User Manual NanoLib

Python

Valid with NanoLib version 0.8.0

User Manual Version: 1.0.2

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

NOTICE

Malfunction from 32-bit operation!

► Use 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).

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	Fieldbus adapters / cables
0.7.1	■ 64-bit system (mandatory) ■ Windows 10: *w/ Visual Studio; C++ Redistributable; Python 3.7 to 3.9* ■ Linux: *Ubuntu 18.04.2 LTS w/ Python 3.7 to 3.9*	■ 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*
0.8.0	■ Windows 10: *Python 3.10 added* ■ Linux: *ARM64 added w/ Python 3.7 to 3.10*	■ VCP / USB hub: *now uniform USB* ■ USB mass storage: *USB cable* ■ REST: *Ethernet cable*

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 classes as source code (API)
- Libraries that facilitate the communication via the fieldbus: *nanolibm_canopen.dll*, *nanolibm_modbus.dll*
- 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 organize freely customizable motor controller / fieldbus functions around a strictly preconfigured core. NanoLib contains the following modules:

User interface (API)

Interface and helper classes which

- grant access to your controller's OD (object dictionary)
- are based on the NanoLib core functionalities.

NanoLib core

Libraries which

- implement the API functionality
- interact with bus libraries.

Communication 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 library *nanolib_python.pyd*. 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 Python 3.7, 3.8 or 3.9 from www.python.org/.
- 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:

- 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	*param_a*	Additional comment if needed.
	param_b	
Returns	*ResultVoid*	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	*ResultBusHwIds*	Delivers a fieldbus ID array.

openBusHardwareWithProtocol ()

Use this function to connect bus hardware.

```
openBusHardwareWithProtocol(self, busHwId, 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.

```
getProtocolSpecificAccessor (self, 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.

```
setBusState (self, busHwId, 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.

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

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

addDevice ()

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

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

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

connectDevice ()

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

```
connectDevice (self, 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 a device's name by *deviceHandle*.

```
getDeviceName (self, 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 a device's product code by *deviceHandle*.

```
getDeviceProductCode (self, 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 by *deviceHandle*.

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

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

getDeviceId ()

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

```
getDeviceId (self)
```

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 all devices' ID from the NanoLib internal list.

```
getDeviceIds (self)
```

Returns	*ResultDeviceIds*	Delivers a device ID list.

getDeviceUid ()

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

```
getDeviceUid (self)
```

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 a device's serial number from the NanoLib internal list.

```
getDeviceSerialNumber (self)
```

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

getDeviceHardwareVersion ()

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

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

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

getDeviceFirmwareBuildId ()

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

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

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

getDeviceBootloaderVersion ()

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

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

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

getDeviceBootloaderBuildId ()

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

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

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

getDeviceState ()

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

```
getDeviceState (self, 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.

```
setDeviceState (self, deviceHandle, 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 return a specific device's last known connection state by *deviceHandle* (= *Disconnected, Connected, ConnectedBootloader*)

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

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

checkConnectionState ()

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

.

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

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

assignObjectDictionary ()

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

```
assignObjectDictionary (self, deviceHandle, 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 by *deviceHandle*.

```
getAssignedObjectDictionary (self, 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.

```
objectDictionaryLibrary (self)
```

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.

```
setLoggingLevel (self, 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.

```
readNumber (self)
```

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.

```
readNumberArray (self, deviceHandle, 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.

```
readBytes (self, 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.

```
readString (self)
```

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.

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

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.

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

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.

```
firmwareUpload (self, deviceHandle, fwData, 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.

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

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

bootloaderUpload ()

Use this function to update your controller bootloader.

```
bootloaderUpload (self, deviceHandle, btData, 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.

```
bootloaderUploadFromFile (self, deviceHandle, bootloaderAbsoluteFilePath,
 callback)
```

Parameters	*deviceHandle*	Specifies which bus device NanoLib should update.
	bootloaderAbsoluteFilePath	Path to file containing bootloader data (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.

```
bootloaderFirmwareUpload (self, deviceHandle, btData, fwData, 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.

```
bootloaderFirmwareUploadFromFile (self, deviceHandle,
 bootloaderAbsoluteFilePath, absoluteFilePath, callback)
```

Parameters	*deviceHandle*	Specifies which bus device NanoLib should update.
	bootloaderAbsoluteFilePath	Path to file containing bootloader data (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.

```
nanojUpload (self, deviceHandle, vmmData, 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.

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

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

disconnectDevice ()

Use this function to disconnect your device.

```
disconnectDevice (self, 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.

```
removeDevice (self, 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.

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

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

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.)
- 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.

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.

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

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.

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

addOption ()

Creates additional keys and values.

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

Parameters	*key*	Example: BAUD_RATE_OPTIONS_NAME
	value	Example: BAUD_RATE_1000K

equals ()

Compares the BusHardwareOptions to existing ones.

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

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.

```
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 CanBus	**canBus** = CanBus ()
const Serial	**serial** = Serial ()

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	**baudRate** = CanBaudRate ()
const Ixxat	**ixxat** = Ixxat ()

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 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 ()

7.10 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"

7.11 DeviceHandle

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

DeviceHandle ()

Returns	*ResultVoid*

7.12 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.

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.

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

Returns	*boolean*

getBusHarwareId ()

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 extraId_*	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).

```
getExtraStringId(self)
```

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

7.13 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*

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*

7.14 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.

7.15 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.

```
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)
```

7.16 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.

Parameters	*index*	From 0 to 65535 (0xFFFF) incl.
	subindex	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	*0xIIII:0xSS*	Default string representation

7.17 OdLibrary

Use this programming interface to create instances of the *ObjectDictionary* class from XML (by *assignObjectDictionary,* you can then associate them to a specific device). Successfully created *ObjectDictionary* instances are stored in the *OdLibrary* object to be accessed by index. The *ODLibrary* class represents an object dictionary library and has the following public member functions:

getObjectDictionaryCount

```
getObjectDictionaryCount(self)
```

getObjectDictionary

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

addObjectDictionaryFromFile

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

addObjectDictionary

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

7.18 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

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)
```

7.19 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

7.20 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 ()

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

```
getScanTimeout(self)
```

setScanTimeout ()

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

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

getResponseTimeout ()

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

```
getResponseTimeout(self)
```

setResponseTimeout ()

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

```
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)
```

7.21 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};`

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

7.22 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.

```
hasError(self)
```

Returns	*false*	Means: call success. Use *getResult ()* to read out the value.
	true	Means: call failure. Use *getError ()* 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)
```

7.22.1 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:

7.22.2 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	

7.22.3 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*

7.22.4 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>*

7.22.5 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>*

7.22.6 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>*

7.22.7 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>*

7.22.8 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>*

7.22.9 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*

7.22.10 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*

7.22.11 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<DeviceId>*

7.22.12 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<DeviceId>*

7.22.13 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<DeviceId>*

7.23 NlcErrorCode

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

Error Code		Details
Success	Category:	None.
	Description:	No error.
	Reason:	The operation completed successfully.
GeneralError	Category:	Unspecified.
	Description:	Unspecified error.
	Reason:	Failure that cannot be assigned to other categories.
BusUnavailable	Category:	Bus.
	Description:	Hardware bus not available.
	Reason:	Bus does not exist or is no longer available.
CommunicationError	Category:	Communication.
	Description:	Communication unreliable.
	Reason:	Unexpected data, wrong CRC, frame or parity errors, etc.
ProtocolError	Category:	Protocol.
	Description:	Protocol error.
	Reason:	Response following unsupported protocol option, device report unsupported protocol, error in the protocol (say, SDO segment sync bit), etc.
ODDoesNotExist	Category:	Object dictionary.
	Description:	OD address inexistent.
	Reason:	The address does not exist in the object dictionary.
ODInvalidAccess	Category:	Object dictionary.

Error Code	Details	
	Description:	Access to OD address invalid.
	Reason:	Attempt to write a read-only, or to read from a write-only, address.
ODTypeMismatch	Category:	Object dictionary.
	Description:	Type mismatch.
	Reason:	The value cannot be converted to the specified type, say, in an attempt to treat a string as a number.
OperationAborted	Category:	Application.
	Description:	Operation aborted.
	Reason:	The operation has been aborted on application request. Returns only on operation interrupt by callback function, say, from bus-scanning.
OperationNotSupported	Category:	Common.
	Description:	Operation not supported.
	Reason:	The operation is not supported on the hardware bus or device.
InvalidOperation	Category:	Common.
	Description:	Operation incorrect or invalid.
	Reason:	The requested operation is incorrect in the current context or invalid with the current arguments. Attempt to reconnect to an already connected bus or device, or to disconnect from a bus or a device already disconnected. Attempt to perform bootloader operation in firmware mode or vice versa.
InvalidArguments	Category:	Common.
	Description:	Argument invalid.
	Reason:	The arguments passed are invalid.
AccessDenied	Category:	Common.
	Description:	Access is denied.
	Reason:	The current execution context does not have sufficient privileges or capabilities to perform the requested operation.
ResourceNotFound	Category:	Common.
	Description:	Specified resource not found.
	Reason:	The specified hardware bus, protocol, device, OD address on device, or file was not found.
ResourceUnavailable	Category:	Common.
	Description:	Specified resource not available.
	Reason:	The specified resource does not exist or is temporarily unavailable in part or in full.
OutOfMemory	Category:	Common.
	Description:	Insufficient memory.

Error Code		Details
	Reason:	Not enough memory resources are available to process this command.
TimeOutError	Category:	Common.
	Description:	Operation timed out.
	Reason:	The operation returned because the timeout period expired. Timeout may be a device response time, a time to gain shared or exclusive access to a resource, or a time to switch the bus or device to a suitable state.

7.24 NlcCallback

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

callback ()

```
callback(self)
```

Returns *ResultVoid*

7.25 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 ()

```
callback(self)
```

Returns *ResultVoid*

7.26 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 ()

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

Returns *ResultVoid*

7.27 Serial

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

:string **BAUD_RATE_OPTIONS_NAME** = "serial baud rate"

SerialBaudRate	**baudRate** = SerialBaudRate ()
string	**PARITY_OPTIONS_NAME** = "serial parity"
SerialParity	**parity** = SerialParity ()

7.28 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"

7.29 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 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.

9 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
1.0.0 (06/2021)	Edition	0.7.0
1.0.1 (11/2021)	■ More *ObjectEntryDataType* (complex and profile-specific) ■ *IOError* return if *connectDevice* and *scanDevices* find none ■ Only 100 ms nominal timeout for CanOpen / Modbus ■ Added *OdTypesHelper* class	0.7.1
1.0.2 (03/2022)	■ Python 3.10 / Linux ARM64 support added ■ USB mass storage / REST / Profinet DCP support added ■ NanoLib Modbus: VCP / USB hub unified to USB ■ Fixed: Modbus TCP scanning returns results. ■ Fixed: Modbus TCP communication latency remains constant. ■ Added: □ checkConnectionState () □ getDeviceBootloaderVersion () □ ResultProfinetDevices □ NlcErrorCode (replaced *NanotecExceptions*)	0.8.0