardware ID.

```
BusHardwareId nlc::DeviceId::getBusHardwareId () const
```

Returns BusHardwareId

getDescription ()

Reads out the device description (maybe unused).

```
std::string nlc::DeviceId::getDescription () const
```

Returns *string*

getDeviceId ()

Reads out the device ID (maybe unused).

```
unsigned int nlc::DeviceId::getDeviceId () const
```

Returns *unsigned int*

toString ()

Returns the object as a string.

```
std::string nlc::DeviceId::toString () const
```

Returns *string*

getExtraId ()

Reads out the extra ID of the device (may be unused).

```
const std::vector <uint8_t>&getExtraId () const
```

Returns	*vector<uint8_t>*	A vector of the additional extra ID's (may be empty); meaning depends on the bus.

getExtraStringId ()

Reads out the extra string ID of the device (may be unused).

```
std::string getExtraStringId () const
```

Returns	*string*	The additional string ID (may be empty); meaning depends on the bus.

8.17 LogLevelConverter

This class returns your log level as a string.

```
static std::string toString (nlc::LogLevel logLevel)
```

8.18 LogModuleConverter

This class returns your library-specific log modulesetLoggingLevel () as a string.

static std::string toString (nlc::LogModule logModule)

```
static std::string toString (nlc::LogModule logModule)
```

8.19 ObjectDictionary

This class represents an object dictionary of a controller and has the following public member functions:

getDeviceHandle ()

```
virtual ResultDeviceHandle getDeviceHandle () const
```

Returns *ResultDeviceHandle*

getObject ()

```
virtual ResultObjectSubEntry getObject (OdIndex const odIndex)
```

Returns ResultObjectSubEntry

getObjectEntry ()

```
virtual ResultObjectEntry getObjectEntry (uint16_t index)
```

Returns	*ResultObjectEntry*	Informs on an object's properties.

getXmlFileName ()

```
virtual ResultString getXmlFileName () const
```

Returns	*ResultString*	Returns the XML file name as a string.

readNumber ()

```
virtual ResultInt readNumber (OdIndex const odIndex)
```

Returns ResultInt

readNumberArray ()

```
virtual ResultArrayInt readNumberArray (uint16_t const index)
```

Returns ResultArrayInt

readString ()

```
virtual ResultString readString (OdIndex const odIndex)
```

Returns ResultString

readBytes ()

```
virtual ResultArrayByte readBytes (OdIndex const odIndex)
```

Returns ResultArrayByte

writeNumber ()

```
virtual ResultVoid writeNumber (OdIndex const odIndex, const int64_t value)
```

Returns ResultVoid

writeBytes ()

```
virtual ResultVoid writeBytes (OdIndex const OdIndex, std::vector <uint8_t>
 const & data)
```

Returns ResultVoid

Related Links
OdIndex

8.20 ObjectEntry

This class represents an object entry of the object dictionary, has the following static protected attribute and public member functions:

```
static nlc::ObjectSubEntry invalidObject
```

getName ()

Reads out the name of the object as a string.

```
virtual std::string getName () const
```

getPrivate ()

Checks if the object is private.

```
virtual bool getPrivate () const
```

getIndex ()

Reads out the address of the object index.

```
virtual uint16_t getIndex () const
```

getDataType ()

Reads out the data type of the object.

```
virtual nlc::ObjectEntryDataType getDataType () const
```

getObjectCode ()

Reads out the object code:

Null	0x00
Deftype	0x05
Defstruct	0x06
Var	0x07
Array	0x08
Record	0x09

```
virtual nlc::ObjectCode getObjectCode () const
```

getObjectSaveable ()

Checks if the object is saveable and it's category (see product manual for more details):

`APPLICATION`, `COMMUNICATION`, `DRIVE`, `MISC_CONFIG`, `MODBUS_RTU`, `NO`, `TUNING`, `CUSTOMER`, `ETHERNET`, `CANOPEN`, `VERIFY1020`, `UNKNOWN_SAVEABLE_TYPE`

```
virtual nlc::ObjectSaveable getObjectSaveable () const
```

getMaxSubIndex ()

Reads out the number of subindices supported by this object.

```
virtual uint8_t getMaxSubIndex () const
```

getSubEntry ()

```
virtual nlc::ObjectSubEntry & getSubEntry (uint8_t subIndex)
```

See also ObjectSubEntry.

8.21 ObjectSubEntry

This class represents an object sub-entry (subindex) of the object dictionary and has the following public member functions:

getName ()

Reads out the name of the object as a string.

```
virtual std::string getName () const
```

getSubIndex ()

Reads out the address of the subindex.

```
virtual uint8_t getSubIndex () const
```

getDataType ()

Reads out the data type of the object.

```
virtual nlc::ObjectEntryDataType getDataType () const
```

getSdoAccess ()

Checks if the subindex is accessible via SDO:

ReadOnly	1
WriteOnly	2
ReadWrite	3
NoAccess	0

```
virtual nlc::ObjectSdoAccessAttribute getSdoAccess () const
```

getPdoAccess ()

Checks if the subindex is accessible/mappable via PDO:

Tx	1
Rx	2
TxRx	3
No	0

```
virtual nlc::ObjectPdoAccessAttribute getPdoAccess () const
```

getBitLength ()

Checks the subindex length.

```
virtual uint32_t getBitLength () const
```

getDefaultValueAsNumeric ()

Reads out the default value of the subindex for numeric data types.

```
virtual ResultInt getDefaultValueAsNumeric (std::string const & key) const
```

getDefaultValueAsString ()

Reads out the default value of the subindex for string data types.

```
virtual ResultString getDefaultValueAsString (std::string const & key) const
```

getDefaultValues ()

Reads out the default values of the subindex.

```
virtual std::map <std::string, std::string> getDefaultValues () const
```

readNumber ()

Reads out the numeric actual value of the subindex.

```
virtual ResultInt readNumber () const
```

readString ()

Reads out the string actual value of the subindex.

```
virtual ResultString readString () const
```

readBytes ()

Reads out the actual value of the subindex in bytes.

```
virtual ResultArrayByte readBytes () const
```

writeNumber ()

Writes a numeric value in the subindex.

```
virtual ResultVoid writeNumber (const int64_t value) const
```

writeBytes ()

Writes a value in the subindex in bytes.

```
virtual ResultVoid writeBytes (std::vector <uint8_t> const & data) const
```

8.22 OdIndex

Use this class (immutable from creation on) to wrap and locate object directory indices / sub-indices. A device's OD has up to 65535 (0xFFFF) rows and 255 (0xFF) columns; with gaps between the discontinuous rows. See the CANopen standard and your product manual for more detail.

OdIndex ()

Constructs a new OdIndex object.

```
nlc::OdIndex::OdIndex (uint16_t index, uint8_t subIndex)
```

Parameters	*index*	From 0 to 65535 (0xFFFF) incl.
	subindex	From 0 to 255 (0xFF) incl.

getIndex ()

Reads out the index (from 0x0000 to 0xFFFF).

```
uint16_t nlc::OdIndex::getIndex () const
```

Returns	*uint16_t*

getSubindex ()

Reads out the sub-index (from 0x00 to 0xFF)

```
uint8_t nlc::OdIndex::getSubIndex () const
```

Returns *uint8_t*

toString ()

Returns the index and subindex as a string. The string default *0xIIII:0xSS* reads as follows:

- I = index from 0x0000 to 0xFFFF
- S = sub-index from 0x00 to 0xFF

```
std::string nlc::OdIndex::toString () const
```

Returns *0xIIII:0xSS* Default string representation

8.23 OdLibrary

Use this programming interface to create instances of the *ObjectDictionary* class from XML. By *assignObjectDictionary,* you can then bind each instance to a specific device due to a uniquely created identifier. *ObjectDictionary* instances thus created are stored in the *OdLibrary* object to be accessed by index. The *ODLibrary* class loads ObjectDictionary items from file or array, stores them, and has the following public member functions:

getObjectDictionaryCount ()

```
virtual uint32_t getObjectDictionaryCount () const
```

getObjectDictionary ()

```
virtual ResultObjectDictionary getObjectDictionary (uint32_t odIndex)
```

Returns ResultObjectDictionary

addObjectDictionaryFromFile ()

```
virtual ResultObjectDictionary addObjectDictionaryFromFile (std::string const
 & absoluteXmlFilePath)
```

Returns ResultObjectDictionary

addObjectDictionary ()

```
virtual ResultObjectDictionary addObjectDictionary (std::vector <uint8_t>
 const & odXmlData, const std::string &xmlFilePath = std::string ())
```

Returns ResultObjectDictionary

8.24 OdTypesHelper

In addition to the following public member functions, this class contains custom data types. **Note:** To check your custom data types, look for the `enum class ObjectEntryDataType` in *od_types.hpp*.

uintToObjectCode ()

Converts unsigned integers to object code:

Null	0x00
Deftype	0x05

Defstruct	0x06
Var	0x07
Array	0x08
Record	0x09

```
static ObjectCode uintToObjectCode (unsigned int objectCode)
```

isNumericDataType ()

Informs if a data type is numeric or not.

```
static bool isNumericDataType (ObjectEntryDataType dataType)
```

isDefstructIndex ()

Informs if an object is a definition structure index or not.

```
static bool isDefstructIndex (uint16_t typeNum)
```

isDeftypeIndex ()

Informs if an object is a definition type index or not.

```
static bool isDeftypeIndex (uint16_t typeNum)
```

isComplexDataType ()

Informs if a data type is complex or not.

```
static bool isComplexDataType (ObjectEntryDataType dataType)
```

uintToObjectEntryDataType ()

Converts unsigned integers to OD data type.

```
sstatic ObjectEntryDataType uintToObjectEntryDataType (uint16_t
 objectDataType)
```

objectEntryDataTypeToString ()

Converts OD data type to string.

```
static std::string objectEntryDataTypeToString (ObjectEntryDataType
 odDataType)
```

stringToObjectEntryDatatype ()

Converts string to OD data type if possible. Otherwise, returns UNKNOWN_DATATYPE.

```
static ObjectEntryDataType stringToObjectEntryDatatype (std::string
 dataTypeString)
```

objectEntryDataTypeBitLength ()

Informs on bit length of an object entry data type.

```
static uint32_t objectEntryDataTypeBitLength (ObjectEntryDataType const &
 dataType)
```

8.25 RESTfulBus struct

This struct contains the communication configuration options for the RESTful interface (over Ethernet). It contains the following public attributes:

const std::string	`CONNECT_TIMEOUT_OPTION_NAME` = "RESTful Connect Timeout"
const unsigned long	`DEFAULT_CONNECT_TIMEOUT` = 200
const std::string	`REQUEST_TIMEOUT_OPTION_NAME` = "RESTful Request Timeout"
const unsigned long	`DEFAULT_REQUEST_TIMEOUT` = 200
const std::string	`RESPONSE_TIMEOUT_OPTION_NAME` = "RESTful Response Timeout"
const unsigned long	`DEFAULT_RESPONSE_TIMEOUT` = 750

8.26 ProfinetDCP

Under **Linux,** the calling application needs `CAP_NET_ADMIN` and `CAP_NET_RAW` capabilities. To enable: *sudo setcap 'cap_net_admin,cap_net_raw+eip' ./executable*. In **Windows,** the ProfinetDCP interface uses *WinPcap* (tested with version 4.1.3) or *Npcap* (tested with versions 1.60 and 1.30). It thus searches the dynamically loaded *wpcap.dll* library in the following order (**Note:** no current *Win10Pcap* support):

1. *Nanolib.dll* directory
2. Windows system directory *SystemRoot%\System32*
3. Npcap installation directory *SystemRoot%\System32\Npcap*
4. Environment path

This class represents a Profinet DCP interface and has the following public member functions:

getScanTimeout ()

Informs on a device scan timeout (default = 2000 ms).

```
virtual uint32_t nlc::ProfinetDCP::getScanTimeout () const
```

setScanTimeout ()

Sets a device scan timeout (default = 2000 ms).

```
virtual void nlc::setScanTimeout (uint32_t timeoutMsec)
```

getResponseTimeout ()

Informs on a device response timeout for setup, reset and blink operations (default = 1000 ms).

```
virtual uint32_t nlc::ProfinetDCP::getResponseTimeout () const
```

setResponseTimeout ()

Informs on a device response timeout for setup, reset and blink operations (default = 1000 ms).

```
virtual void nlc::ProfinetDCP::setResponseTimeout (uint32_t timeoutMsec)
```

isServiceAvailable ()

Use this function to check Profinet DCP service availability.

- Network adapter validity / availability
- Windows: WinPcap / Npcap availability
- Linux: CAP_NET_ADMIN / CAP_NET_RAW capabilities

```
virtual ResultVoid nlc::ProfinetDCP::isServiceAvailable (const BusHardwareId &
 busHardwareId)
```

Parameters	*BusHardwareId*	Hardware ID of Profinet DCP service to check.
Returns	*true*	Service is available.
	false	Service is unavailable.

scanProfinetDevices ()

Use this function to scan the hardware bus for the presence of Profinet devices.

```
virtual ResultProfinetDevices scanProfinetDevices (const BusHardwareId &
 busHardwareId)
```

Parameters	*BusHardwareId*	Specifies each fieldbus to open.
Returns	ResultProfinetDevices	Hardware is open.

setupProfinetDevice ()

Establishes the following device settings:

- Device name
- IP address
- Network mask
- Default gateway

```
virtual ResultVoid nlc::setupProfinetDevice (const BusHardwareId &
 busHardwareId, const ProfinetDevice struct & profinetDevice, bool
 savePermanent)
```

resetProfinetDevice ()

Stops the device and resets it to factory defaults.

```
virtual ResultVoid nlc::resetProfinetDevice (const BusHardwareId &
 busHardwareId, const ProfinetDevice & profinetDevice)
```

blinkProfinetDevice ()

Commands the Profinet device to start blinking its Profinet LED.

```
virtual ResultVoid nlc::blinkProfinetDevice (const BusHardwareId &
 busHardwareId, const ProfinetDevice &profinetDevice)
```

validateProfinetDeviceIp ()

Use this function to check device's IP address.

```
virtual ResultVoid validateProfinetDeviceIp (const BusHardwareId
 &busHardwareId, const ProfinetDevice & profinetDevice)
```

Parameters	*BusHardwareId*	Specifies the hardware ID to check.
	ProfinetDevice	Specifies the *Profinet* device to validate.

Returns	*ResultVoid*

8.27 ProfinetDevice struct

The Profinet device data have the following public attributes:

std::string	deviceName
std::string	deviceVendor
std::array< uint8_t, 6 >	macAddress
uint32_t	ipAddress
uint32_t	netMask
uint32_t	defaultGateway

The MAC address is provided as array in format `macAddress = {xx, xx, xx, xx, xx, xx};` whereas IP address, network mask and gateway are all interpreted as big endian hex numbers, such as:

IP address: 192.168.0.2	0xC0A80002
Network mask: 255.255.0.0	0xFFFF0000
Gateway: 192.168.0.1	0xC0A80001

8.28 Result classes

Use the "optional" return values of these classes to check if a function call had success or not, and also locate the fail reasons. On success, the *hasError ()* function returns *false*. By *getResult ()*, you can read out the result value as per type (ResultInt etc.). If a call fails, you read out the reason by *getError ()*.

Protected attributes	*string*	errorString
	NlcErrorCode	errorCode
	uint32_t	exErrorCode

Also, this class has the following public member functions:

hasError ()

Reads out a function call's success.

```
bool nlc::Result::hasError () const
```

Returns	*true*	Failed call. Use *getError ()* to read out the value.
	false	Sucessful call. Use *getResult ()* to read out the value.

getError ()

Reads out the reason if a function call fails.

```
const std::string nlc::Result::getError () const
```

Returns	*const string*

result ()

The following functions aid in defining the exact results:

```
Result (std::string const & errorString_)
```

```
Result (NlcErrorCode const & errCode, std::string const & errorString_)
```

```
Result (NlcErrorCode const & errCode, const uint32_t exErrCode, std::string
 const & errorString_)
```

```
Result (Result const & result)
```

getErrorCode ()

Read the NlcErrorCode.

```
NlcErrorCode getErrorCode () const
```

getExErrorCode ()

```
uint32_t getExErrorCode () const
```

8.28.1 ResultVoid

NanoLib sends you an instance of this class if the function returns void. The class inherits the public functions and protected attributes from the result class and has the following public member functions:

ResultVoid ()

The following functions aid in defining the exact void result:

```
ResultVoid (std::string const &errorString_)
```

```
ResultVoid (NlcErrorCode const & errCode, std::string const & errorString_)
```

```
ResultVoid (NlcErrorCode const & errCode, const uint32_t exErrCode, std::
string const & errorString_)
```

```
ResultVoid (Result const & result)
```

8.28.2 ResultInt

NanoLib sends you an instance of this class if the function returns an integer. The class inherits the public functions / protected attributes from the result class and has the following public member functions:

getResult ()

Returns the integer result if a function call had success.

```
int64_t getResult () const
```

Returns *int64_t*

ResultInt ()

The following functions aid in defining the exact integer result:

```
ResultInt (int64_t result_)
```

```
ResultInt (std::string const & errorString_)
```

```
ResultInt (NlcErrorCode const & errCode, std::string const & errorString_)
```

```
ResultInt (NlcErrorCode const & errCode, const uint32_t exErrCode, std::string
 const & errorString_)
```

```
ResultInt (Result const & result)
```

8.28.3 ResultString

NanoLib sends you an instance of this class if the function returns a string. The class inherits the public functions / protected attributes from the result class and has the following public member functions:

getResult ()

Reads out the string result if a function call had success.

```
const std::string nlc::ResultString::getResult () const
```

Returns *const string*

ResultString ()

The following functions aid in defining the exact string result:

```
ResultString (std::string const & message, bool hasError_)
```

```
ResultString (NlcErrorCode const & errCode, std::string const & errorString_)
```

```
ResultString (NlcErrorCode const & errCode, const uint32_t exErrCode, std::
string const & errorString_)
```

```
ResultString (Result const & result)
```

8.28.4 ResultArrayByte

NanoLib sends you an instance of this class if the function returns a byte array. The class inherits the public functions / protected attributes from the result class and has the following public member functions:

getResult ()

Reads out the byte vector if a function call had success.

```
const std::vector <uint8_t> nlc::ResultArrayByte::getResult () const
```

Returns *const vector<uint8_t>*

ResultArrayByte ()

The following functions aid in defining the exact byte array result:

```
ResultArrayByte (std::vector <uint8_t> const & result_)
```

```
ResultArrayByte (std::string const & errorString_)
```

```
ResultArrayByte (NlcErrorCode const & errCode, std::string const & error
String_)
```

```
ResultArrayByte (NlcErrorCode const & errCode, const uint32_t exErrCode, std::
string const & errorString_)
```

```
ResultArrayByte (Result const & result)
```

8.28.5 ResultArrayInt

NanoLib sends you an instance of this class if the function returns an integer array. The class inherits the public functions / protected attributes from the result class and has the following public member functions:

getResult ()

Reads out the integer vector if a function call had success.

```
const std::vector <int64_t> nlc::ResultArrayInt::getResult () const
```

Returns *const vector<uint64_t>*

ResultArrayInt ()

The following functions aid in defining the exact integer array result:

```
ResultArrayInt (std::vector <int64_t> const & result_)
```

```
ResultArrayInt (std::string const & errorString_)
```

```
ResultArrayInt (NlcErrorCode const & errCode, std::string const & error
String_)
```

```
ResultArrayInt (NlcErrorCode const & errCode, const uint32_t exErrCode, std::
string const & errorString_)
```

```
ResultArrayInt (Result const & result)
```

8.28.6 ResultBusHwIds

NanoLib sends you an instance of this class if the function returns a bus hardware ID array. The class inherits the public functions / protected attributes from the result class and has the following public member functions:

getResult ()

Reads out the bus-hardware-ID vector if a function call had success.

```
const std::vector <BusHardwareId> nlc::ResultBusHwIds::getResult () const
```

Parameters *const vector<BusHardwareId>*

ResultBusHwIds ()

The following functions aid in defining the exact bus-hardware-ID-array result:

```
ResultBusHwIds (std::vector <BusHardwareId> const & result_)
```

```
ResultBusHwIds (std::string const & errorString_)
```

```
ResultBusHwIds (NlcErrorCode const & errCode, std::string const &
 errorString_)
```

```
ResultBusHwIds (NlcErrorCode const & errCode, const uint32_t exErrCode,
 std::string const & errorString_)
```

```
ResultBusHwIds (Result const & result)
```

8.28.7 ResultDeviceId

NanoLib sends you an instance of this class if the function returns a device ID. The class inherits the public functions / protected attributes from the result class and has the following public member functions:

getResult ()

Reads out the device ID vector if a function call had success.

```
DeviceId nlc::ResultDeviceId::getResult () const
```

Returns *const vector<DeviceId>*

ResultDeviceId ()

The following functions aid in defining the exact device ID result:

```
ResultDeviceId (DeviceId const & result_)
```

```
ResultDeviceId (std::string const & errorString_)
```

```
ResultDeviceId (NlcErrorCode const & errCode, std::string const &
 errorString_)
```

```
ResultDeviceId (NlcErrorCode const & errCode, const uint32_t exErrCode,
 std::string errorString_)
```

```
ResultDeviceId (Result const & result)
```

8.28.8 ResultDeviceIds

NanoLib sends you an instance of this class if the function returns a device ID array. The class inherits the public functions / protected attributes from the result class and has the following public member functions:

getResult ()

Returns the device ID vector if a function call had success.

```
DeviceId nlc::ResultDeviceIds::getResult () const
```

Returns *const vector<DeviceId>*

ResultDeviceIds ()

The following functions aid in defining the exact device-ID-array result:

```
ResultDeviceIds (std::vector <DeviceId> const & result_)
```

```
ResultDeviceIds (std::string const & errorString_)
```

```
ResultDeviceIds (NlcErrorCode const & errCode, std::string const &
 errorString_)
```

```
ResultDeviceIds (NlcErrorCode const & errCode, const uint32_t exErrCode,
 std::string const & errorString_)
```

```
ResultDeviceIds (Result const & result)
```

8.28.9 ResultDeviceHandle

NanoLib sends you an instance of this class if the function returns the value of a device handle. The class inherits the public functions / protected attributes from the result class and has the following public member functions:

getResult ()

Reads out the device handle if a function call had success.

```
DeviceHandle nlc::ResultDeviceHandle::getResult () const
```

Returns *DeviceHandle*

ResultDeviceHandle ()

The following functions aid in defining the exact device handle result:

```
ResultDeviceHandle (DeviceHandle const & result_)
```

```
ResultDeviceHandle (std::string const & errorString_)
```

```
ResultDeviceHandle (NlcErrorCode const & errCode, std::string const &
 errorString_)
```

```
ResultDeviceHandle (NlcErrorCode const & errCode, const uint32_t exErrCode,
 std::string const & errorString_)
```

```
ResultDeviceHandle (Result const & result)
```

8.28.10 ResultObjectDictionary

NanoLib sends you an instance of this class if the function returns the content of an object dictionary. The class inherits the public functions / protected attributes from the result class and has the following public member functions:

getResult ()

Reads out the device ID vector if a function call had success.

```
const nlc::ObjectDictionary & nlc::ResultObjectDictionary::getResult () const
```

Returns	*const vector<ObjectDictionary>*

ResultObjectDictionary ()

The following functions aid in defining the exact object dictionary result:

```
ResultObjectDictionary (nlc::ObjectDictionary const & result_)
```

```
ResultObjectDictionary (std::string const & errorString_)
```

```
ResultObjectDictionary (NlcErrorCode const & errCode, std::string const &
 errorString_)
```

```
ResultObjectDictionary (NlcErrorCode const & errCode, const uint32_t
 exErrCode, std::string const & errorString_)
```

```
ResultObjectDictionary (Result const & result)
```

8.28.11 ResultConnectionState

NanoLib sends you an instance of this class if the function returns a device-connection-state info. The class inherits the public functions / protected attributes from the result class and has the following public member functions:

getResult ()

Reads out the device handle if a function call had success.

```
DeviceConnectionStateInfo nlc::ResultConnectionState::getResult () const
```

Returns	*DeviceConnectionStateInfo*	Connected / Disconnected / ConnectedBootloader

ResultConnectionState ()

The following functions aid in defining the exact connection state result:

```
ResultConnectionState (DeviceConnectionStateInfo const & result_)
```

```
ResultConnectionState (std::string const & errorString_)
```

```
ResultConnectionState (NlcErrorCode const & errCode, std::string const &
 errorString_)
```

```
ResultConnectionState (NlcErrorCode const & errCode, const uint32_t exErrCode,
 std::string const & errorString_)
```

```
ResultConnectionState (Result const & result)
```

8.28.12 ResultObjectEntry

NanoLib sends you an instance of this class if the function returns an object entry. The class inherits the public functions / protected attributes from the result class and has the following public member functions:

getResult ()

Returns the device ID vector if a function call had success.

```
nlc::ObjectEntry const& nlc::ResultObjectEntry::getResult () const
```

Returns *const ObjectEntry*

ResultObjectEntry ()

The following functions aid in defining the exact object entry result:

```
ResultObjectEntry (nlc::ObjectEntry const & result_)
```

```
ResultObjectEntry (std::string const & errorString_)
```

```
ResultObjectEntry (NlcErrorCode const & errCode, std::string const &
 errorString_)
```

```
ResultObjectEntry (NlcErrorCode const & errCode, const uint32_t exErrCode,
 std::string const & errorString_)
```

```
ResultObjectEntry (Result const & result)
```

8.28.13 ResultObjectSubEntry

NanoLib sends you an instance of this class if the function returns an object sub-entry. The class inherits the public functions / protected attributes from the result class and has the following public member functions:

getResult ()

Returns the device ID vector if a function call had success.

```
nlc::ObjectSubEntry const & nlc::ResultObjectSubEntry::getResult () const
```

Returns *const ObjectSubEntry*

ResultObjectSubEntry ()

The following functions aid in defining the exact object sub-entry result:

```
ResultObjectSubEntry (nlc::ObjectEntry const & result_)
```

```
ResultObjectSubEntry (std::string const & errorString_)
```

```
ResultObjectSubEntry (NlcErrorCode const & errCode, std::string const &
 errorString_)
```

```
ResultObjectSubEntry (NlcErrorCode const & errCode, const uint32_t exErrCode,
 std::string const & errorString_)
```

```
ResultObjectSubEntry (Result const & result)
```

8.28.14 ResultProfinetDevices

NanoLib sends you an instance of this class if the function returns a Profinet device. The class inherits the public functions / protected attributes from the result class and has the following public member functions:

getResult ()

Reads out the Profinet device vector if a function call had success.

```
const std::vector <ProfinetDevice> & getResult () const
```

ResultProfinetDevices ()

The following functions aid in defining the exact Profinet devices.

```
ResultProfinetDevices (const std::vector <ProfinetDevice> & profinetDevices)
```

```
ResultProfinetDevices (const Result & result)
```

```
ResultProfinetDevices (const std::string &errorText, NlcErrorCode errorCode =
 NlcErrorCode::GeneralError, uint32_t extendedErrorCode = 0)
```

8.28.15 ResultSampleDataArray

NanoLib sends you an instance of this class if the function returns a sample data array. The class inherits the public functions / protected attributes from the result class and has the following public member functions:

getResult ()

Reads out the data array if a function call had success.

```
const std::vector <SampleData> & getResult () const
```

ResultSampleDataArray ()

The following functions aid in defining the exact Profinet devices.

```
ResultSampleDataArray (const std::vector <SampleData> & dataArray)
```

```
ResultSampleDataArray (const std::string &errorDesc, const NlcErrorCode
 errorCode = NlcErrorCode::GeneralError, const uint32_t extendedErrorCode = 0)
```

```
ResultSampleDataArray (const ResultSampleDataArray & other)
```

```
ResultSampleDataArray (const Result & result)
```

8.28.16 ResultSamplerState

NanoLib sends you an instance of this class if the function returns a sampler state.This class inherits the public functions / protected attributes from the result class and has the following public member functions:

getResult ()

Reads out the sampler state vector if a function call had success.

```
SamplerState getResult () const
```

Returns	*SamplerState>*	Unconfigured / Configured / Ready / Running / Completed / Failed / Cancelled

ResultSamplerState ()

The following functions aid in defining the exact sampler state.

```
ResultSamplerState (const SamplerState state)
```

```
ResultSamplerState (const std::string & errorDesc,
const NlcErrorCode errorCode = NlcErrorCode::GeneralError, const uint32_t
 extendedErrorCode = 0)
```

```
ResultSamplerState (const ResultSamplerState & other)
```

```
ResultSamplerState (const Result & result)
```

8.29 NlcErrorCode

If something goes wrong, the result classes report one of the error codes listed in this enumeration.

Error code	C: Category | D: Description | R: Reason
Success	**C:** None. **D:** No error. **R:** The operation completed successfully.
GeneralError	**C:** Unspecified. **D:** Unspecified error. **R:** Failure that fits no other category.
BusUnavailable	**C:** Bus. **D:** Hardware bus not available. **R:** Bus inexistent, cut-off or defect.
CommunicationError	**C:** Communication. **D:** Communication unreliable. **R:** Unexpected data, wrong CRC, frame or parity errors, etc.
ProtocolError	**C:** Protocol. **D:** Protocol error. **R:** Response after unsupported protocol option, device report unsupported protocol, error in the protocol (say, SDO segment sync bit), etc. **R:** A response or device report to unsupported protocol (options) or to errors in protocol (say, SDO segment sync bit), etc. **R:** Unsupported protocol (options) or error in protocol (say, SDO segment sync bit), etc.
ODDoesNotExist	**C:** Object dictionary. **D:** OD address inexistent. **R:** No such address in the object dictionary.
ODInvalidAccess	**C:** Object dictionary. **D:** Access to OD address invalid. **R:** Attempt to write a read-only, or to read from a write-only, address.
ODTypeMismatch	**C:** Object dictionary. **D:** Type mismatch. **R:** Value unconverted to specified type, say, in an attempt to treat a string as a number.
OperationAborted	**C:** Application. **D:** Process aborted. **R:** Process cut by application request. Returns only on operation interrupt by callback function, say, from bus-scanning.
OperationNotSupported	**C:** Common. **D:** Process unsupported. **R:** No hardware bus / device support.
InvalidOperation	**C:** Common. **D:** Process incorrect in current context, or invalid with current argument. **R:** A reconnect attempt to already connected buses / devices. A disconnect attempt to already disconnected ones. A bootloader operation attempt in firmware mode or vice versa.
InvalidArguments	**C:** Common. **D:** Argument invalid. **R:** Wrong logic or syntax.
AccessDenied	**C:** Common. **D:** Access is denied. **R:** Lack of rights or capabilities to perform the requested operation.
ResourceNotFound	**C:** Common. **D:** Specified item not found. **R:** Hardware bus, protocol, device, OD address on device, or file was not found.
ResourceUnavailable	**C:** Common. **D:** Specified item not found. **R:** busy, inexistent, cut-off or defect.
OutOfMemory	**C:** Common. **D:** Insufficient memory. **R:** Too little memory to process this command.
TimeOutError	**C:** Common. **D:** Process timed out. **R:** Return after time-out expired. Timeout may be a device response time, a time to gain shared or exclusive resource access, or a time to switch the bus / device to a suitable state.

8.30 NlcCallback

This parent class for callbacks has the following public member function:

callback ()

```
virtual ResultVoid callback ()
```

Returns ResultVoid

8.31 NlcDataTransferCallback

Use this callback class for data transfers (firmware update, NanoJ upload etc.).

1. For a firmware upload: Define a "co-class" extending this one with a custom callback method implementation.
2. Use the "co-class's" instances in *NanoLibAccessor.uploadFirmware ()* calls.

The main class itself has the following public member function:

callback ()

```
virtual ResultVoid callback (nlc::DataTransferInfo info, int32_t data)
```

Returns ResultVoid

8.32 NlcScanBusCallback

Use this callback class for bus scanning.

1. Define a "co-class" extending this one with a custom callback method implementation.
2. Use the "co-class's" instances in *NanoLibAccessor.scanDevices ()* calls.

The main class itself has the following public member function.

callback ()

```
virtual ResultVoid callback (nlc::BusScanInfo info, std::vector <DeviceId>
 const & devicesFound, int32_t data)
```

Returns *ResultVoid*

8.33 NlcLoggingCallback

Use this callback class for logging callbacks.

1. Define a class that extends this class with a custom callback method implementation
2. Use a pointer to its instances in order to set a callback by `NanoLibAccessor > setLoggingCallback (...)`.

```
virtual void callback (const std::string & payload_str, const std::string &
 formatted_str, const std::string & logger_name, const unsigned int log_level,
 const std::uint64_t time_since_epoch, const size_t thread_id)
```

8.34 SamplerInterface

Use this class to configure, start and stop the sampler, or to get sampled data and fetch a sampler's status or last error. The class has the following public member functions.

configure ()

Configures a sampler.

```
virtual ResultVoid nlc::SamplerInterface::configure (const DeviceHandle
 deviceHandle, const SamplerConfiguration & samplerConfiguration)
```

Parameters	[in] *deviceHandle*	Specifies what device to configure the sampler for.
	[in] *samplerConfiguration*	Specifies the values of configuration attributes.
Returns	*ResultVoid*	Confirms that a void function has run.

getData ()

Gets the sampled data.

```
virtual ResultSampleDataArray nlc::SamplerInterface::getData (const
 DeviceHandle deviceHandle)
```

Parameters	[in] *deviceHandle*	Specifies what device to get the data for.
Returns	*ResultSampleDataArray*	Delivers the sampled data, which can be an empty array if *samplerNotify* is active on start.

getLastError ()

Gets a sampler's last error.

```
virtual ResultVoid nlc::SamplerInterface::getLastError (const DeviceHandle
 deviceHandle)
```

Returns	*ResultVoid*	Confirms that a void function has run.

getState ()

Gets a sampler's status.

```
virtual ResultSamplerState nlc::SamplerInterface::getState (const DeviceHandle
 deviceHandle)
```

Returns	ResultSamplerState	Delivers the sampler state.

start ()

Starts a sampler.

```
virtual ResultVoid nlc::SamplerInterface::start (const DeviceHandle
 deviceHandle, SamplerNotify* samplerNotify, int64_t applicationData)
```

Parameters	[in] *deviceHandle*	Specifies what device to start the sampler for.
	[in] SamplerNotify	Specifies what optional info to report (can be *nullptr*).
	[in] *applicationData*	Option: Forwards application-related data (a user-defined 8-bit array of value / device ID / index, or a datetime, a variable's / function's pointer, etc.) to *samplerNotify*.
Returns	*ResultVoid*	Confirms that a void function has run.

stop ()

Stops a sampler.

```
virtual ResultVoid nlc::SamplerInterface::stop (const DeviceHandle
 deviceHandle)
```

Parameters	[in] *deviceHandle*	Specifies what device to stop the sampler for.
Returns	*ResultVoid*	Confirms that a void function has run.

8.35 SamplerConfiguration struct

This struct contains the data sampler's configuration options (static or not).

Public attributes

std::vector <OdIndex>	*trackedAddresses*	Up to 12 OD addresses to be sampled.
uint32_t	*version*	A structure's version.
uint32_t	*durationMilliseconds*	Sampling duration in ms, from 1 to 65535
uint16_t	*periodMilliseconds*	Sampling period in ms.
uint16_t	*numberOfSamples*	Samples amount.
uint16_t	*preTriggerNumberOfSamples*	Samples pre-trigger amount.
bool	*usingSoftwareImplementation*	Use software implementation.
bool	*usingNewFWSamplerImplementation*	Use FW implementation for devices with a FW version v24xx or newer.
SamplerMode	*mode*	*Normal, repetitive* or *continuous* sampling.
SamplerTriggerCondition	*triggerCondition*	**Start trigger conditions**: `TC_FALSE` = 0x00 `TC_TRUE` = 0x01 `TC_SET` = 0x10 `TC_CLEAR` = 0x11 `TC_RISING_EDGE` = 0x12 `TC_FALLING_EDGE` = 0x13 `TC_BIT_TOGGLE` = 0x14 `TC_GREATER` = 0x15 `TC_GREATER_OR_EQUAL` = 0x16 `TC_LESS` = 0x17 `TC_LESS_OR_EQUAL` = 0x18 `TC_EQUAL` = 0x19 `TC_NOT_EQUAL` = 0x1A `TC_ONE_EDGE` = 0x1B `TC_MULTI_EDGE` = 0x1C, **OdIndex**, ***triggerValue***
SamplerTrigger	*SamplerTrigger*	A trigger to start a sampler?

Static public attributes

static constexpr size_t	`SAMPLER_CONFIGURATION_VERSION` = 0x01000000
static constexpr size_t	`MAX_TRACKED_ADDRESSES` = 12

8.36 SamplerNotify

Use this class to activate sampler notifications when you start a sampler. The class has the following public member function.

notify ()

Delivers a notification entry.

```
virtual void nlc::SamplerNotify::notify (const ResultVoid & lastError, const
 SamplerState samplerState, const std::vector <SampleData> & sampleDatas,
 int64_t applicationData)
```

Parameters	[in] *lastError*	Reports the last error occurred while sampling.
	[in] *samplerState*	Reports the sampler status at notification time: Unconfigured / Configured / Ready / Running / Completed / Failed / Cancelled.
	[in] *sampleDatas*	Reports the sampled-data array.
	[in] *applicationData*	Reports application-specific data.

8.37 SampleData struct

This struct contains the sampled data.

uin64_t iterationNumber	Starts at 0 and only increases in repetitive mode.
std::vector<SampledValues>	Contains he array of sampled values.

8.38 SampledValue struct

This struct contains the sampled values.

in64_t value	Contains the value of a tracked OD address.
uin64_t CollectTimeMsec	Contains the collection time in milliseconds, relative to the sample beginning.

8.39 SamplerTrigger struct

This struct contains the trigger settings of the sampler.

SamplerTriggerCondition condition	The trigger condition:`TC_FALSE` = 0x00 `TC_TRUE` = 0x01 `TC_SET` = 0x10 `TC_CLEAR` = 0x11 `TC_RISING_EDGE` = 0x12 `TC_FALLING_EDGE` = 0x13 `TC_BIT_TOGGLE` = 0x14 `TC_GREATER` = 0x15 `TC_GREATER_OR_EQUAL` = 0x16 `TC_LESS` = 0x17 `TC_LESS_OR_EQUAL` = 0x18 `TC_EQUAL` = 0x19 `TC_NOT_EQUAL` = 0x1A `TC_ONE_EDGE` = 0x1B `TC_MULTI_EDGE` = 0x1C
OdIndex	The trigger's OdIndex (address).
uin32_t value	Condition value or bit number (starting from bit zero).

8.40 Serial struct

Find here your serial communication options and the following public attributes:

const std::string	`BAUD_RATE_OPTIONS_NAME` = "serial baud rate"
const SerialBaudRate	*baudRate* =SerialBaudRate struct

const std::string	PARITY_OPTIONS_NAME = "serial parity"
const SerialParity	*parity* = SerialParity struct

8.41 SerialBaudRate struct

Find here your serial communication baud rate and the following public attributes:

const std::string	BAUD_RATE_7200 = "7200"
const std::string	BAUD_RATE_9600 = "9600"
const std::string	BAUD_RATE_14400 = "14400"
const std::string	BAUD_RATE_19200 = "19200"
const std::string	BAUD_RATE_38400 = "38400"
const std::string	BAUD_RATE_56000 = "56000"
const std::string	BAUD_RATE_57600 = "57600"
const std::string	BAUD_RATE_115200 = "115200"
const std::string	BAUD_RATE_128000 = "128000"
const std::string	BAUD_RATE_256000 = "256000"

8.42 SerialParity struct

Find here your serial parity options and the following public attributes:

const std::string	NONE = "none"
const std::string	ODD = "odd"
const std::string	EVEN = "even"
const std::string	MARK = "mark"
const std::string	SPACE = "space"

9 Licenses

NanoLib API interface headers and example source code are licensed by Nanotec Electronic GmbH & Co. KG under the Creative Commons Attribution 3.0 Unported License (*CC BY*). Library parts provided in binary format (core and fieldbus communication libraries) are licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License (*CC BY ND*).

Creative Commons

The following human-readable summary won't substitute the license(s) itself. You can find the respective license at creativecommons.org and linked below. You are free to:

CC BY 3.0

- **Share:** See right.
- **Adapt:** Remix, transform, and build upon the material for any purpose, even commercially.

CC BY-ND 4.0

- **Share:** Copy and redistribute the material in any medium or format.

The licensor cannot revoke the above freedoms as long as you obey the following license terms:

CC BY 3.0

- **Attribution:** You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **No additional restrictions:** You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

CC BY-ND 4.0

- **Attribution:** See left. **But:** Provide a link to this other license.
- **No derivatives:** If you remix, transform, or build upon the material, you may not distribute the modified material.
- **No additional restrictions:** See left.

Note: You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

Note: No warranties given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

10 Imprint, contact, versions

Document	+ Added │ > Changed │ # Fixed	Product
1.4.2 2024.12	> Re-work of the provided examples.	1.3.0
1.4.1 2024.10	+ NanoLib Modbus: Added device locking mechanism for Modbus VCP. # NanoLib Core: Fixed connection state check. # NanoLib Code: Corrected bus hardware reference removal.	1.2.1
1.4.0 2024.09	+ NanoLib-CANopen: Support for *Peak* PCAN-USB adapter (IPEH-002021/002022).	1.2.0
1.3.3 2024.07	> NanoLib Core: Changed logging callback interface (LogLevel replaced by LogModule). # NanoLib Logger: Separation between core and modules has been corrected. # Modbus TCP: Fixed firmware update for FW4. # EtherCAT: Fixed NanoJ program upload for Core5. # EtherCAT: Fixed firmware update for Core5.	1.1.3
1.3.2 2024.05	# Modbus RTU: Fixed timing issues with low baud rates during firmware update. # RESTful: Fixed NanoJ program upload.	1.1.2
1.3.1 2024.04	# NanoLib Modules Sampler: Correct reading of sampled boolean values.	1.1.1
1.3.0 2024.02	+ Java 11 support for all platforms. + Python 3.11 / 3.12 support for all platforms. + New logging callback interface (see examples). + Callback sinks for NanoLib Logger. > Update logger to version 1.12.0. > NanoLib Modules Sampler: Support now for Nanotec controller firmware v24xx. > NanoLib Modules Sampler: Change in structure used for sampler configuration. > NanoLib Modules Sampler: Continuous mode is synonymous with *endless;* the trigger condition is checked once; the number of samples must be *0.* > NanoLib Modules Sampler: Normal priority for the thread that collects data in firmware mode. > NanoLib Modules Sampler: Rewritten algorithm to detect transition between *Ready & Running state.* # NanoLib Core: No more *Access Violation (0xC0000005)* on closing 2 or more devices using the same bus hardware. # NanoLib Core: No more *Segmentation Fault* on attaching a PEAK adapter under Linux. # NanoLib Modules Sampler: Correct sampled-values reading in firmware mode. # NanoLib Modules Sampler: Correct configuration of 502X:04. # NanoLib Modules Sampler: Correct mixing of buffers with channels. # NanoLib-Canopen: Increased CAN timeouts for robustness and correct scanning at lower baudrates. # NanoLib-Modbus: VCP detection algorithm for special devices (USB-DA-IO).	1.1.0
1.2.2 2022.09	+ EtherCAT support.	1.0.1 (B349)
1.2.1 2022.08	+ Note on VS project settings in Configure your project.	1.0.0 (B344)
1.2.0 2022.08	+ *getDeviceHardwareGroup ()*. + *getProfinetDCP (isServiceAvailable)*. + *getProfinetDCP (validateProfinetDeviceIp)*. + *autoAssignObjectDictionary ()*. + *getXmlFileName ()*. + *const std::string & xmlFilePath* in *addObjectDictionary ()*. + *getSamplerInterface ()*.	1.0.0 (B341)

Document	+ Added | > Changed | # Fixed	Product
	+ *rebootDevice ()*. + Error code *ResourceUnavailable* for *getDeviceBootloaderVersion ()*, *~VendorId ()*, *~HardwareVersion ()*, *~SerialNumber*, and *~Uid*. > *firmwareUploadFromFile* now *uploadFirmwareFromFile ()*. > *firmwareUpload ()* now *uploadFirmware ()*. > *bootloaderUploadFromFile ()* now *uploadBootloaderFromFile ()*. > *bootloaderUpload ()* now *uploadBootloader ()*. > *bootloaderFirmwareUploadFromFile ()* to *uploadBootloaderFirmwareFromFile ()*. > *bootloaderFirmwareUpload ()* now *uploadBootloaderFirmware ()*. > *nanojUploadFromFile ()* now *uploadNanoJFromFile ()*. > *nanojUpload ()* now *uploadNanoJ ()*. > *objectDictionaryLibrary ()* now *getObjectDictionaryLibrary ()*. > *String_String_Map* now *StringStringMap*. > NanoLib-Common: faster execution of *listAvailableBusHardware* and *openBusHardwareWithProtocol* with Ixxat adapter. > NanoLib-CANopen: default settings used (*1000k* baudrate, Ixxat bus number *0*) if bus hardware options empty. > NanoLib-RESTful: admin permission obsolete for communication with Ethernet bootloaders under Windows if *npcap* / *winpcap* driver is available. # NanoLib-CANopen: bus hardware now opens crashless with empty options. # NanoLib-Common: *openBusHardwareWithProtocol ()* with no memory leak now.	
1.1.2 2022.03	+ Linux ARM64 support. + USB mass storage / REST / Profinet DCP support. + *checkConnectionState ()*. + *getDeviceBootloaderVersion ()*. + *ResultProfinetDevices*. + *NlcErrorCode* (replaced *NanotecExceptions*). + NanoLib Modbus: VCP / USB hub unified to USB. > Modbus TCP scanning returns results. < Modbus TCP communication latency remains constant.	0.8.0
1.1.1 2021.11	+ More *ObjectEntryDataType* (complex and profile-specific). + *IOError* return if *connectDevice ()* and *scanDevices ()* find none. + Only 100 ms nominal timeout for CanOpen / Modbus.	0.7.1
1.1.0 2021.06	+ Modbus support (plus USB Hub via VCP). + Chapter *Creating your own Linux project*. + *extraHardwareSpecifier* to *BusHardwareId ()*. + *extraId_* and *extraStringId_* to *DeviceId ()*.	0.7.0
1.0.1 2021.06	+ *setBusState ()*. + *getDeviceBootloaderBuildId ()*. + *getDeviceFirmwareBuildId ()*. + *getDeviceHardwareVersion ()*. # Bugfixes.	0.5.1
1.0.0 2021.05	Edition.	0.5.1