User Manual NanoLib

C++

Valid with NanoLib version 0.7.1

User Manual Version: 1.1.1

Contents

1 Document aim and typography

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. Before using the product, please note the font styles and typefaces that encode this document.

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) an expression in a *foreign language*.

- Example 1: Select *File* > *New* > *Blank Document*.
- Example 2: Open the *Tool* tab and select *Comment*.
- Example 3: In principle, this document distinguishes between:
  - User (= *Nutzer; usuario; utente* [pt.]; *utilisateur; utente* [it.] etc.).
  - 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 our **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, you need to prepare your PC and inform yourself about the intended use and the library limitations.

2.1 System and hardware requirements

NanoLib is executable only under 64-bit operating systems. It supports all Nanotec products with CANopen, Modbus RTU (including USB via virtual comport) and Modbus TCP. **Note:** Follow valid OEM instructions to set the latency to the minimun value possible if you encounter problems when using an FTDI-based USB adapter.

NanoLib	64-bit OS requirements	Fieldbus adapters / cables
v 0.7.1	■ **Windows 10:** with Visual Studio; C++ extensions ■ **Linux:** Ubuntu 18.04.2 LTS with Python 3.7 or later	■ **CANopen:** □ IXXAT USB-to-CAN V2 □ Nanotec ZK-USB-CAN-1 ■ **Modbus RTU:** □ Nanotec ZK-USB-RS485-1 or equivalent USB-RS485 adapter □ USB cable via virtual comport (VCP) ■ **Modbus TCP:** □ Ethernet cable according to product datasheet.

2.2 Intended use and audience

NanoLib is a program library for the operation of, and communication with, Nanotec controllers. NanoLib is intended to be used as a software component in a wide range of industrial applications where Nanotec controllers are installed.

The underlying operating system and the used hardware (PC) on which NanoLib is intended to run do not provide real-time capability. NanoLib can therefore not be used for applications that require synchronous multi-axis movement or are generally time-sensitive.

Under no circumstances may this Nanotec product be integrated as a safety component in a product or system. All products containing a component manufactured by Nanotec must, upon delivery to the end user, be provided with corresponding warning notices and instructions for safe use and safe operation. All warning notices provided by Nanotec must be passed on directly to the end user.

NanoLib solely and exclusively addresses duly skilled programmers in industrial application scenarios.

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 the communication via the fieldbus: *nanolibm_canopen.dll*, *nanolibm_modbus.dll*
- Core functions as libraries in binary format: *nanolib.dll, nanolib.lib*
- Example project: *NanolibExample.sln* (Visual Studio project) and *nanolib_example.cpp* (main file)

For scope of warranty, please observe our terms and conditions for either EMEA / APAC or AMERICA, and strictly follow 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 organize freely customizable motor controller / fieldbus functions around a strictly preconfigured core. NanoLib contains the following modules:

User interface (API)	NanoLib core	Communication libraries
Interface and helper classes which ■ grant access to your controller's OD (object dictionary) ■ are based on the NanoLib core functionalities.	Libraries which ■ implement the API functionality ■ interact with bus libraries.	Fieldbus-specific libraries which ■ serve as 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 the hardware (fieldbus adapter).
- Connect to 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 repsonsible 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

The communication libraries provided by NanoLib (*nanolibm_canopen.dll, nanolibm_modbus.dll*) serve as hardware abstraction layer between core and controller. The core loads these libraries at startup time from the designated project folder and uses them to establish communication with the controller via the corresponding protocol.

4 Getting started

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

4.1 Prepare your system

Prepare the PC along your OS.

- In **Windows:** Install the latest Microsoft Visual Studio with C++ extensions.
- Via **Linux Bash:** To install *make* and *gcc*, call:

```
sudo apt install build-essentials
```

4.2 Install the adapter driver for Windows

Only after due driver installation, you may use the IXXAT USB-to-CAN V2 adapter. **Note:** All other supported adapters do not require a driver installation Refer to the product manual of USB drives, to find out how to activate the virtual comport (VCP).

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

4.3 Install the adapter driver for Linux

Only after due driver installation, you may use the IXXAT USB-to-CAN V2 adapter. **Note:** For the other supported adapters you just need to provide the necessary permissions with the command: `sudo chmod +777 /dev/ttyACM*` (* is the device number). Refer to the product manual of USB drives, to find out how to activate the virtual comport (VCP) if necessary.

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
```

4.4 Connect your hardware

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

1. Connect your adapter to the controller using a suitable cable.
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 communication settings of the Nanotec controller according to the instructions in the product manual.

4.5 Load NanoLib

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

1. According to your region and needs: Download NanoLib from our website for either EMEA / APAC or AMERICA.
2. Unzip all files and folders from the NanoLib download package.

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 *NanolibExample.sln* consists of:

- *nanolib_example.cpp* (main file)
- *nanolib_helper.hpp* and *.cpp* (helper class for wrapping the NanoLib accessor)

In Windows with Visual Studio

1. Open the *NanolibExample.sln* file.
2. Open the *nanolib_example.cpp*(main file).
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 "src/nanolib_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. In the folder *bin* there will be an executable file *example*. In the bash navigate 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 it leads to the error

   ```
   "./example: error while loading shared libraries: libnanolib.so: cannot
    open shared object file: No such file or directory"
   ```

   the installation of the shared objects was not successful, In this case, follow the next steps.
4. Create a new folder within /usr/local/lib. Administrator privileges are neccessary. Therefore type in the bash

   ```
   sudo mkdir /usr/local/lib/nanotec
   ```
5. Copy all shared objects from the *lib* folder of the zip file

   ```
   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 demonstrates the typical workflow for working with 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. Read/write from/to the object dictionary of the controller (examples provided in `objectDictionaryAccessExamples()`, line 10).
6. Close the connection to the device.
7. Close the connection 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 your NanoLib project.

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)inc;%(AdditionalIncludeDirectories)
```

7. Select *Linker* and go to *Additional Library Directories.*
8. Insert:

```
$(ProjectDir)lib;%(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. Find all shared objects in the *lib* folder of your unzipped NanoLib installation package.
2. 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. Open the unzipped installation package.
4. Copy all shared objects from the lib folder via:

```
install ./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>/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 the classes of NanoLib's User Interface 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 ().
10. Familiarize yourself with the class's following public member functions:

listAvailableBusHardware ()

Use this function to list the available fieldbus hardware.

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

Returns	*ResultBusHwIds*	Delivers a fieldbus ID array.

openBusHardwareWithProtocol ()

Use this function to establish connection with a bus hardware.

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

Parameters	*busHwId*	Specifies the fieldbus to open.
	busHwOpt	Specifies fieldbus opening options.
Returns	*ResultVoid*	Confirms the execution of a void function.

getProtocolSpecificAccessor ()

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

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

Parameters	*busHwId*	Specifies the fieldbus to get the accessor for.
Returns	*ResultVoid*	Confirms the execution of a void function.

setBusState ()

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

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

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

scanDevices ()

Use this function to scan for devices in the network.

```
virtual ResultDeviceIds nlc::NanoLibAccessor::scanDevices (BusHardwareId const
 & 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 the NanoLib internal device list and 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 establish connection with a device using *deviceHandle*.

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should connect to.
Returns	*ResultVoid*	Confirms the execution of a void function.
	IOError	Informs that a device is not found.

getDeviceName ()

Use this function to get the device name using *deviceHandle*.

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should get the name for.

Returns	*ResultString*	Delivers a device name as a string.

getDeviceProductCode ()

Use this function to get the device product code using *deviceHandle*.

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should get the product code for.
Returns	*ResultInt*	Delivers the product code as an integer.

getDeviceVendorId ()

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

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should get the vendor id for.
Returns	*ResultInt*	Delivers the product code as an integer.

getDeviceId ()

Use this function to get the device ID of a specific device from the NanoLib internal list.

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should get the device ID for.
Returns	*ResultDeviceId*	Delivers a device ID.

getDeviceIds ()

Use this function to get the device ID of all devices from the NanoLib internal list.

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

Returns	*ResultDeviceIds*	Delivers a device ID list.

getDeviceUid ()

Use this function to get the device ID of a specific device from the NanoLib internal list.

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should get the device ID for.
Returns	*ResultDeviceId*	Delivers a device ID.

getDeviceSerialNumber ()

Use this function to get the serial of a device from the NanoLib internal list.

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should get the serial number for.
Returns	*ResultString*	Delivers a device name as a string.

getDeviceBootloaderBuildId ()

Use this function to get a bus device's bootloader build ID via device handle.

```
virtual ResultDeviceId nlc::NanoLibAccessor:: (DeviceHandle const device
Handle)
```

Parameters	*deviceHandle*	Specifies which bus device NanoLib should get the bootloader build ID for.
Returns	*ResultString*	Delivers a device name as a string.

getDeviceFirmwareBuildId ()

Use this function to get a bus device's firmware build ID via device handle.

```
virtual ResultDeviceId nlc::NanoLibAccessor::getDeviceFirmwareBuildId (Device
Handle const deviceHandle)
```

Parameters	*deviceHandle*	Specifies which bus device NanoLib should get the firmware build ID for.
Returns	*ResultString*	Delivers a device name as a string.

getDeviceHardwareVersion ()

Use this function to get a bus device's hardware version via device handle.

```
virtual ResultDeviceId nlc::NanoLibAccessor::getDeviceHardwareVersion (Device
Handle const deviceHandle)
```

Parameters	*deviceHandle*	Specifies which bus device NanoLib should get the hardware version for.
Returns	*ResultString*	Delivers a device name as a string.

getDeviceState ()

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

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should get the state for.
Returns	*ResultString*	Delivers a device name as a string.

setDeviceState ()

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

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should set the state for.
	state	Assigns a bus-specific state as a string value.
Returns	*ResultVoid*	Confirms the execution of a void function.

getConnectionState ()

Use this function to get the connection state of a specific device using *deviceHandle*.

```
virtual ResultConnectionState nlc::NanoLibAccessor::getConnectionState (Device
Handle const deviceHandle)
```

Parameters	*deviceHandle*	Specifies which bus device NanoLib should set the state for.
Returns	*ResultConnectionState*	Delivers a connection state.

assignObjectDictionary ()

Use this function to assign an object dictionary to *deviceHandle*.

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

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

getAssignedObjectDictionary ()

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

```
virtual ResultObjectDictionary nlc::NanoLibAccessor::getAssignedObject
Dictionary (DeviceHandle const deviceHandle)
```

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

objectDictionaryLibrary ()

This function returns a reference to the object dictionary library.

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

Returns	*OdLibrary&*	Shows which object dictionary is assigned to what library.

setLoggingLevel ()

Use this function to set the needed logging level and limit the console output of the library.

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

Parameters	*level*	The following levels are possible:

0 = *Off*	Switches off the logging entirely.
1 = *Trace*	Lowest level, logs everything (expect huge logfiles).
2 = *Debug*	Logs only debug information.
3 = *Info*	Default level.
4 = *Warn*	Message on recoverable problems.
5 = *Error*	Highest level, only for messages followed very likely by a progam exit.

readNumber ()

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

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should read 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::readArray (DeviceHandle const
 deviceHandle, uint16_t const & index)
```

Parameters	*deviceHandle*	Specifies which bus device NanoLib should read from.
	index	Array object index..
Returns	*ResultArrayInt*	Delivers an array of integers.

readBytes ()

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

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should read from.
	odIndex	Specifies the (sub-) index to read from.
Returns	*ResultArrayByte*	Delivers an array of bytes.

readString ()

Use this function to read strings from the object directory.

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should read from.

	odIndex	Specifies the (sub-) index to read from.
Returns	*ResultString*	Delivers a device name as a string.

writeNumber ()

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

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should write to.
	value	The uninterpreted value (can be signed, unsigned, fix16.16).
	odIndex	Specifies the (sub-) index to read from.
	bitLength	Length in bit.
Returns	*ResultVoid*	Confirms the execution of a void function.

writeBytes ()

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

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should write to.
	data	Byte vector / array.
	odIndex	Specifies the (sub-) index to read from.
Returns	*ResultVoid*	Confirms the execution of a void function.

firmwareUpload ()

Use this function to update your controller firmware.

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should update.
	fwData	Array containing firmware data.
	NlcDataTransferCallback	A data progress tracer.
Returns	*ResultVoid*	Confirms the execution of a void function.

firmwareUploadFromFile ()

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

```
virtual ResultVoid nlc::NanoLibAccessor:::firmwareUploadFromFile (Device
Handle const deviceHandle, std::string const & absoluteFilePath, NlcData
TransferCallback* callback)
```

Parameters	*deviceHandle*	Specifies which bus device NanoLib should update.
	absoluteFilePath	Path to file containing firmware data (std::string).
	NlcDataTransferCallback	A data progress tracer.
Returns	*ResultVoid*	Confirms the execution of a void function.

bootloaderUpload ()

Use this function to update your controller bootloader.

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should update.
	btData	Array containing bootloader data.
	NlcDataTransferCallback	A data progress tracer.
Returns	*ResultVoid*	Confirms the execution of a void function.

bootloaderUploadFromFile ()

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

```
virtual ResultVoid nlc::NanoLibAccessor:::bootloaderUploadFromFile (Device
Handle const deviceHandle, std::string const & bootloaderAbsoluteFilePath, Nlc
DataTransferCallback* callback)
```

Parameters	*deviceHandle*	Specifies which bus device NanoLib should update.
	bootloaderAbsoluteFilePath	Path to file containing bootloader data (std::string)
	NlcDataTransferCallback	A data progress tracer.
Returns	*ResultVoid*	Confirms the execution of a void function.

bootloaderFirmwareUpload ()

Use this function to update your controller bootloader and firmware.

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should update.
	btData	Array containing bootloader data.
	fwData	Array containing firmware data.
	NlcDataTransferCallback	A data progress tracer.
Returns	*ResultVoid*	Confirms the execution of a void function.

bootloaderFirmwareUploadFromFile ()

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

```
virtual ResultVoid nlc::NanoLibAccessor:::bootloaderUploadFromFile (Device
Handle const deviceHandle, std::string const & bootloaderAbsoluteFilePath,
 std::string const & absoluteFilePath, NlcDataTransferCallback* callback)
```

Parameters	*deviceHandle*	Specifies which bus device NanoLib should update.
	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 the execution of a void function.

nanojUpload ()

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

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should upload to.
	vmmData	Array containing NanoJ data.
	NlcDataTransferCallback	A data progress tracer.
Returns	*ResultVoid*	Confirms the execution of a void function.

nanojUploadFromFile ()

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

```
virtual ResultVoid nlc::NanoLibAccessor:::nanojUploadFromFile (DeviceHandle
 const deviceHandle, std::string const & absoluteFilePath, NlcDataTransfer
Callback* callback)
```

Parameters	*deviceHandle*	Specifies which bus device NanoLib should upload to.
	absoluteFilePath	Path to file containing NanoJ data (std::string).
	NlcDataTransferCallback	A data progress tracer.
Returns	*ResultVoid*	Confirms the execution of a void function.

disconnectDevice ()

Use this function to disconnect your device.

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should disconnect from.
Returns	*ResultVoid*	Confirms the execution of a void function.

removeDevice ()

Use this function to remove your device from the internal NanoLib device list.

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

Parameters	*deviceHandle*	Specifies which bus device NanoLib should remove from the list.
Returns	*ResultVoid*	Confirms the execution of a void function.

closeBusHardware ()

Use this function to close the connection to your fieldbus hardware.

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

Parameters	*busHwId*	Specifies the fieldbus to close the connection to.
Returns	*ResultVoid*	Confirms the execution of a void function.

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 ()

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 &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 (USB location info for example)
	*name_*	A friendly name (= *AdapterName (Port)* etc.)

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 (= MAC address etc.).

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

Reads out 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]

Creates a new bus hardware option object.

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

Use the function `addOption (std::string const & key, std::string const & value)` to add key-value pairs.

BusHardwareOptions () [2/2]

Creates 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
	value	Example: BAUD_RATE_1000K

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 ()

Reads out 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 ()

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 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 ()

8.10 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"

8.11 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)
```

Returns *ResultVoid*

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
```

8.12 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. Thus, a CAN bus may use the integer ID.

DeviceId ()

Creates a new device ID object.

```
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 the bus (= CANopen node ID etc.).
	*description_*	A description (maybe empty); subject to the bus.
	*extraId_*	An additional ID (may be empty), meaning is depending on the bus.
	*extraStringId_*	An additional String Id (may be empty), meaning is depending on the bus.

equals ()

Compares new to existing objects.

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

Returns *boolean*

getBusHarwareId ()

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 ()

Reads out the object as a string.

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

Returns	*string*	

getExtraId()

Get 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 *extraIds* (may be empty), meaning is depending on the bus.

getExtraStringId()

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

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

Returns	*string*	The additional *StringId* (may be empty), meaning is depending on the bus.

8.13 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*

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*

8.14 ObjectEntry

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

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

The class has the following public member functions:

getName

Reads out the name of the object.

```
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 (variable, array etc.).

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

getObjectSaveable

Checks if the object is saveable.

```
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.

8.15 ObjectSubEntry

Class representing an object sub-entry (subindex) of the object dictionary and has the following public member functions:

getName

Reads out the name of the subindex.

```
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 subindex.

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

getSdoAccess

Checks if the subindex is accessible via SDO.

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

getPdoAccess

Checks if the subindex is accessible/mappable via PDO.

```
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
```

8.16 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 for further details.

OdIndex ()

Creates 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 ()

Reads out the (sub-) index 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

8.17 OdLibrary

This class represents an object dictionary library and has the following public member functions:

getObjectDictionaryCount

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

getObjectDictionary

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

addObjectDictionaryFromFile

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

addObjectDictionary

```
virtual ResultObjectDictionary addObjectDictionary (std::vector<uint8_t> const
 & odXmlData)
```

8.18 OdTypesHelper

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

uintToObjectCode

Converts unsigned integers to object code.

```
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.

```
static ObjectEntryDataType uintToObjectEntryDataType (unsigned int objectData
Type)
```

objectEntryDataTypeToString

Converts OD data type to string.

```
static std::string objectEntryDataTypeToString (ObjectEntryDataType odData
Type)
```

stringToObjectEntryDatatype

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

```
static ObjectEntryDataType stringToObjectEntryDatatype (std::string dataType
String)
```

objectEntryDataTypeBitLength

Informs on bit length of an object entry data type.

```
static uint32_t objectEntryDataTypeBitLength (ObjectEntryDataType const & data
Type)
```

8.19 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 a success, the *hasError ()* function returns *false*. Via *getResult ()*, you can read out the result value (depending on the result type, e.g., ResultInt). If your call fails, you can read out the reason via *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*	Means: call success. Use *getResult ()* to read out the value.
	false	Means: call failure. Use *getError ()* 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 (NanotecException const & exception)
```

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

getErrorCode () const

```
NlcErrorCode getErrorCode () const
```

getExErrorCode () const

```
uint32_t getExErrorCode () const
```

8.19.1 ResultVoid

NanoLib sends you an instance of this class if the function returns void. This 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 (NanotecException const & exception)
```

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

8.19.2 ResultInt

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

getResult ()

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

```
int64_t nlc::ResultInt::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 (NanotecException const & exception)
```

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

8.19.3 ResultString

NanoLib sends you an instance of this class if the function returns a string. This class inherits the public functions and 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 (NanotecException const & exception)
```

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

8.19.4 ResultArrayByte

NanoLib sends you an instance of this class if the function returns a byte array. This class inherits the public functions and 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 (NanotecException const & exception)
```

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

8.19.5 ResultArrayInt

NanoLib sends you an instance of this class if the function returns an integer array. This class inherits the public functions and 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 (NanotecException const & exception)
```

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

8.19.6 ResultBusHwIds

NanoLib sends you an instance of this class if the function returns a bus-hardware-ID array. This class inherits the public functions and 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 & error
String_)
```

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

```
ResultBusHwIds (NanotecException const & exception)
```

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

8.19.7 ResultDeviceId

NanoLib sends you an instance of this class if the function returns a device ID. This class inherits the public functions and 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 (NlcErrorCode const & errCode, std::string const & error
String_)
```

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

```
ResultDeviceId (NanotecException const & exception)
```

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

8.19.8 ResultDeviceIds

NanoLib sends you an instance of this class if the function returns a device ID array. This class inherits the public functions and 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 & error
String_)
```

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

```
ResultDeviceIds (NanotecException const & exception)
```

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

8.19.9 ResultDeviceHandle

NanoLib sends you an instance of this class if the function returns the monitoring outcome of a device handle. This class inherits the public functions and 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 & error
String_)
```

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

```
ResultDeviceHandle (NanotecException const & exception)
```

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

8.19.10 ResultConnectionState

NanoLib sends you an instance of this class if the function returns a device-connection-state info. This class inherits the public functions and 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 *DeviceHandle*

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 (NanotecException const & exception)
```

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

8.19.11 ResultObjectDictionary

NanoLib sends you an instance of this class if the function returns the monitoring outcome of an object dictionary. This class inherits the public functions and 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<DeviceId>*

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 exErr
Code, std::string const & errorString_)
```

```
ResultObjectDictionary (NanotecException const & exception)
```

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

8.19.12 ResultObjectEntry

NanoLib sends you an instance of this class if the function returns an object entry. This class inherits the public functions and 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 vector<DeviceId>*

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 & error
String_)
```

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

```
ResultObjectEntry (NanotecException const & exception)
```

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

8.19.13 ResultObjectSubEntry

NanoLib sends you an instance of this class if the function returns an object sub-entry. This class inherits the public functions and 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 vector<DeviceId>*

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 & error
String_)
```

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

```
ResultObjectSubEntry (NanotecException const & exception)
```

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

8.20 NlcCallback

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

callback ()

```
virtual ResultVoid callback ()
```

Returns *ResultVoid*

8.21 NlcDataTransferCallback

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

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

The class has the following public member function:

callback ()

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

Returns *ResultVoid*

8.22 NlcScanBusCallback

Use this callback class for bus scanning.

1. Define a class extending this one with a custom callback method implementation.
2. Use the instances of the new class in *NanoLibAccessor.scanDevices ()* calls.

The class has the following public member function.

callback ()

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

Returns *ResultVoid*

8.23 Serial

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 ()
const std::string	**PARITY_OPTIONS_NAME** = "serial parity"
const SerialParity	**parity** = SerialParity ()

8.24 SerialBaudRate

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"

8.25 SerialParity

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"

8.26 NanotecException classes

Check these classes if an operation went wrong due to time-outs or illogical / invalid addresses, arguments, protocols, resources etc. Class functions are:

```
NanotecException(std::string const & msg, NlcErrorCode errCode, uint32_t exErr
Code)
```

```
virtual char const * what () const noexcept
```

```
NlcErrorCode getErrorCode () const
```

```
uint32_t getExErrorCode () const
```

8.26.1 AbortException

If an operation stalls, use this class to override the abort mechanism. The class offers the function `AbortException(std::string const &message, uint32_t abortCode)`. From NanotecException, it also inherits the following public functions:

```
NanotecException(std::string const & msg, NlcErrorCode errCode, uint32_t exErr
Code)
```

```
virtual char const * what () const noexcept
```

```
NlcErrorCode getErrorCode () const
```

```
uint32_t getExErrorCode () const
```

8.26.2 InvalidAddressException

If an object dictionary address is wrong, use this class to address the object correctly. The class offers the function `InvalidAddressException(std::string const & message)`. From NanotecException, it also inherits the following public functions:

```
NanotecException(std::string const & msg, NlcErrorCode errCode, uint32_t exErr
Code)
```

```
virtual char const * what () const noexcept
```

```
NlcErrorCode getErrorCode () const
```

```
uint32_t getExErrorCode () const
```

8.26.3 ProtocolException

Use this class on unexpected messages from the counterpart (= firmware etc.), or on unfullfilled preconditions, to meet the protocol. The class offers the function `ProtocolException(std::string const & message)`. From NanotecException, it also inherits the following public functions:

```
NanotecException(std::string const & msg, NlcErrorCode errCode, uint32_t exErr
Code)
```

```
virtual char const * what () const noexcept
```

```
NlcErrorCode getErrorCode () const
```

```
uint32_t getExErrorCode () const
```

8.26.4 ResourceException

Use this class to fix errors with required local resources, CAN adapters etc. The class offers the function `ResourceException(std::string const & message)`. From NanotecException, it also inherits the following public functions:

```
NanotecException(std::string const & msg, NlcErrorCode errCode, uint32_t exErr
Code)
```

```
virtual char const * what () const noexcept
```

```
NlcErrorCode getErrorCode () const
```

```
uint32_t getExErrorCode () const
```

8.26.5 TimeoutException

If a certain period exceeds (say, a 100 ms time-out for CANopen / Modbus), you can use this class to change or end the time-out. The class offers the function `TimeoutException(std::string const & message)`. From NanotecException, it also inherits the following public functions:

```
NanotecException(std::string const & msg, NlcErrorCode errCode, uint32_t exErr
Code)
```

```
virtual char const * what () const noexcept
```

```
NlcErrorCode getErrorCode () const
```

```
uint32_t getExErrorCode () const
```

9 Licenses

The NanoLib interface headers (*API*) and the example source code provided are licensed by Nanotec Electronic GmbH & Co. KG under the Creative Commons Attribution 3.0 Unported License (*CC BY*). The parts of the library 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 does not 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, document history

Nanotec Electronic GmbH & Co. KG │ Kapellenstraße 6 │ 85622 Feldkirchen │ Germany

Tel. +49 (0)89 900 686-0 │ Fax +49 (0)89 900 686-50 │ info@nanotec.de │ www.nanotec.com

Document	Changes	Product release
1.0.0 (05/2021)	Edition	0.5.1
1.0.1 (06/2021)	Bugfixes and (added): ■ setBusState () ■ getDeviceBootloaderBuildId () ■ getDeviceFirmwareBuildId () ■ getDeviceHardwareVersion ()	0.5.1
1.1.0 (06/2021)	■ Added Modbus support including USB Hub via VCP and (also added): ■ Chapter Creating your own Linux project ■ *extraHardwareSpecifier* to BusHardwareId () ■ *extraId_* and *extraStringId_* to DeviceId ()	0.7.0
1.1.1 (11/2021)	■ 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