

User Manual NanoLib

Python

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.....	6
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 adapter driver for Windows.....	8
4.3 Install the adapter driver for Linux.....	8
4.4 Connect your hardware.....	9
4.5 Load NanoLib.....	9
5 Windows Setup.....	10
6 Linux Setup.....	12
7 Classes / functions reference.....	14
7.1 NanoLibAccessor.....	14
7.2 BusHardwareId.....	24
7.3 BusHardwareOptions.....	25
7.4 BusHwOptionsDefault.....	26
7.5 CanBaudRate.....	26
7.6 CanBus.....	26
7.7 CanOpenNmtService.....	26
7.8 CanOpenNmtState.....	26
7.9 EtherCATBus Struct.....	27
7.10 EtherCATState Struct.....	27
7.11 Ixxat.....	28
7.12 IxxatAdapterBusNumber.....	28
7.13 DeviceHandle.....	28
7.14 DeviceId.....	28
7.15 ObjectDictionary.....	29
7.16 ObjectEntry.....	31
7.17 ObjectSubEntry.....	32
7.18 OdIndex.....	33
7.19 OdLibrary.....	34
7.20 OdTypesHelper.....	34
7.21 RESTfulBus struct.....	35
7.22 ProfinetDCP.....	36

7.23	ProfinetDevice.....	37
7.24	Result classes.....	37
7.24.1	ResultVoid.....	38
7.24.2	ResultInt.....	38
7.24.3	ResultString.....	38
7.24.4	ResultArrayByte.....	38
7.24.5	ResultArrayInt.....	38
7.24.6	ResultBusHwIds.....	39
7.24.7	ResultDeviceId.....	39
7.24.8	ResultDeviceIds.....	39
7.24.9	ResultDeviceHandle.....	39
7.24.10	ResultConnectionState.....	39
7.24.11	ResultObjectDictionary.....	40
7.24.12	ResultObjectEntry.....	40
7.24.13	ResultObjectSubEntry.....	40
7.25	NlcErrorCode.....	40
7.26	NlcCallback.....	41
7.27	NlcDataTransferCallback.....	41
7.28	NlcScanBusCallback.....	42
7.29	SamplerInterface.....	42
7.30	SamplerConfiguration.....	43
7.31	SamplerNotify.....	44
7.32	Serial.....	44
7.33	SerialBaudRate.....	44
7.34	SerialParity.....	45
8	Licenses.....	46
9	Imprint, contact, versions.....	47

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. Before product use, please observe the document's typefaces and conventions.

Underlined text marks a cross reference or hyperlink.

- Example 1: For exact instructions on the NanoLibAccessor, see Setup.
- Example 2: Install the lxxat 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*. 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 setLogLevel(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, you need to 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!

- ▶ Use, and consistently maintain, a 64-bit system.
- ▶ Follow valid OEM instructions.

NanoLib is executable only under 64-bit operating systems. It supports all Nanotec products with CANopen, Modbus RTU (including USB via virtual comport), Modbus TCP. Version 0.8.0 and higher also supports USB mass storage and Ethernet (via REST). Version 1.0.0 and higher adds EtherCAT support. **Note:** Follow valid OEM instructions to set the latency to the minimum possible value if you encounter problems when using an FTDI-based USB adapter.

Version	Requirements (64-bit system mandatory)	Fieldbus adapters / cables
0.7.1	<ul style="list-style-type: none"> ■ Windows 10: <i>Visual Studio</i> w/ <i>C++</i> redistributable; <i>Python 3.7 to 3.9</i> ■ Linux: <i>Ubuntu 18.04.2 LTS</i> w/ <i>Python 3.7 to 3.9</i> 	<ul style="list-style-type: none"> ■ CANopen: <i>IXXAT USB-to-CAN V2; Nanotec ZK-USB-CAN-1</i> ■ Modbus RTU: <i>Nanotec ZK-USB-RS485-1 or equivalent USB-RS485 adapter; USB cable via virtual comport (VCP)</i> ■ Modbus TCP: <i>Ethernet cable according to product datasheet</i>
0.8.0	<ul style="list-style-type: none"> ■ Windows 10: <i>Python 3.10</i> added ■ Linux: <i>ARM64</i> added w/ <i>Python 3.7 to 3.10</i> 	<ul style="list-style-type: none"> ■ VCP / USB hub: <i>now uniform USB</i> ■ USB mass storage: <i>USB cable</i> ■ REST: <i>Ethernet cable</i>
1.0.0	Windows 10 w/ <i>Visual Studio</i> <ul style="list-style-type: none"> ■ CANopen: <i>Ixxat VCI</i> driver (optional) ■ EtherCat module / Profinet DCP: <i>Npcap</i> or <i>WinPcap</i> ■ RESTful module: <i>Npcap</i>, <i>WinPcap</i>, or admin permissions to communicate w/ Ethernet bootloaders 	<ul style="list-style-type: none"> ■ EtherCAT: <i>Ethernet cable</i>
1.0.0	Linux w/ <i>Ubuntu</i> <ul style="list-style-type: none"> ■ Profinet DCP: <i>CAP_NET_ADMIN</i> and <i>CAP_NET_RAW</i> capabilities ■ CANopen: <i>Ixxat ECI</i> driver ■ EtherCat: <i>CAP_NET_ADMIN</i>, <i>CAP_NET_RAW</i> and <i>CAP_SYS_NICE</i> capabilities ■ RESTful: <i>CAP_NET_ADMIN</i> capability to communicate w/ Ethernet bootloaders (also recommended: <i>CAP_NET_RAW</i>) 	<ul style="list-style-type: none"> ■ EtherCAT: <i>Ethernet cable</i>

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.

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

In no case may you integrate this Nanotec product 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 on all Nanotec-issued warning notices straight 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 classes as source code (API)
- Libraries that facilitate communication by fieldbus: *nanolibm_canopen.dll*, *nanolibm_modbus.dll* etc.
- Core functions as library in binary format: *_nanolib_python.pyd*
- Example code: *nanolib_example.py* and *nanolib_helper.py*

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 arrange freely customizable motor controller / fieldbus functions around a strictly pre-built core. NanoLib contains the following modules:

User interface (API)	NanoLib core	Communication libraries
Interface and helper classes which	Libraries which	Fieldbus-specific libraries which
<ul style="list-style-type: none"> ■ grant access to your controller's OD (object dictionary) ■ are based on the NanoLib core functionalities. 	<ul style="list-style-type: none"> ■ implement the API functionality ■ interact with bus libraries. 	<ul style="list-style-type: none"> ■ 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 library *nanolib_python.pyd*. 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

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 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 *Ixxat* driver, **do** prepare your PC along the operating system first. To prepare the PC along your Windows OS, install *Python 3.7 to 3.10* from their Website. To install *make* and *gcc* by *Linux Bash*, call `sudo apt install build-essentials`. Then, **do 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 Ixxat driver (optionally *Ixxat ECI*).
3. Optionally, install an Ixxat ECI driver.
4. Connect the driver to the CAN-to-USB adapter.
5. Link all relevant devices to the adapter.
6. Only then, power up the devices.

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:

- [Windows Setup](#).
- [Linux Setup](#).

5 Windows Setup

Prerequisites



NOTICE

In order to avoid name conflict with other products with similar names the Python pip package is called "nano*tec*lib".

In this chapter you will read how to setup *NanoLib* with Python in Windows.

1. Install Python 3.7, 3.8 or 3.9 from www.python.org/. **Note:** This will work only with python 64 bit!
2. Nanotec recommends using a virtual environment before installing nanoteclib, open a CMD and use the following commands to setup a virtual environment:

```
mkdir test_project
cd test_project
python -m venv .env
.env\Scripts\activate.bat
```

→ In case the setup was successful the CMD is prefixed with (.env), e.g. "(.env) C:\test_project>"

3. The package *wheel* is necessary to install *nanoteclib*:

```
pip install wheel
```

Installing the pip package

In order to use the *NanoLib* it needs to be installed within python. This chapter describes the procedure without using a virtual environment.

1. Open a CMD or powershell and navigate to the folder of the zipped pip package.
2. Type

```
pip install [Zip-Filename]
```

into the console and press Enter. A lot of information will be printed out, the last line in case of a success is

```
Successfully installed nanoteclib-win-N.N.N
```

where N.N.N is the version number of the NanoLib.

Check the installation

To check, if the installation has worked, use the following steps:

1. Open a command line or a powershell, if you haven't already done so.
2. Type in

```
python
```

and press Enter to open the python shell. The screen will show something similar to this:

```
Python <>
Type "help", "copyright", "credits" or "license" for more information.
{{>>> }}
```

3. In this python shell type

```
import nanoteclib
```

and press Enter. In case no error occurs, the installation was successful.

4. You can now leave the python shell by typing in

```
exit()
```

and press Enter.

Running the example project

Run the file *nanotec_example.py* on a command line or powershell like this

```
python <PATH_TO_EXAMPLE_FOLDER>\nanotec_example.py
```

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 the code).
6. Close the connection to the device.
7. Close the connection to the adapter.

6 Linux Setup

NOTICE



In order to avoid name conflict with other products with similar names, the Python pip package is called "nano*tec*lib".

In this chapter you will read how to setup *NanoLib* with Python in Linux.

Prerequisites

1. A python 3 installation is required.
2. Nanotec recommends using pip and "virtual environment". Install both with the following bash command:

```
sudo apt install python3-pip python3-venv -y
```

3. We recommend using a virtual environment before installing nanoteclib, use the following commands to setup a virtual environment:

```
mkdir test_project
cd test_project
python3 -m venv .env
source ./env/bin/activate
```

→ In case the setup was successful the bash is prefixed with (.env), e.g. (.env)
username@hostname:~/test_project\$

4. The package *wheel* is necessary to install nanoteclib:

```
pip3 install wheel
```

Installing the pip package

In order to use the NanotecLib, the library needs to be installed within python. This chapter describes the procedure without using a virtual environment.

1. Open a bash, navigate to the project folder and activate the virtual environment.
2. Type

```
pip3 install PATH_TO_NANOTEC_LIB_TAR_GZ/nanoteclib-N.N.N.tar.gz
```

into the console and press Enter. A lot of information will be printed out, the last line in case of a success is

```
Successfully installed nanoteclib-N.N.N
```

where N.N.N is the version number of the NanotecLib.

Check the Installation

To check, if the installation has worked, use the following steps:

1. Open a bash, if you haven't already done so.
2. Type in

```
python3
```

and press Enter to open the python shell. The screen will show something similar to this:

```
Python <>
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>>
```

3. In this python shell type

```
import nanoteclib
```

and press Enter. In case no error occurs, the installation was successful.

4. You can now leave the python shell by typing in

```
exit()
```

and press "Enter".

Running the example project

Run the file "nanotec_example.py" on a bash like this

```
python3 <PATH_TO_EXAMPLE_FOLDER>\nanotec_example.py
```

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 the code).
6. Close the connection to the device.
7. Close the connection to the adapter.

7 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.

Parameters	<i>param_a</i>	Additional comment if needed.
	<i>param_b</i>	
Returns	<i>ResultVoid</i>	Additional comment if needed.

7.1 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 available fieldbus hardware.

```
listAvailableBusHardware (self)
```

Returns	<i>ResultBusHwIds</i>	Delivers a <u>fieldbus ID array</u> .
---------	-----------------------	---------------------------------------

openBusHardwareWithProtocol ()

Use this function to connect bus hardware.

```
openBusHardwareWithProtocol (self, busHwId, busHwOpt)
```

Parameters	<i>busHwId</i>	Specifies the <u>fieldbus</u> to open.
	<i>busHwOpt</i>	Specifies <u>fieldbus opening options</u> .
Returns	<i>ResultVoid</i>	Confirms that a <u>void function</u> has run.

isBusHardwareOpen ()

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

```
isBusHardwareOpen (self, busHardwareId)
```

Parameters	<i>BusHardwareId</i>	Specifies each <u>fieldbus</u> to open.
Returns	<i>true</i>	Hardware is open.
	<i>false</i>	Hardware is closed.

getProtocolSpecificAccessor ()

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

```
getProtocolSpecificAccessor (self, busHwId)
```

Parameters	<i>busHwId</i>	Specifies the <u>fieldbus</u> to get the accessor for.
Returns	<i>ResultVoid</i>	Confirms that a <u>void function</u> has run.

getProfinetDCP (isServiceAvailable)

Use this function to check if *Profinet DCP* is available and if the network adapter is valid / available:

■ Windows: *WinPcap / Npcap* availability ■ Linux: *CAP_NET_ADMIN* and *CAP_NET_RAW* capabilities

```
isServiceAvailable (self, busHardwareId)
```

Parameters	<i>busHwId</i>	Specifies the device to check <i>ProfinetDCP</i> availability for.
Returns	<i>ResultVoid</i>	Confirms that a <u>void function</u> has run.

getProfinetDCP (validateProfinetDeviceIp)

Use this function to validate a *Profinet DCP* device's IP address.

```
validateProfinetDeviceIp (self, busHardwareId, profinetDevice
```

Parameters	<i>busHwId</i>	Specifies the device to check <i>ProfinetDCP</i> availability for.
	<i>ProfinetDevice</i>	Contains the <u>Profinet device</u> data.
Returns	<i>ResultVoid</i>	Confirms that a <u>void function</u> has run.

getSamplerInterface ()

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

```
getSamplerInterface (self)
```

Returns	<i>SamplerInterface</i>	Refers to the <u>sampler interface</u> class.
---------	-------------------------	---

setBusState ()

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

```
setBusState (self, busHwId, state)
```

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

scanDevices ()

Use this function to scan for devices in the network.

```
scanDevices (self, busHwId, callback)
```

Parameters	<i>busHwId</i>	Specifies the <u>fieldbus</u> to scan.
	<i>callback</i>	<u>NlcScanBusCallback</u> progress tracer.

Returns	<i>ResultDeviceIds</i>	Delivers a <u>device ID</u> array.
	<i>IOError</i>	Informes 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.

```
addDevice (self, deviceId)
```

Parameters	<i>deviceId</i>	Specifies the device to add to the list.
Returns	<i>ResultDeviceHandle</i>	Delivers a <u>device handle</u> .

connectDevice ()

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

```
connectDevice (self, deviceHandle)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall connect to.
Returns	<i>ResultVoid</i>	Confirms that a <u>void function</u> has run.
	<i>IOError</i>	Informes that a device is not found.

getDeviceName ()

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

```
getDeviceName (self, deviceHandle)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall get the name for.
Returns	<i>ResultString</i>	Delivers device names as a <u>string</u> .

getDeviceProductCode ()

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

```
getDeviceProductCode (self, deviceHandle)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall get the product code for.
Returns	<i>ResultInt</i>	Delivers product codes as an <u>integer</u> .

getDeviceVendorId ()

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

```
ggetDeviceVendorId (self, deviceHandle)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall get the vendor ID for.
Returns	<i>ResultInt</i>	Delivers vendor ID's as an <u>integer</u> .
	<i>ResourceUnavailable</i>	Informes that <u>no data</u> is found.

getDeviceId ()

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

```
getDeviceId (self)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall get the device ID for.
Returns	<i>ResultDeviceId</i>	Delivers a <u>device ID</u> .

getDeviceIds ()

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

```
getDeviceIds (self)
```

Returns	<i>ResultDeviceIds</i>	Delivers a <u>device ID list</u> .
---------	------------------------	------------------------------------

getDeviceUid ()

Use this function to get a device's unique ID (96 bit / 12 bytes) from the NanoLib internal list.

```
getDeviceUid (self)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall get the unique ID for.
Returns	<i>ResultArrayByte</i>	Delivers unique ID's as a <u>byte array</u> .
	<i>ResourceUnavailable</i>	Informs that <u>no data</u> is found.

getDeviceSerialNumber ()

Use this function to get a device's serial number from the NanoLib internal list.

```
getDeviceSerialNumber (self)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall get the serial number for.
Returns	<i>ResultString</i>	Delivers serial numbers as a <u>string</u> .
	<i>ResourceUnavailable</i>	Informs that <u>no data</u> is found.

getDeviceHardwareGroup ()

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

```
getDeviceHardwareGroup (self, deviceHandle)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall get the hardware group for.
Returns	<i>ResultInt</i>	Delivers hardware groups as an <u>integer</u> .

getDeviceHardwareVersion ()

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

```
getDeviceHardwareVersion (self, deviceHandle)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall get the hardware version for.
Returns	<i>ResultString</i>	Delivers device names as a <u>string</u> .

*ResourceUnavailable*Informs that no data is found.**getDeviceFirmwareBuildId ()**Use this function to get a bus device's firmware build ID by *deviceHandle*.

```
getDeviceFirmwareBuildId (self, deviceHandle)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall get the firmware build ID for.
Returns	<i>ResultString</i>	Delivers device names as a <u>string</u> .

getDeviceBootloaderVersion ()Use this function to get a bus device's bootloader version via *deviceHandle*.

```
getDeviceBootloaderVersion (self, deviceHandle)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall get the bootloader version for.
Returns	<i>ResultInt</i> <i>ResourceUnavailable</i>	Delivers bootloader versions as an <u>integer</u> . Informs that <u>no data</u> is found.

getDeviceBootloaderBuildId ()Use this function to get a bus device's bootloader build ID via *deviceHandle*.

```
getDeviceBootloaderBuildId (self, deviceHandle)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall get the bootloader build ID for.
Returns	<i>ResultString</i>	Delivers device names as a <u>string</u> .

rebootDevice ()Use this function to return a reboot the bus device via *deviceHandle*.

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

Parameters	<i>deviceHandle</i>	Specifies the <u>fieldbus</u> to reboot.
Returns	<i>ResultVoid</i>	Confirms that a <u>void function</u> has run.

getDeviceState ()

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

```
getDeviceState (self, deviceHandle)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall get the state for.
Returns	<i>ResultString</i>	Delivers device names as a <u>string</u> .

setDeviceState ()

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

```
setDeviceState(self, deviceHandle, state):
```

Parameters	<i>deviceHandle</i> <i>state</i>	Specifies what bus device NanoLib shall set the state for. Assigns a bus-specific state as a string value.
Returns	<i>ResultVoid</i>	Confirms that a <u>void function</u> has run.

getConnectionState ()

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

```
getConnectionState (self, deviceHandle)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall get the connection state for.
Returns	<i>ResultConnectionState</i>	Delivers a <u>connection state</u> (= <i>Disconnected</i> , <i>Connected</i> , <i>ConnectedBootloader</i>).

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.

```
checkConnectionState (self, deviceHandle)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall check the connection state for.
Returns	<i>ResultConnectionState</i>	Delivers a <u>connection state</u> (= not <i>Disconnected</i>).

assignObjectDictionary ()

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

```
assignObjectDictionary (self, deviceHandle, objectDictionary)
```

Parameters	<i>deviceHandle</i> <i>objectDictionary</i>	Specifies what bus device NanoLib shall assign the OD to.
Returns	<i>ResultObjectDictionary</i>	Shows the <u>properties of an object dictionary</u> .

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.

```
autoAssignObjectDictionary (self, deviceHandle, dictionariesLocationPath)
```

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

getAssignedObjectDictionary ()

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

```
getAssignedObjectDictionary (self, deviceHandle)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall get the assigned OD for.
Returns	<i>ResultObjectDictionary</i>	Shows the <u>properties</u> of an object dictionary.

getObjectDictionaryLibrary ()

This function returns an OdLibrary reference.

```
getObjectDictionaryLibrary (self)
```

Returns	<i>OdLibrary&</i>	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*.

```
setLoggingLevel (self, level)
```

Parameters	<i>level</i>	The following log detailings are possible:
0 = <i>Off</i>	No logging at all.	
1 = <i>Trace</i>	Lowest level (largest log file); logs any feasible detail, plus software start / stop.	
2 = <i>Debug</i>	Logs debug information (= interim results, content sent or received, etc.)	
3 = <i>Info</i>	Default level; logs informational messages.	
4 = <i>Warn</i>	Logs problems that did occur but won't stop the current algorithm.	
5 = <i>Error</i>	Highest level (smallest log file); logs just severe trouble that did stop the algorithm.	

readNumber ()

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

```
readNumber (self, deviceHandle, odIndex)
```

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

readNumberArray ()

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

```
readNumberArray (self, deviceHandle, index)
```

Parameters	<i>deviceHandle</i> <i>index</i>	Specifies what bus device NanoLib shall read from. Array object index.
Returns	<i>ResultArrayInt</i>	Delivers an <u>integer array</u> .

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)
```

```
readBytes (self, odIndex)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall read from.
	<i>odIndex</i>	Specifies the <u>(sub-) index</u> to read from.
Returns	<i>ResultArrayByte</i>	Delivers a <u>byte array</u> .

readString ()

Use this function to read strings from the object directory.

```
readString (self)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall read from.
	<i>odIndex</i>	Specifies the <u>(sub-) index</u> to read from.
Returns	<i>ResultString</i>	Delivers device names as a <u>string</u> .

writeNumber ()

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

```
writeNumber (self, deviceHandle, value, odIndex, bitLength)
```

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

writeBytes ()

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

```
writeBytes (self, deviceHandle, data, odIndex)
```

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

uploadFirmware ()

Use this function to update your controller firmware.

```
uploadFirmware (self, deviceHandle, fwData, callback)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall update.
	<i>fwData</i>	Array containing firmware data.
	<i>NlcDataTransferCallback</i>	A <u>data progress</u> tracer.
Returns	<i>ResultVoid</i>	Confirms that a <u>void function</u> has run.

uploadFirmwareFromFile ()

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

```
uploadFirmwareFromFile (self, deviceHandle, absoluteFilePath, callback)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall update.
------------	---------------------	---

	<i>absoluteFilePath</i>	Path to file containing firmware data (string).
	<i>NlcDataTransferCallback</i>	A data progress tracer.
Returns	<i>ResultVoid</i>	Confirms that a void function has run.

uploadBootloader ()

Use this function to update your controller bootloader.

```
uploadBootloader (self, deviceHandle, btData, callback)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall update.
	<i>btData</i>	Array containing bootloader data.
	<i>NlcDataTransferCallback</i>	A data progress tracer.
Returns	<i>ResultVoid</i>	Confirms that a void function has run.

uploadBootloaderFromFile ()

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

```
uploadBootloaderFromFile (self, deviceHandle, bootloaderAbsolutePath, callback)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall update.
	<i>bootloaderAbsolutePath</i>	Path to file containing bootloader data (string).
	<i>NlcDataTransferCallback</i>	A data progress tracer.
Returns	<i>ResultVoid</i>	Confirms that a void function has run.

uploadBootloaderFirmware ()

Use this function to update your controller bootloader and firmware.

```
uploadBootloaderFirmware (self, deviceHandle, btData, fwData, callback)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall update.
	<i>btData</i>	Array containing bootloader data.
	<i>fwData</i>	Array containing firmware data.
	<i>NlcDataTransferCallback</i>	A data progress tracer.
Returns	<i>ResultVoid</i>	Confirms that a void function has run.

uploadBootloaderFirmwareFromFile ()

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

```
uploadBootloaderFirmwareFromFile (self, deviceHandle, bootloaderAbsolutePath, absoluteFilePath, callback)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall update.
	<i>bootloaderAbsolutePath</i>	Path to file containing bootloader data (string).
	<i>absoluteFilePath</i>	Path to file containing firmware data (uint8_t).
	<i>NlcDataTransferCallback</i>	A data progress tracer.
Returns	<i>ResultVoid</i>	Confirms that a void function has run.

uploadNanoJ ()

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

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

```
uploadNanoJ (self, deviceHandle, vmmData, callback)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall upload to.
	<i>vmmData</i>	Array containing NanoJ data.
	<i>NlcDataTransferCallback</i>	A data progress tracer.
Returns	<i>ResultVoid</i>	Confirms that a void function has run.

uploadNanoJFromFile ()

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

```
uploadNanoJFromFile (self, deviceHandle, absoluteFilePath, callback)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall upload to.
	<i>absoluteFilePath</i>	Path to file containing NanoJ data (string).
	<i>NlcDataTransferCallback</i>	A data progress tracer.
Returns	<i>ResultVoid</i>	Confirms that a void function has run.

disconnectDevice ()

Use this function to disconnect your device.

```
disconnectDevice (self, deviceHandle)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall disconnect from.
Returns	<i>ResultVoid</i>	Confirms that a void function has run.

removeDevice ()

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

```
removeDevice (self, deviceHandle)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib shall delist.
Returns	<i>ResultVoid</i>	Confirms that a void function has run.

closeBusHardware ()

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

```
closeBusHardware (self, busHwId)
```

Parameters	<i>busHwId</i>	Specifies the fieldbus to close the connection to.
Returns	<i>ResultVoid</i>	Confirms that a void function has run.

7.2 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.)
- Protocol to use (= Modbus TCP, CANopen etc.)
- Bus hardware specifier (= serial port name, MAC address etc.)
- Friendly name

BusHardwareId ()

Creates a new bus hardware ID object.

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

equals ()

Compares a new bus hardware ID to existing ones.

```
equals (self, other)
```

Parameters	<i>other</i>	Another object of the same class.
Returns	<i>true</i>	If both are equal in all values.
	<i>false</i>	If the values differ.

getBusHardware ()

Reads out the bus hardware string.

```
getBusHardware (self)
```

Returns *string*

getHardwareSpecifier ()

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

```
getHardwareSpecifier (self)
```

Returns *string*

getName ()

Reads out the bus hardware's friendly name.

```
getName (self)
```

Returns *string*

getProtocol ()

Reads out the bus protocol string.

```
getProtocol (self)
```

Returns *string*

toString ()

Reads out the bus hardware ID as a string.

```
toString (self)
```

Returns *string*

7.3 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.

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

BusHardwareOptions () [2/2]

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

Parameters	<i>options</i>	A map with options for the bus hardware to operate.
------------	----------------	---

addOption ()

Creates additional keys and values.

```
addOption(self, key, value)
```

Parameters	<i>key</i>	Example: BAUD_RATE_OPTIONS_NAME
	<i>value</i>	Example: BAUD_RATE_1000K

equals ()

Compares the BusHardwareOptions to existing ones.

```
equals (self, other)
```

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

getOptions ()

Reads out all added key-value pairs.

```
getOptions (self)
```

Returns *string map*

toString ()

Reads out all keys / values as a string.

```
toString (self)
```

Returns *string*

7.4 BusHwOptionsDefault

This default configuration options class has the following public attributes:

const <u>CanBus</u>	<i>canBus</i> = CanBus ()
const <u>Serial</u>	<i>serial</i> = Serial ()
const <u>RESTfulBus</u>	<i>restfulBus</i> = RESTfulBus()
const <u>EtherCATBus</u>	<i>ethercatBus</i> = EtherCATBus()

7.5 CanBaudRate

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

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

7.6 CanBus

Default configuration options class with the following public attributes:

string	BAUD_RATE_OPTIONS_NAME = "can adapter baud rate"
const CanBaudRate	<i>baudRate</i> = <u>CanBaudRate</u> ()
const lxxat	<i>ixxat</i> = <u>lxxat</u> ()

7.7 CanOpenNmtService

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

string	START = "START"
string	STOP = "STOP"
string	PRE_OPERATIONAL = "PRE_OPERATIONAL"
string	RESET = "RESET"
string	RESET_COMMUNICATION = "RESET_COMMUNICATION"

7.8 CanOpenNmtState

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

string	STOPPED = "STOPPED"
string	PRE_OPERATIONAL = "PRE_OPERATIONAL"
string	OPERATIONAL = "OPERATIONAL"
string	INITIALIZATION = "INITIALIZATION"
string	UNKNOWN = "UNKNOWN"

7.9 EtherCATBus Struct

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

string NETWORK_FIRMWARE_STATE_OPTION_NAME = "Network Firmware State"	Network state treated as firmware mode. Acceptable values (default = PRE_OPERATIONAL):
	<ul style="list-style-type: none"> ■ EtherCATState::PRE_OPERATIONAL ■ EtherCATState::SAFE_OPERATIONAL ■ EtherCATState::OPERATIONAL
string DEFAULT_NETWORK_FIRMWARE_STATE = "PRE_OPERATIONAL"	
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"	
string DEFAULT_SHARED_LOCK_TIMEOUT_OPTION_NAME = "Shared Lock Timeout"	Timeout in milliseconds to acquire shared lock on the network (default = 250 ms).
const unsigned int SHARED_EXCLUSIVE_LOCK_TIMEOUT = "250"	
string READ_TIMEOUT_OPTION_NAME = "Timeout"	Timeout in milliseconds for a read operation (default = 700 ms).
const unsigned int DEFAULT_READ_TIMEOUT = "700"	
string WRITE_TIMEOUT_OPTION_NAME = "Timeout"	Timeout in milliseconds for a write operation (default = 200 ms).
const unsigned int DEFAULT_WRITE_TIMEOUT = "200"	
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"	
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"	
string PDO_IO_ENABLED_OPTION_NAME = "PDO IO Enabled"	Enables or disables PDO processing for digital in- / outputs. ("True" or "False" only; default = "True").
string DEFAULT_PDO_IO_ENABLED = "True"	

7.10 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:

string	NONE = "NONE"
--------	---------------

```

string          PRE_OPERATIONAL = "PRE_OPERATIONAL"
string          OPERATIONAL = "OPERATIONAL"
string          SAFE_OPERATIONAL = "SAFE_OPERATIONAL"
string          INIT = "INITI"
string          BOOT = "BOOT"

```

7.11 Ixxat

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

```

string          ADAPTER_BUS_NUMBER_OPTIONS_NAME = "ixxat adapter bus number"
const IxxatAdapterBusNumber adapterBusNumber = IxxatAdapterBusNumber ()

```

7.12 IxxatAdapterBusNumber

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

```

string          BUS_NUMBER_0_DEFAULT = "0"
string          BUS_NUMBER_1 = "1"
string          BUS_NUMBER_2 = "2"
string          BUS_NUMBER_3 = "3"

```

7.13 DeviceHandle

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

DeviceHandle ()

Returns *ResultVoid*

7.14 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.

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

equals ()

Compares new to existing objects.

```

equals (self, other)

```

Returns *boolean*

getBusHardwareId ()

Reads out the bus hardware ID.

```
getBusHardwareId (self)
```

Returns *BusHardwareId*

getDescription ()

Reads out the device description (maybe unused).

```
getDescription (self)
```

Returns *string*

getDeviceId ()

Reads out the device ID (maybe unused).

```
getDeviceId (self)
```

Returns *unsigned int*

toString ()

Reads out the object as a string.

```
toString (self)
```

Returns *string*

getExtraId ()

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

```
getExtraId (self)
```

Returns *vector extrald_*

A vector of the additional extra ID's (may be empty), meaning is depending on the bus.

getExtraStringId ()

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

```
getExtraStringId (self)
```

Returns *string*

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

7.15 ObjectDictionary

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

getDeviceHandle ()

```
getDeviceHandle (self)
```

Returns *ResultDeviceHandle*

getObject ()

```
getObject (self, odIndex)
```

Returns *ResultObjectSubEntry*

getObjectEntry ()

```
getObjectEntry (self, 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 ()

```
readNumber (self, odIndex)
```

Returns *ResultInt*

readNumberArray ()

```
readNumberArray (self, index)
```

Returns *ResultArrayInt*

readString ()

```
readString (self, odIndex)
```

Returns *ResultString*

readBytes ()

```
readBytes (self, odIndex)
```

Returns *ResultArrayByte*

writeNumber ()

```
writeNumber (self, odIndex, value)
```

Returns *ResultVoid*

writeBytes ()

```
writeBytes (self, odIndex, data)
```

Returns *ResultVoid*

Related Links

[OdIndex](#)

7.16 ObjectEntry

This class represents an object entry of the object dictionary

The class has the following public member functions:

getName ()

Reads out the name of the object.

```
getName (self)
```

getPrivate ()

Checks if the object is private.

```
getPrivate (self)
```

getIndex ()

Reads out the address of the object index.

```
getIndex (self)
```

getDataType ()

Reads out the data type of the object.

```
getDataType (self)
```

getObjectCode ()

Reads out the object code (variable, array etc.).

```
getObjectCode (self)
```

getObjectSaveable ()

Checks if the object is saveable.

```
getObjectSaveable (self)
```

getMaxSubIndex ()

Reads out the number of subindices supported by this object.

```
getMaxSubIndex (self)
```

getSubEntry ()

```
getSubEntry (self, subIndex)
```

See also [ObjectSubEntry](#).

7.17 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.

```
getName (self)
```

getSubIndex ()

Reads out the address of the subindex.

```
getSubIndex (self)
```

getDataType ()

Reads out the data type of the subindex.

```
getDataType (self)
```

getSdoAccess ()

Checks if the subindex is accessible via SDO.

```
getSdoAccess (self)
```

getPdoAccess ()

Checks if the subindex is accessible/mappable via PDO.

```
getPdoAccess (self)
```

getBitLength ()

Checks the subindex length.

```
getBitLength (self)
```

getDefaultValueAsString ()

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

```
getDefaultValueAsString (self, key)
```

getDefaultValues ()

Reads out the default values of the subindex.

```
getDefaultValues (self)
```


readNumber ()

Reads out the numeric actual value of the subindex.

```
readNumber (self)
```

readString ()

Reads out the string actual value of the subindex.

```
readString (self)
```

readBytes ()

Reads out the actual value of the subindex in bytes.

```
readBytes (self)
```

writeNumber ()

Writes a numeric value in the subindex.

```
writeNumber (self, value)
```

writeBytes ()

Writes a value in the subindex in bytes.

```
writeBytes (self, data)
```

7.18 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.

Parameters	<i>index</i>	From 0 to 65535 (0xFFFF) incl.
	<i>subindex</i>	From 0 to 255 (0xFF) incl.

getIndex ()

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

```
getIndex (self)
```

getSubindex ()

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

```
getSubIndex (self)
```

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
```

```
toString (self)
```

Returns `0xIII:0xSS`

Default string representation

7.19 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 produced 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 ()

```
getObjectDictionaryCount (self)
```

getObjectDictionary ()

```
getObjectDictionary (self, odIndex)
```

addObjectDictionaryFromFile ()

```
addObjectDictionaryFromFile (self, absoluteXmlFilePath)
```

addObjectDictionary ()

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

```
addObjectDictionary (self, odXmlData)
```

7.20 OdTypesHelper

In addition to the following public member functions, this class contains custom data types. **Note:** To check your custom data types, open *Nanolib.py* and look for *ObjectEntryDataType_* prefixes.

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 ()

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

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

isDeftypeIndex ()

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

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

isComplexDataType ()

Informes 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  
objectDataType)
```

objectEntryDataTypeToString ()

Converts OD data type to string.

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

stringToObjectEntryDatatype ()

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

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

objectEntryDataTypeBitLength ()

Informes on bit length of an object entry data type.

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

7.21 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

7.22 ProfinetDCP

Windows-implemented, the ProfinetDCP interface uses *Win10Pcap* or *Npcap*. It thus searches the dynamically loaded *wpcap.dll* library in the following order:

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

Under Linux, the calling application must have `CAP_NET_ADMIN` and `CAP_NET_RAW` capabilities. To enable: `sudo setcap 'cap_net_admin,cap_net_raw+eip' ./executable`

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

getScanTimeout ()

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

```
getScanTimeout (self)
```

setScanTimeout ()

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

```
setScanTimeout (self, timeoutMsec)
```

getResponseTimeout ()

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

```
getResponseTimeout (self)
```

setResponseTimeout ()

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

```
setResponseTimeout (self, timeoutMsec)
```

setupProfinetDevice ()

Establishes the following device settings:

- Device name / vendor
- MAC/IP address
- Network mask
- Gateway

```
setupProfinetDevice (self, busHardwareId, profinetDevice, savePermanent)
```

resetProfinetDevice ()

Stops the device and resets it to factory defaults.

```
resetProfinetDevice (self, busHardwareId, profinetDevice)
```

blinkProfinetDevice ()

Commands the Profinet device to start blinking its Profinet LEDs.

```
blinkProfinetDevice (self, busHardwareId, profinetDevice)
```

7.23 ProfinetDevice

The Profinet device data, created from the *profinet_dcp.hpp* header file, 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 the format: `macAddress = {0, 0, 0, 0, 0, 0};`. Whereas IP address, network mask and gateway are all interpreted as big endian hex numbers. For example:

IP address: 192.168.0.2	0xC0A80002
Netowrk mask: 255.255.0.0	0xFFFF0000
Gateway: 192.168.0.2	0xC0A80001

7.24 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	<i>string</i>	errorString
	<i>NlcErrorCode</i>	errorCode
	<i>uint32_t</i>	exErrorCode

Also, this class has the following public member functions:

hasError ()

Reads out a function call's success.

```
hasError (self)
```

Returns	<i>false</i>	Successful call. Use <i>getResult ()</i> to read out the value.
	<i>true</i>	Failed call. Use <i>getError ()</i> to read out the value.

getError ()

Reads out the reason if a function call fails.

```
getError (self)
```

Returns *const string*

getErrorCode () const

```
getErrorCode (self)
```

getExErrorCode () const

```
uint32_t getExErrorCode () const
```

```
getExErrorCode (self)
```

7.24.1 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:

7.24.2 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.

```
getResult (self)
```

Returns

7.24.3 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.

```
getResult (self)
```

Returns *const string*

7.24.4 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.

```
getResult (self)
```

Returns *const vector<uint8_t>*

7.24.5 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.

```
getResult (self)
```

Returns *const vector<uint64_t>*

7.24.6 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.

```
getResult (self)
```

Parameters *const*
vector<BusHardwareId>

7.24.7 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.

```
getResult (self)
```

Returns *const vector<DeviceId>*

7.24.8 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.

```
getResult (self)
```

Returns *const vector<DeviceId>*

7.24.9 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.

```
getResult (self)
```

Returns *DeviceHandle*

7.24.10 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.

```
getResult (self)
```

Returns *DeviceHandle*

7.24.11 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.

```
getResult (self)
```

Returns *const vector<Deviceld>*

7.24.12 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.

```
getResult (self)
```

Returns *const vector<Deviceld>*

7.24.13 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.

```
getResult (self)
```

Returns *const vector<Deviceld>*

7.25 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 busy, inexistent, cut-off or defect.
CommunicationError	C: Communication. D: Communication unreliable. R: Unexpected data, wrong CRC, frame or parity errors, etc.

Error code	C: Category D: Description R: Reason
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.

7.26 NlcCallback

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

callback ()

```
callback (self)
```

Returns *ResultVoid*

7.27 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 ()

```
callback (self)
```

Returns *ResultVoid*

7.28 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 ()

```
callback (self, info, devicesFound, data)
```

Returns *ResultVoid*

7.29 SamplerInterface

Use this class to configure, start and stop samplers, or to get sampled data ..., and even to 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] <i>deviceHandle</i>	Specifies what device to configure the sampler for.
	[in] <i>samplerConfiguration</i>	Specifies the values of <u>configuration attributes</u> .
Returns	<i>ResultVoid</i>	Confirms that a <u>void function</u> has run.

getData ()

Gets the sampled data.

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

Parameters	[in] <i>deviceHandle</i>	Specifies what device to get the data for.
Returns	<i>ResultSampleDataArray</i>	Delivers the sampled data, which can be an empty array if <u>samplerNotify</u> is active on <u>start ()</u> .

getLastError ()

Gets a sampler's last error.

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

Returns	<i>ResultVoid</i>	Confirms that a <u>void function</u> has run.
---------	-------------------	---

getState ()

Gets a sampler's status.

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

Returns *ResultSamplerState* Delivers the sampler condition.

start ()

Starts a sampler.

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

Parameters	[in] <i>deviceHandle</i> [in] <i>samplerNotify</i> [in] <i>applicationData</i>	Specifies what device to start the sampler for. Specifies what optional info to report (can be <i>nullptr</i>). 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 <i>samplerNotify</i> .
Returns	<i>ResultVoid</i>	Confirms that a <u>void function</u> has run.

stop ()

Stops a sampler.

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

Parameters	[in] <i>deviceHandle</i>	Specifies what device to stop the sampler for.
Returns	<i>ResultVoid</i>	Confirms that a <u>void function</u> has run.

7.30 SamplerConfiguration

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

Public attributes

std::vector <OdIndex>	<i>trackedAddresses</i>	OD addresses to be sampled.
<u>OdIndex</u>	<i>triggerAddress</i>	OD address of start trigger.
uint32_t	<i>triggerValue</i>	Start trigger condition value / bit.
uint16_t	<i>periodMilliseconds</i>	Sampling period in ms.
uint16_t	<i>numberOfSamples</i>	Samples amount.
uint16_t	<i>preTriggerNumberOfSamples</i>	Samples pre-trigger amount.
bool	<i>forceSoftwareImplementation</i>	A software emulation for devices that support no sampling (as their firmware can't read <i>trackedAddresses</i> data directly).
SamplerMode	<i>mode</i>	<i>Normal</i> , <i>repetitive</i> or <i>continuous</i> sampling.
SamplerTriggerCondition	<i>triggerCondition</i>	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

```

Static public attributes

static constexpr size_t MAX_TRACKED_ADDRESSES = 12 Up to 12 OD addresses to track.

7.31 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] <i>lastError</i>	Reports the last error occurred while sampling.
[in] <i>samplerState</i>	Reports the sampler status at notification time.
[in] <i>sampleDatas</i>	Reports the sampled-data array.
[in] <i>applicationData</i>	Reports application-specific data.

7.32 Serial

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

:string	BAUD_RATE_OPTIONS_NAME = "serial baud rate"
SerialBaudRate	<i>baudRate</i> = SerialBaudRate ()
string	PARITY_OPTIONS_NAME = "serial parity"
SerialParity	<i>parity</i> = SerialParity ()

7.33 SerialBaudRate

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

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

7.34 SerialParity

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

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

8 Licenses

The NanoLib interface (*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 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.

9 Imprint, contact, versions

© 2022 Nanotec Electronic GmbH & Co. KG. All rights reserved. No portion of this document to be reproduced without prior written consent. Specifications subject to change without notice. Errors, omissions, and modifications excepted. Original version.

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	++ Added >> Changed ## Fixed	Product
1.0.0 ^{2021.06}	Edition	0.7.0
1.0.1 ^{2021.11}	<ul style="list-style-type: none"> ■ ++ More <u>ObjectEntryDataType</u> (complex and profile-specific) ■ ++ <u>IOError</u> return if <u>connectDevice</u> and <u>scanDevices</u> find none ■ ++ Only 100 ms nominal timeout for CanOpen / Modbus ■ ++ <u>OdTypesHelper</u> class 	0.7.1
1.0.2 ^{2022.03}	<ul style="list-style-type: none"> ■ ++ Python 3.10 / Linux ARM64 support added ■ ++ USB mass storage / <u>REST</u> / <u>Profinet DCP</u> support added ■ ++ <u>checkConnectionState</u> () ■ ++ <u>getDeviceBootloaderVersion</u> () ■ ++ <u>ResultProfinetDevices</u> ■ ++ <u>NlcErrorCode</u> (replaced <u>NanotecExceptions</u>) ■ ++ NanoLib Modbus: VCP / USB hub unified to USB ■ ## Modbus TCP scanning returns results. ■ ## Modbus TCP communication latency remains constant. 	0.8.0
1.1.0 ^{2022.08}	<ul style="list-style-type: none"> ■ ++ <u>getDeviceHardwareGroup</u> () ■ ++ <u>getProfinetDCP</u> (<u>isServiceAvailable</u>) ■ ++ <u>getProfinetDCP</u> (<u>validateProfinetDeviceIp</u>) ■ ++ <u>autoAssignObjectDictionary</u> () ■ ++ <u>getXmlFileName</u> () ■ ++ <u>const std::string & xmlFilePath</u> in <u>addObjectDictionary</u> () ■ ++ <u>getSamplerInterface</u> () ■ ++ <u>rebootDevice</u> () ■ ++ Error code <u>ResourceUnavailable</u> for <u>getDeviceBootloaderVersion</u> (), <u>~VendorId</u> (), <u>~HardwareVersion</u> (), <u>~SerialNumber</u>, and <u>~Uid</u> ■ >> <u>firmwareUploadFromFile</u> now <u>uploadFirmwareFromFile</u> () ■ >> <u>firmwareUpload</u> () now <u>uploadFirmware</u> () ■ >> <u>bootloaderUploadFromFile</u> () now <u>uploadBootloaderFromFile</u> () ■ >> <u>bootloaderUpload</u> () now <u>uploadBootloader</u> () ■ >> <u>bootloaderFirmwareUploadFromFile</u> () to <u>uploadBootloaderFirmwareFromFile</u> () ■ >> <u>bootloaderFirmwareUpload</u> () now <u>uploadBootloaderFirmware</u> () ■ >> <u>nanojUploadFromFile</u> () now <u>uploadNanoJFromFile</u> () ■ >> <u>nanojUpload</u> () now <u>uploadNanoJ</u> () ■ >> <u>objectDictionaryLibrary</u> () now <u>getObjectDictionaryLibrary</u> () ■ >> <u>String_String_Map</u> now <u>StringStringMap</u> ■ >> Nanolib-Common: faster execution of <u>listAvailableBusHardware</u> and <u>openBusHardwareWithProtocol</u> with <u>Ixxat</u> adapter ■ >> Nanolib-CANopen: default settings used (<u>1000k</u> baudrate, <u>Ixxat</u> bus number 0) if bus hardware options empty ■ >> Nanolib-RESTful: admin permission obsolete for communication with Ethernet bootloaders under Windows if <u>npcap</u> / <u>winpcap</u> driver is available ■ ## NanoLib-CANopen: bus hardware now opens crashless with empty options 	1.0.0

Document	++ Added >> Changed ## Fixed	Product
1.1.1 ^{2022.09}	■ ## NanoLib-Common: <i>openBusHardwareWithProtocol ()</i> with no memory leak now	
	■ ++ EtherCAT support	1.0.1
	■ ## NanoLib-Modbus: <i>scanDevice</i> for ModbusTCP protocol returns an error when non-ModbusTCP devices are present on the bus	(B349)