

User Manual NanoLib

Python

Contents

1 Document aim and typography.....	4
2 Before you start.....	5
2.1 System and hardware requirements.....	5
2.2 Intended use and audience.....	5
2.3 Scope of delivery and warranty.....	5
3 The NanoLib architecture.....	6
3.1 User interface.....	6
3.2 NanoLib core.....	6
3.3 Communication libraries.....	6
4 Getting started.....	7
4.1 Prepare your system.....	7
4.2 Install the adapter driver for Windows.....	7
4.3 Install the adapter driver for Linux.....	7
4.4 Connect your hardware.....	7
4.5 Load NanoLib.....	8
5 Windows Setup.....	9
6 Linux Setup.....	11
7 Classes / functions reference.....	13
7.1 NanoLibAccessor.....	13
7.2 BusHardwareId.....	21
7.3 BusHardwareOptions.....	22
7.4 BusHwOptionsDefault.....	23
7.5 CanBaudRate.....	23
7.6 CanBus.....	23
7.7 CanOpenNmtService.....	23
7.8 CanOpenNmtState.....	24
7.9 Ixxat.....	24
7.10 IxxatAdapterBusNumber.....	24
7.11 DeviceHandle.....	24
7.12 DeviceId.....	24
7.13 ObjectDictionary.....	26
7.14 ObjectEntry.....	27
7.15 ObjectSubEntry.....	28
7.16 OdIndex.....	29
7.17 OdLibrary.....	30
7.18 OdTypesHelper.....	30
7.19 Result classes.....	31
7.19.1 ResultVoid.....	32
7.19.2 ResultInt.....	32
7.19.3 ResultString.....	32

7.19.4 ResultArrayByte.....	33
7.19.5 ResultArrayInt.....	33
7.19.6 ResultBusHwIds.....	33
7.19.7 ResultDeviceId.....	33
7.19.8 ResultDeviceIds.....	33
7.19.9 ResultDeviceHandle.....	34
7.19.10 ResultConnectionState.....	34
7.19.11 ResultObjectDictionary.....	34
7.19.12 ResultObjectEntry.....	34
7.19.13 ResultObjectSubEntry.....	34
7.20 NlcCallback.....	35
7.21 NlcDataTransferCallback.....	35
7.22 NlcScanBusCallback.....	35
7.23 Serial.....	35
7.24 SerialBaudRate.....	36
7.25 SerialParity.....	36
7.26 NanotecException classes.....	36
7.26.1 AbortException.....	36
7.26.2 InvalidAddressException.....	37
7.26.3 ProtocolException.....	37
7.26.4 ResourceException.....	37
7.26.5 TimeoutException.....	37
8 Licenses.....	38
9 Imprint, contact, document history.....	39

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 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*.
- Example 2: Open the *Tool* tab and select *Comment*.
- Example 3: In principle, this document distinguishes between:
 - User (= *Nutzer; usuario; utente* [pt.]; *utilisateur; utente* [it.] etc.).
 - Third-party user (= *Drittnutzer; tercero usuario; terceiro utente; tiers utilisateur; terzo utente* etc.).
 - End user (= *Endnutzer; usuario final; utente final; utilisateur final; utente finale* etc.).

Courier marks code blocks or programming commands.

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

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

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

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

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

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

2 Before you start

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

2.1 System and hardware requirements

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

NanoLib	64-bit OS requirements	Fieldbus adapters / cables
v 0.7.1	<ul style="list-style-type: none"> ■ Windows 10: with Python 3.7, 3.8 or 3.9 ■ Linux: Ubuntu 18.04.2 LTS with Python 3.7, 3.8 or 3.9 	<ul style="list-style-type: none"> ■ CANopen: <ul style="list-style-type: none"> □ IXXAT USB-to-CAN V2 □ Nanotec ZK-USB-CAN-1 ■ Modbus RTU: <ul style="list-style-type: none"> □ Nanotec ZK-USB-RS485-1 or equivalent USB-RS485 adapter □ USB cable via virtual comport (VCP) ■ Modbus TCP: <ul style="list-style-type: none"> □ Ethernet cable according to product datasheet.

2.2 Intended use and audience

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

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

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

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

2.3 Scope of delivery and warranty

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

- | | |
|---|--|
| <ul style="list-style-type: none"> ■ Interface classes as source code (API) ■ Libraries that facilitate the communication via the fieldbus: <i>nanolibm_canopen.dll</i>, <i>nanolibm_modbus.dll</i> | <ul style="list-style-type: none"> ■ Core functions as library in binary format: <i>_nanolib_python.pyd</i> ■ Example code: <i>nanolib_example.py</i> and <i>nanolib_helper.py</i> |
|---|--|

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

3 The NanoLib architecture

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

User interface (API)	NanoLib core	Communication libraries
Interface and helper classes which	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 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 "nanotec*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 the available fieldbus hardware.

```
listAvailableBusHardware (self)
```

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

openBusHardwareWithProtocol ()

Use this function to establish connection with a 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 the execution of a <u>void function</u> .

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 the execution of a <u>void function</u> .

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 the execution of a <u>void function</u> .

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 the NanoLib internal device list and 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 establish connection with a device using *deviceHandle*.

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

Parameters	<i>deviceHandle</i>	Specifies which bus device NanoLib should connect to.
Returns	<i>ResultVoid</i>	Confirms the execution of a <u>void function</u> .
	<i>IOError</i>	Informes that a device is not found.

getDeviceName ()

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

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

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

getDeviceProductCode ()

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

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

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

getDeviceVendorId ()

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

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

Parameters	<i>deviceHandle</i>	Specifies which bus device NanoLib should get the vendor id for.
Returns	<i>ResultInt</i>	Delivers the product code as an <u>integer</u> .

getDeviceId ()

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

```
getDeviceId(self)
```

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

getDeviceIds ()

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

```
getDeviceIds(self)
```

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

getDeviceUid ()

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

```
getDeviceUid(self)
```

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

getDeviceSerialNumber ()

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

```
getDeviceSerialNumber(self)
```

Parameters	<i>deviceHandle</i>	Specifies which bus device NanoLib should get the serial number for.
Returns	<i>ResultString</i>	Delivers a device name as a <u>string</u> .

getDeviceBootloaderBuildId ()

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

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

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

getDeviceFirmwareBuildId ()

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

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

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

getDeviceHardwareVersion ()

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

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

Parameters	<i>deviceHandle</i>	Specifies which bus device NanoLib should get the hardware version for.
Returns	<i>ResultString</i>	Delivers a device name as a <u>string</u> .

getDeviceState ()

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

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

Parameters	<i>deviceHandle</i>	Specifies which bus device NanoLib should get the state for.
Returns	<i>ResultString</i>	Delivers a device name as a <u>string</u> .

setDeviceState ()

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

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

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

getConnectionState ()

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

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

Parameters	<i>deviceHandle</i>	Specifies which bus device NanoLib should set the state for.
Returns	<i>ResultConnectionState</i>	Delivers a <u>connection state</u> .

assignObjectDictionary ()

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

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

Parameters	<i>deviceHandle</i>	Specifies which bus device NanoLib should assign the OD to.
	<i>objectDictionary</i>	
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 using *deviceHandle*.

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

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

objectDictionaryLibrary ()

This function returns a reference to the object dictionary library.

```
objectDictionaryLibrary(self)
```

Returns	<i>OdLibrary&</i>	Shows which <u>object dictionary</u> 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	<i>level</i>	The following levels are possible:
0 = <i>Off</i>	Switches off the logging entirely.	
1 = <i>Trace</i>	Lowest level, logs everything (expect huge logfiles).	
2 = <i>Debug</i>	Logs only debug information.	
3 = <i>Info</i>	Default level.	
4 = <i>Warn</i>	Message on recoverable problems.	
5 = <i>Error</i>	Highest level, only for messages followed very likely by a program exit.	

readNumber ()

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

```
readNumber(self)
```

Parameters	<i>deviceHandle</i>	Specifies which bus device NanoLib should read from.
	<i>odIndex</i>	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>	Specifies which bus device NanoLib should read from.
	<i>index</i>	Array object index..
Returns	<i>ResultArrayInt</i>	Delivers an <u>array of integers</u> .

readBytes ()

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

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

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

readString ()

Use this function to read strings from the object directory.

```
readString(self)
```

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

writeNumber ()

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

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

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

writeBytes ()

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

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

Parameters	<i>deviceHandle</i>	Specifies which bus device NanoLib should 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 the execution of a <u>void function</u> .

firmwareUpload ()

Use this function to update your controller firmware.

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

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

bootloaderUpload ()

Use this function to update your controller bootloader.

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

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

bootloaderUploadFromFile ()

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

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

Parameters	<i>deviceHandle</i>	Specifies which bus device NanoLib should update.
	<i>bootloaderAbsolutePath</i>	Path to file containing bootloader data (string)
	<i>NlcDataTransferCallback</i>	A data progress tracer.
Returns	<i>ResultVoid</i>	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	<i>deviceHandle</i>	Specifies which bus device NanoLib should update.
	<i>btData</i>	Array containing bootloader data.
	<i>fwData</i>	Array containing firmware data.
	<i>NlcDataTransferCallback</i>	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,
    bootloaderAbsolutePath, absoluteFilePath, callback)
```

Parameters	<i>deviceHandle</i>	Specifies which bus device NanoLib should 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 <u>data progress</u> tracer.
Returns	<i>ResultVoid</i>	Confirms the execution of a <u>void function</u> .

nanojUpload ()

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

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

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

nanojUploadFromFile ()

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

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

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

disconnectDevice ()

Use this function to disconnect your device.

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

Parameters	<i>deviceHandle</i>	Specifies which bus device NanoLib should disconnect from.
Returns	<i>ResultVoid</i>	Confirms the execution of a <u>void function</u> .

removeDevice ()

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

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

Parameters	<i>deviceHandle</i>	Specifies which bus device NanoLib should remove from the list.
Returns	<i>ResultVoid</i>	Confirms the execution of a <u>void function</u> .

closeBusHardware ()

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

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

Parameters	<i>busHwId</i>	Specifies the <u>fieldbus</u> to close the connection to.
Returns	<i>ResultVoid</i>	Confirms the execution of a <u>void function</u> .

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_ protocol_ hardwareSpecifier_ extraHardwareSpecifier name_</i>	Hardware type (= ZK-USB-CAN-1 etc.) Bus communication protocol (= CANopen etc.) The specifier of a hardware (= COM3 etc.) The extra specifier of the hardware (USB location info for example) 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 false</i>	If both are equal in all values. If the values differ.

getBusHardware ()

Reads out the bus hardware string.

```
getBusHardware(self)
```

Returns	<i>string</i>
---------	---------------

getHardwareSpecifier ()

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

```
getHardwareSpecifier(self)
```

Returns	<i>string</i>
---------	---------------

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 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 lxxat      lxxat = lxxat ()
```

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_*
deviceId_
description_

Identifier of the bus.

An index; subject to the bus (= CANopen node ID etc.).

A description (maybe empty); subject to the bus.

extrald_

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*

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

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

Returns	<i>string</i>	The additional <i>StringId</i> (may be empty), meaning is depending on the bus.
---------	---------------	---

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

Returns *ResultDeviceHandle*Returns *ResultObjectSubEntry*Returns *ResultObjectEntry*Returns ResultIntReturns *ResultArrayInt*Returns *ResultString*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	<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 *0xIII: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

7.17 OdLibrary

This 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

Informes 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 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

Informes on bit length of an object entry data type.

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

7.19 Result classes

Use the "optional" return values of these classes to check if a function call had success or not, and also locate the fail reasons. On a success, the *hasError ()* function returns *false*. Via *getResult ()*, you can read out the result value (depending on the result type, e.g., [ResultInt](#)). If your call fails, you can read out the reason via *getError ()*.

Protected attributes	<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 *true*
 false

Means: call success. Use *getResult ()* to read out the value.
 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)
```

7.19.1 ResultVoid

NanoLib sends you an instance of this class if the function returns void. This class inherits the public functions and protected attributes from the [result class](#) and has the following public member functions:

7.19.2 ResultInt

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

getResult ()

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

```
getResult(self)
```

Returns

7.19.3 ResultString

NanoLib sends you an instance of this class if the function returns a string. This class inherits the public functions and protected attributes from the [result class](#) and has the following public member functions:

getResult ()

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

```
getResult(self)
```

Returns *const string*

7.19.4 ResultArrayByte

NanoLib sends you an instance of this class if the function returns a byte array. This class inherits the public functions and protected attributes from the [result class](#) and has the following public member functions:

getResult ()

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

```
getResult (self)
```

Returns *const vector<uint8_t>*

7.19.5 ResultArrayInt

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

getResult ()

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

```
getResult (self)
```

Returns *const vector<uint64_t>*

7.19.6 ResultBusHwIds

NanoLib sends you an instance of this class if the function returns a [bus-hardware-ID](#) array. This class inherits the public functions and protected attributes from the [result class](#) and has the following public member functions:

getResult ()

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

```
getResult (self)
```

Parameters *const vector<BusHardwareId>*

7.19.7 ResultDeviceId

NanoLib sends you an instance of this class if the function returns a [device ID](#). This class inherits the public functions and protected attributes from the [result class](#) and has the following public member functions:

getResult ()

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

```
getResult (self)
```

Returns *const vector<DeviceId>*

7.19.8 ResultDeviceIds

NanoLib sends you an instance of this class if the function returns a [device ID](#) array. This class inherits the public functions and protected attributes from the [result class](#) and has the following public member functions:

getResult ()

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

```
getResult (self)
```

Returns `const vector<Deviceld>`

7.19.9 ResultDeviceHandle

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

getResult ()

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

```
getResult (self)
```

Returns `DeviceHandle`

7.19.10 ResultConnectionState

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

getResult ()

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

```
getResult (self)
```

Returns `DeviceHandle`

7.19.11 ResultObjectDictionary

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

getResult ()

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

```
getResult (self)
```

Returns `const vector<Deviceld>`

7.19.12 ResultObjectEntry

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

getResult ()

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

```
getResult (self)
```

Returns `const vector<Deviceld>`

7.19.13 ResultObjectSubEntry

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

getResult ()

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

```
getResult (self)
```

Returns *const vector<DeviceId>*

7.20 NlcCallback

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

callback ()

```
callback (self)
```

Returns *ResultVoid*

7.21 NlcDataTransferCallback

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

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

The class has the following public member function:

callback ()

```
callback (self)
```

Returns *ResultVoid*

7.22 NlcScanBusCallback

Use this callback class for bus scanning.

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

The class has the following public member function.

callback ()

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

Returns *ResultVoid*

7.23 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 ()

7.24 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"

7.25 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"

7.26 NanotecException classes

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

```
what(self)
```

```
getErrorCode(self)
```

```
getExErrorCode(self)
```

7.26.1 AbortException

If an operation stalls, use this class to override the abort mechanism. The class offers the function

`AbortException(message, abortCode)`

. From `NanotecException`, it also inherits the following public functions:

```
what(self)
```

```
getErrorCode(self)
```

```
getExErrorCode(self)
```

7.26.2 InvalidAddressException

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

```
what(self)
```

```
getErrorCode(self)
```

```
getExErrorCode(self)
```

7.26.3 ProtocolException

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

```
what(self)
```

```
getErrorCode(self)
```

```
getExErrorCode(self)
```

7.26.4 ResourceException

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

```
what(self)
```

```
getErrorCode(self)
```

```
getExErrorCode(self)
```

7.26.5 TimeoutException

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

```
what(self)
```

```
getErrorCode(self)
```

```
getExErrorCode(self)
```

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

© 2021 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	Changes	Product release
1.0.0 (06/2021)	Edition	0.7.0
1.0.1 (11/2021)	<ul style="list-style-type: none">■ More <i>ObjectEntryDataType</i> (complex and profile-specific)■ <i>IOError</i> return if <i>connectDevice</i> and <i>scanDevices</i> find none■ Only 100 ms nominal timeout for CanOpen / Modbus■ Added <i>OdTypesHelper</i> class	0.7.1