User Manual NanoLib

Python

Valid with product version 1.4.0

User Manual Version: 1.3.4

Contents

1 Document aim and conventions

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

Underlined text marks a cross reference or hyperlink.

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

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

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

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

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

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

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

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

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

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

2 Before you start

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

2.1 System and hardware requirements

Notice

Malfunction from 32-bit operation or discontinued system!

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

►Observe OEM discontinuations and ~instructions.

NanoLib ***1.4.0*** supports all Nanotec products with CANopen, Modbus RTU (also USB on virtual *com* port), Modbus TCP, EtherCat, and Profinet. For **older** NanoLibs: See changelog in the imprint. At **your** risk only: legacy-system use. **Note:** Follow valid OEM instructions to set the latency as low as possible if you face problems when using an FTDI-based USB adapter.

Requirements (64-bit system mandatory)

Windows 10 or 11

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

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

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

Language, fieldbus adapters, cables

Python 3.7 to 3.13

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

2.2 Intended use and audience

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

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

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

2.3 Scope of delivery and warranty

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

- Interface classes as source code (API)
- Libraries that facilitate communication: *nanolibm_[yourfieldbus].dll* etc.
- Core functions as library in binary format: *_nanolib_python_x_x.pyd*
- Example code: *example.py*

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

3 The *NanoLib* architecture

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

User interface (API)	NanoLib core	Communication libraries
Interface and helper classes which	Libraries which	Fieldbus-specific libraries which
• access you to your controller's OD (object dictionary) • base on the NanoLib core functionalities.	• implement the API functionality • interact with bus libraries.	• do interface between NanoLib core and bus hardware.

3.1 User interface

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

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

3.2 *NanoLib* core

The *NanoLib* core comes with the 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

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

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

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

4 Getting started

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

4.1 Prepare your system

Before installing the adapter drivers, do prepare your PC along the operating system first. To prepare the PC along your Windows OS, install *Python 3.7* to *3.12* from their Website. To install *make* and *gcc* by *Linux Bash,* call `sudo apt install build-essentials`. Do then enable `CAP_NET_ADMIN`, `CAP_NET_RAW`, and `CAP_SYS_NICE` capabilities for the application that uses NanoLib:

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

4.2 Install the *Ixxat* adapter driver for Windows

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

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

4.3 Install the *Peak* adapter driver for Windows

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

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

4.4 Install the *Ixxat* adapter driver for Linux

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

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

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

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

```
unzip eci_driver_linux_amd64.zip
```

3. Install the driver via:

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

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

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

4.5 Install the *Peak* adapter driver for Linux

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

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

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

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

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

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

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

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

```
make all
```

```
sudo make install
```

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

   ```
   lsmod | grep pcan
   ```

   b) … and the shared library:

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

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

4.6 Connect your hardware

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

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

4.7 Load *NanoLib*

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

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

- Windows Setup.
- Linux Setup.

5 Windows Setup

A 64-bit system is mandatory to set up *NanoLib* with Python in Windows. **Note:**To avert conflict with similar-named products, Python's pip package is called *nanotec_nanolib_win*.

1. Install *Python ≥ 3.7* from www.python.org/.
2. Use `pip3 install wheel` to install NanoLib. Nanotec advises using *pip* and *virtual environment*.
3. In a CMD: Use `sudo apt install python3-pip python3-venv -y` to install both.
4. Set a virtual environment as follows:

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

   → The activation script name / location may **differ** as per Python version. On setup **success,** the CMD shows an *(.env)* prefix, say, *(.env) C:\test_project>*.
5. Find and extract *nanolib_python_win_N.N.N.zip* to your chosen folder, and locate the **.whl* file there.
6. In the CMD: Type `pip3 install <yourFolder>\python_win\nanotec_nanolib_win-N.N.N-py3-none-win_amd64.whl` and press *Enter.*
7. Wait for the shell to produce a success report ending on `Successfully installed nanotec-nanolib_win-N.N.N`, with *N.N.N* telling the NanoLib version.
8. To check if the installation has worked, open a CMD, if you haven't already.
9. Type `python3` and press *Enter* to open Python's shell and see something like this:

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

10. In Python: Type `import nanotec_nanolib` and press *Enter*. The installation worked if no error shows.
11. You can now leave Python by typing `exit()` and press *Enter*.

Running the example project

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

- '**_functions_example.py*' files, holding the implementations for NanoLib interface functions
- '**_callback_example.py**' files, bearing those for various callbacks (scan, data and logging)
- the *'menu_*.py'* file, which holds the menu logic and code
- the *example.py* file or main program, creating the menu and initializing all parameters used
- the *sampler_example.py* file, bearing the example implementation for sampler usage.

In a CMD: Change to the directory `<PATH_TO_EXAMPLE_FOLDER>\nanolibexample` and run the *example.py* file. The example, a CLI application, has a menu interface with context-based menu entries that are enabled or disabled as per context state and allow you to select and run various library functions along the typical workflow for handling a controller:

1. Check the PC for attached hardware (adapters) and list them.
2. Connect to one of them.
3. Scan the bus for attached controllers.
4. Connect to a device.
5. Test some library functions: Read/write from/to the controller's object dictionary; update the firmware; upload and run a *NanoJ* program; start, run and tune the motor; configure and use the logging/sampler.
6. Close the connection, *first* to the device, *then* to the adapter. **Note:** Find more motion command examples for various operation modes in nanotec.com's *Knowledge Base*.

6 Linux Setup

For a *NanoLib* setup with Python in Linux, please **note** that Python's pip package is called *nanotec_nanolib_linux* to avert conflict with similar-named products.

1. Install *Python* ≥ *3.7* from www.python.org/.
2. Use `pip3 install wheel` to install NanoLib. Nanotec advises using *pip* and *virtual environment*.
3. In a bash: Use `sudo apt install python3-pip python3-venv -y` to install both.
4. Set a virtual environment as follows:

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

   → On setup **success,** the bash shows an *(.env)* prefix, say, *(.env) username@hostname:~/test_project$*.
5. Find and extract *nanolib_python_linux_[arm64_]N.N.N.tar.gz* to your chosen folder, and locate the **.whl* file there.
6. In the bash: Type `pip3 install <yourFolder>/python_linux[_arm64]/nanotec_nanolib_linux-N.N.N-py3-none-linux_[x86_64|aarch64].whl` and press *Enter.*
7. Wait for the shell to produce a success report ending on `Successfully installed nanotec-nanolib-[x86_64|aarch64]-N.N.N`, with *N.N.N* telling the NanoLib version.
8. To check if the installation has worked, open a bash, if you haven't already.
9. Type `python3` and press *Enter* to open Python's shell and see something like this:

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

10. In Python: Type `import nanotec_nanolib` and press *Enter*. The installation worked if no error shows.
11. You can now leave Python by typing `exit()` and press *Enter*.

Running the example project

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

- *'*_functions_example.py'* files, holding the implementations for NanoLib interface functions
- *'*_callback_example.py*'* files, bearing those for various callbacks (scan, data and logging)
- the *'menu_*.py'* file, which holds the menu logic and code
- the *example.py* file or main program, creating the menu,and initializing all parameters used
- the *sampler_example.py* file, bearing the example implementation for sampler usage.

In a bash: Change to the directory `<PATH_TO_EXAMPLE_FOLDER>\nanolibexample` and run the *example.py* file. The example, a CLI application, has a menu interface with context-based menu entries that are enabled or disabled as per context state and allow you to select and run various library functions along the typical workflow for handling a controller:

1. Check the PC for attached hardware (adapters) and list them.
2. Connect to one of them.
3. Scan the bus for attached controllers.
4. Connect to a device.
5. Test some library functions: Read/write from/to the controller's object dictionary; update the firmware; upload and run a *NanoJ* program; start, run and tune the motor; configure and use the logging/sampler.
6. Close the connection, *first* to the device, *then* to the adapter. **Note:** Find more motion command examples for various operation modes in nanotec.com's *Knowledge Base*.

7 Classes / functions reference

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

ExampleFunction ()

Tells you briefly what the function does.

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 ().

NanoLibAccessor has the 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 that a void function has run.

isBusHardwareOpen ()

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

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

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

getProtocolSpecificAccessor ()

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

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

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

getProfinetDCP ()

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

```
getProfinetDCP (self)
```

Returns	ProfinetDCP

getSamplerInterface ()

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

```
getSamplerInterface (self)
```

Returns	*SamplerInterface*	Refers to the sampler interface class.

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 that a void function has run.

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

getDeviceName ()

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

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

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

getDeviceProductCode ()

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

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

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

getDeviceVendorId ()

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

```
getDeviceVendorId (self, deviceHandle)
```

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

getDeviceId ()

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

```
getDeviceId (self)
```

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

getDeviceIds ()

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

```
getDeviceIds (self)
```

Returns	*ResultDeviceIds*	Delivers a device ID list.

getDeviceUid ()

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

```
getDeviceUid (self)
```

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

getDeviceSerialNumber ()

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

```
getDeviceSerialNumber (self)
```

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

getDeviceHardwareGroup ()

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

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

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

getDeviceHardwareVersion ()

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

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

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

getDeviceFirmwareBuildId ()

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

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

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

getDeviceBootloaderVersion ()

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

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

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

getDeviceBootloaderBuildId ()

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

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

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

rebootDevice ()

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

```
rebootDevice (self, deviceHandle)
```

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

getDeviceState ()

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

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

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

setDeviceState ()

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

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

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

getConnectionState ()

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

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

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

checkConnectionState ()

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

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

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

assignObjectDictionary ()

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

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

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

autoAssignObjectDictionary ()

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

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

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

getAssignedObjectDictionary ()

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

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

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

getObjectDictionaryLibrary ()

This function returns an OdLibrary reference.

```
getObjectDictionaryLibrary (self)
```

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

setLoggingLevel ()

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

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

Parameters	*level*	The following log detailings are possible:

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

setLoggingCallback ()

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

```
setLoggingCallback(self, callback, logModule)
```

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

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

unsetLoggingCallback ()

Use this function to cancel a logging callback pointer.

```
unsetLoggingCallback (self)
```

readNumber ()

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

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

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

readNumberArray ()

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

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

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

readBytes ()

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

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

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

readString ()

Use this function to read strings from the object directory.

```
readString (self)
```

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

writeNumber ()

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

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

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

writeBytes ()

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

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

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

uploadFirmware ()

Use this function to update your controller firmware.

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

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

uploadFirmwareFromFile ()

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

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

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

uploadBootloader ()

Use this function to update your controller bootloader.

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

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

uploadBootloaderFromFile ()

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

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

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

uploadBootloaderFirmware ()

Use this function to update your controller bootloader and firmware.

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

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

uploadBootloaderFirmwareFromFile ()

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

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

Parameters	*deviceHandle*	Specifies what bus device NanoLib updates.
	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 that a void function has run.

uploadNanoJ ()

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

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

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

uploadNanoJFromFile ()

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

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

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

disconnectDevice ()

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

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

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

removeDevice ()

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

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

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

closeBusHardware ()

Use this function to disconnect from your fieldbus hardware.

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

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

7.2 BusHardwareId

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

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

Parameters

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

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 (= network name etc.).

```
getHardwareSpecifier (self)
```

Returns	*string*

getExtraHardwareSpecifier ()

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

```
getExtraHardwareSpecifier (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 ()

Returns 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 and to construct a new bus hardware option object.

addOption ()

Creates additional keys and values.

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

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

equals ()

Compares the BusHardwareOptions to existing ones.

```
equals (self, other)void addOption (String key, String value)
 {NanolibJNI.BusHardwareOptions_addOption (swigCPtr, this, key, value);}
```

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 ()

Returns 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 ()
const RESTfulBus	restfulBus = RESTfulBus()
const EtherCATBus	ethercatBus = EtherCATBus()

7.5 CanBaudRate

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

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

7.6 CanBus

Default configuration options class with the following public attributes:

string	`BAUD_RATE_OPTIONS_NAME` = "can adapter baud rate"
const CanBaudRate	*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 EtherCATBus struct

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

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

7.10 EtherCATState struct

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

string	`NONE` = "NONE"
string	`INIT` = "INIT"
string	`PRE_OPERATIONAL` = "PRE_OPERATIONAL"
string	`BOOT` = "BOOT"
string	`SAFE_OPERATIONAL` = "SAFE_OPERATIONAL"
string	`OPERATIONAL` = "OPERATIONAL"

7.11 Ixxat

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

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

7.12 IxxatAdapterBusNumber

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

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

7.13 Peak

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

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

7.14 PeakAdapterBusNumber

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

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

7.15 DeviceHandle

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

DeviceHandle ()

equals ()

Compares itself to a given device handle.

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

toString ()

Returns a string representation of the device handle.

```
toString (self)
```

7.16 DeviceId

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

- Hardware adapter identifier
- Device identifier
- Description

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

Parameters

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

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 ()

Returns the object as a string.

```
toString (self)
```

Returns	*string*

getExtraId ()

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

```
getExtraId (self)
```

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

getExtraStringId ()

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

```
getExtraStringId (self)
```

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

7.17 LogLevelConverter

This class returns your log level as a string.

```
toString (logLevel)
```

7.18 ObjectDictionary

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

getDeviceHandle ()

```
getDeviceHandle (self)
```

Returns	*ResultDeviceHandle*

getObject ()

```
getObject (self, OdIndex)
```

Returns	ResultObjectSubEntry

getObjectEntry ()

```
getObjectEntry (self, index)
```

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

getXmlFileName ()

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

readNumber ()

```
readNumber (self, OdIndex)
```

Returns	ResultInt

readNumberArray ()

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

Returns	ResultArrayInt

readString ()

```
readString (self, OdIndex)
```

Returns	ResultString

readBytes ()

```
readBytes (self, OdIndex)
```

Returns	ResultArrayByte

writeNumber ()

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

Returns	ResultVoid

writeBytes ()

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

Returns	ResultVoid

Related Links
OdIndex

7.19 ObjectEntry

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

getName ()

Reads out the name of the object as a string.

```
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:

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

```
getObjectCode (self)
```

getObjectSaveable ()

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

`APPLICATION`, `COMMUNICATION`, `DRIVE`, `MISC_CONFIG`, `MODBUS_RTU`, `NO`, `TUNING`, `CUSTOMER`, `ETHER-NET`, `CANOPEN`, `VERIFY1020`, `UNKNOWN_SAVEABLE_TYPE`

```
getObjectSaveable (self)
```

getMaxSubIndex ()

Reads out the number of subindices supported by this object.

```
getMaxSubIndex (self)
```

getSubEntry ()

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

See also ObjectSubEntry.

7.20 ObjectSubEntry

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

getName ()

Reads out the name of the object as a string.

```
getName (self)
```

getSubIndex ()

Reads out the address of the subindex.

```
getSubIndex (self)
```

getDataType ()

Reads out the data type of the object.

```
getDataType (self)
```

getSdoAccess ()

Checks if the subindex is accessible via SDO:

ReadOnly	1
WriteOnly	2
ReadWrite	3
NoAccess	0

```
getSdoAccess (self)
```

getPdoAccess ()

Checks if the subindex is accessible/mappable via PDO:

Tx	1
Rx	2
TxRx	3
No	0

```
getPdoAccess (self)
```

getBitLength ()

Checks the subindex length.

```
getBitLength (self)
```

getDefaultValueAsNumeric ()

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

```
getDefaultValueAsNumeric(self, key)
```

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)
```

7.21 OdIndex

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

getIndex ()

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

```
getIndex (self)
```

getSubindex ()

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

```
getSubIndex (self)
```

toString ()

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

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

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

```
toString (self)
```

Returns	*0xIIII:0xSS*	Default string representation

7.22 OdIndexVector

Helping class that creates a vector of OdIndex objects, to build an object dictionary.

7.23 OdLibrary

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

getObjectDictionaryCount ()

```
getObjectDictionaryCount (self)
```

getObjectDictionary ()

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

Returns	ResultObjectDictionary

addObjectDictionaryFromFile ()

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

Returns	ResultObjectDictionary

addObjectDictionary ()

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

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

Returns	ResultObjectDictionary

7.24 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:

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

```
uintToObjectCode (objectCode)
```

isNumericDataType ()

Informs if a data type is numeric or not.

```
isNumericDataType (dataType)
```

isDefstructIndex ()

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

```
isDefstructIndex (typeNum)
```

isDeftypeIndex ()

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

```
isDeftypeIndex (typeNum)
```

isComplexDataType ()

Informs if a data type is complex or not.

```
isComplexDataType (dataType)
```

uintToObjectEntryDataType ()

Converts unsigned integers to OD data type.

```
uintToObjectEntryDataType (objectDataType)
```

objectEntryDataTypeToString ()

Converts OD data type to string.

```
objectEntryDataTypeToString (odDataType)
```

stringToObjectEntryDatatype ()

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

```
stringToObjectEntryDatatype (dataTypeString)
```

objectEntryDataTypeBitLength ()

Informs on bit length of an object entry data type.

```
objectEntryDataTypeBitLength (dataType)
```

7.25 RESTfulBus struct

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

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

7.26 ProfinetDCP

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

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

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

getScanTimeout ()

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

```
getScanTimeout (self)
```

setScanTimeout ()

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

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

getResponseTimeout ()

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

```
getResponseTimeout (self)
```

setResponseTimeout ()

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

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

isServiceAvailable ()

Use this function to check Profinet DCP service availability.

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

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

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

scanProfinetDevices ()

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

```
scanProfinetDevices (self, busHardwareId)
```

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

setupProfinetDevice ()

Establishes the following device settings:

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

```
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 LED.

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

validateProfinetDeviceIp ()

Use this function to check device's IP address.

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

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

Returns	*ResultVoid*

7.27 ProfinetDevice struct

The Profinet device data have the following public attributes:

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

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

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

7.28 Result classes

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

Protected attributes	*string*	errorString
	NlcErrorCode	errorCode
	uint32_t	exErrorCode

Also, this class has the following public member functions:

hasError ()

Reads out a function call's success.

```
hasError (self)
```

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

getError ()

Reads out the reason if a function call fails.

```
getError (self)
```

Returns	*const string*

getErrorCode ()

Read the NlcErrorCode.

```
getErrorCode (self)
```

getExErrorCode ()

```
uint32_t getExErrorCode () const
```

```
getExErrorCode (self)
```

7.28.1 ResultVoid

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

7.28.2 ResultInt

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

getResult ()

Returns the integer result if a function call had success.

```
getResult (self)
```

Returns

7.28.3 ResultString

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

getResult ()

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

```
getResult (self)
```

Returns	*const string*

7.28.4 ResultArrayByte

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

getResult ()

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

```
getResult (self)
```

Returns	*const vector<uint8_t>*

7.28.5 ResultArrayInt

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

getResult ()

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

```
getResult (self)
```

Returns	*const vector<uint64_t>*

7.28.6 ResultBusHwIds

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

getResult ()

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

```
getResult (self)
```

Parameters	*const vector<BusHardwareId>*

7.28.7 ResultDeviceId

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

getResult ()

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

```
getResult (self)
```

Returns	*const vector<DeviceId>*

7.28.8 ResultDeviceIds

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

getResult ()

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

```
getResult (self)
```

Returns	*const vector<DeviceId>*

7.28.9 ResultDeviceHandle

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

getResult ()

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

```
getResult (self)
```

Returns	*DeviceHandle*

7.28.10 ResultObjectDictionary

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

getResult ()

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

```
getResult (self)
```

Returns	*const vector<ObjectDictionary>*

7.28.11 ResultConnectionState

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

getResult ()

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

```
getResult (self)
```

Returns	*DeviceConnectionStateInfo*	Connected / Disconnected / ConnectedBootloader

7.28.12 ResultObjectEntry

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

getResult ()

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

```
getResult (self)
```

Returns	*const ObjectEntry*

7.28.13 ResultObjectSubEntry

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

getResult ()

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

```
getResult (self)
```

Returns	*const ObjectSubEntry*

7.28.14 ResultProfinetDevices

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

getResult ()

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

```
getResult (self)
```

7.28.15 ResultSampleDataArray

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

getResult ()

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

```
getResult (self)
```

7.28.16 ResultSamplerState

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

getResult ()

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

```
getResult (self)
```

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

7.29 NlcErrorCode

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

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

Error code	**C: Category** │ **D: Description** │ **R: Reason**
ResourceNotFound	**C:** Common. **D:** Specified item not found. **R:** Hardware bus, protocol, device, OD address on device, or file was not found.
ResourceUnavailable	**C:** Common. **D:** Specified item not found. **R:** busy, inexistent, cut-off or defect.
OutOfMemory	**C:** Common. **D:** Insufficient memory. **R:** Too little memory to process this command.
TimeOutError	**C:** Common. **D:** Process timed out. **R:** Return after time-out expired. Timeout may be a device response time, a time to gain shared or exclusive resource access, or a time to switch the bus / device to a suitable state.

7.30 NlcCallback

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

callback ()

```
callback (self)
```

Returns	ResultVoid

7.31 NlcDataTransferCallback

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

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

The main class itself has the following public member function:

callback ()

```
callback (self)
```

Returns	ResultVoid

7.32 NlcScanBusCallback

Use this callback class for bus scanning.

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

The main class itself has the following public member function.

callback ()

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

Returns	*ResultVoid*

7.33 NlcLoggingCallback

Use this callback class for logging callbacks.

1. Define a class that extends this class with a custom callback method implementation

2. Use a pointer to its instances in order to set a callback by `NanoLibAccessor > setLoggingCallback (...)`.

```
callback (self, payload_str, formatted_str, logger_name, log_level,
 time_since_epoch, thread_id)
```

```
callback(self, payload_str, formatted_str, logger_name, log_level,
 time_since_epoch, thread_id)
```

7.34 SamplerInterface

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

configure ()

Configures a sampler.

```
configure(self, deviceHandle, samplerConfiguration)
```

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

getData ()

Gets the sampled data.

```
getData(self, deviceHandle)
```

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

getLastError ()

Gets a sampler's last error.

```
getLastError(self, deviceHandle)
```

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

getState ()

Gets a sampler's status.

```
getState(self, deviceHandle)
```

Returns	ResultSamplerState	Delivers the sampler state.

start ()

Starts a sampler.

```
start(self, deviceHandle, samplerNotify, applicationData)
```

Parameters	[in] *deviceHandle*	Specifies what device to start the sampler for.

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

stop ()

Stops a sampler.

```
stop(self, deviceHandle)
```

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

7.35 SamplerConfiguration struct

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

Public attributes

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

Static public attributes

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

7.36 SamplerNotify

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

notify ()

Delivers a notification entry.

```
notify(self, lastError, samplerState, sampleDatas, applicationData)
```

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

7.37 SampleData struct

This struct contains the sampled data.

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

7.38 SampledValue struct

This struct contains the sampled values.

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

7.39 SamplerTrigger struct

This struct contains the trigger settings of the sampler.

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

7.40 Serial struct

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

:string	`BAUD_RATE_OPTIONS_NAME` = "serial baud rate"
SerialBaudRate	*baudRate* =SerialBaudRate struct
string	`PARITY_OPTIONS_NAME` = "serial parity"
SerialParity	*parity* = SerialParity struct

7.41 SerialBaudRate struct

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"

7.42 SerialParity struct

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

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

Creative Commons

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

CC BY 3.0

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

CC BY-ND 4.0

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

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

CC BY 3.0

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

CC BY-ND 4.0

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

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

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

9 Imprint, contact, versions

Document	+ Added | > Changed | # Fixed	Product
1.3.4 2025.04	+ NanoLib Python: Added support for Python 3.13. > NanoLib Modbus: Standardize send and receive log message output for Modbus RTU, Modbus TCP and Modbus VCP. > NanoLib EtherCAT: Add send and receive log messages for CoE read / write and FoE read / write. > NanoLib UsbMsc: Standardize send and receive log message output. > NanoLib EtherCAT: remove unnecessary reboot for core5. > NanoLib Python: Change python package format for NanoLib from egg to wheel. # NanoLib RESTful: increased reboot timeout to fix an issue where controllers stay in BL after reboot. # NanoLib EtherCAT: fix issue with NanoJ upload for FW ≥ 25.08. # NanoLib EtherCAT: fix mailbox not empty error (increase EC_TIMEOUTTXM time-out). # NanoLib EtherCAT: fix issue if more than one EtherCAT slave is connected (Boot-loaderCommunicationRequests handling). # NanoLib EtherCAT: fix issue with network state after reboot (from BL). # NanoLib EtherCAT: fix issue with network state after reboot (from FW). # NanoLib Examples: Python: correct OdIndex for run_nanoj and stop_nanoj examples. # NanoLib Examples: Python: catch UnicodeEncodeError during error message handling. # NanoLib UsbMsc: increase timeout for finding MSC device after reboot. # NanoLib Modbus: Modbus VCP: fix problems with device availability. # NanoLib Modbus: Modbus VCP: fix problems serial port after reboot.	1.4.0
1.3.3 2025.01	# NanoLib-UsbMsc: fixed issue with wrong error type if accessing non-existent OD. # NanoLib-Examples: fixed reference issue in Java example (close bus hardware). # NanoLib-Modbus: fixed freeze while scanning non-Nanotec devices via Modbus-RTU.	1.3.1
1.3.2 2024.12	> Re-work of the provided examples.	1.3.0
1.3.1 2024.10	+ NanoLib Modbus: Added device locking mechanism for Modbus VCP. # NanoLib Core: Fixed connection state check. # NanoLib Code: Corrected bus hardware reference removal.	1.2.1
1.3.0 2024.09	+ NanoLib-CANopen: Support for *Peak* PCAN-USB adapter (IPEH-002021/002022).	1.2.0
1.2.3 2024.07	> NanoLib Core: Changed logging callback interface (LogLevel replaced by LogMod-ule). # NanoLib Logger: Separation between core and modules has been corrected. # Modbus TCP: Fixed firmware update for FW4. # EtherCAT: Fixed NanoJ program upload for Core5. # EtherCAT: Fixed firmware update for Core5.	1.1.3
1.2.2 2024.05	# Modbus RTU: Fixed timing issues with low baud rates during firmware update. # RESTful: Fixed NanoJ program upload.	1.1.2
1.2.1 2024.04	# NanoLib Modules Sampler: Correct reading of sampled boolean values.	1.1.1
1.2.0 2024.02	+ Java 11 support for all platforms. + Python 3.11 / 3.12 support for all platforms. + New logging callback interface (see examples). + Callback sinks for NanoLib Logger. > Update logger to version 1.12.0. > NanoLib Modules Sampler: Support now for Nanotec controller firmware v24xx. > NanoLib Modules Sampler: Change in structure used for sampler configuration.	1.1.0

Document	+ Added │ > Changed │ # Fixed	Product
	> NanoLib Modules Sampler: Continuous mode is synonymous with *endless;* the trigger condition is checked once; the number of samples must be *0.* > NanoLib Modules Sampler: Normal priority for the thread that collects data in firmware mode. > NanoLib Modules Sampler: Rewritten algorithm to detect transition between *Ready* & *Running state.* # NanoLib Core: No more *Access Violation (0xC0000005)* on closing 2 or more devices using the same bus hardware. # NanoLib Core: No more *Segmentation Fault* on attaching a PEAK adapter under Linux. # NanoLib Modules Sampler: Correct sampled-values reading in firmware mode. # NanoLib Modules Sampler: Correct configuration of 502X:04. # NanoLib Modules Sampler: Correct mixing of buffers with channels. # NanoLib-Canopen: Increased CAN timeouts for robustness and correct scanning at lower baudrates. # NanoLib-Modbus: VCP detection algorithm for special devices (USB-DA-IO).	
1.1.1 2022.09	+ EtherCAT support. # NanoLib-Modbus: *scanDevice* for ModbusTCP protocol returns an error when non-ModbusTCP devices are present on the bus.	1.0.1 (B349)
1.1.0 2022.08	+ *getDeviceHardwareGroup ()*. + *getProfinetDCP (isServiceAvailable)*. + *getProfinetDCP (validateProfinetDeviceIp)*. + *autoAssignObjectDictionary ()*. + *getXmlFileName ()*. + *const std::string & xmlFilePath* in *addObjectDictionary ()*. + *getSamplerInterface ()*. + *rebootDevice ()*. + Error code *ResourceUnavailable* for *getDeviceBootloaderVersion ()*, *~VendorId ()*, *~HardwareVersion ()*, *~SerialNumber*, and *~Uid*. > *firmwareUploadFromFile* now *uploadFirmwareFromFile ()*. > *firmwareUpload ()* now *uploadFirmware ()*. > *bootloaderUploadFromFile ()* now *uploadBootloaderFromFile ()*. > *bootloaderUpload ()* now *uploadBootloader ()*. > *bootloaderFirmwareUploadFromFile ()* to *uploadBootloaderFirmwareFromFile ()*. > *bootloaderFirmwareUpload ()* now *uploadBootloaderFirmware ()*. > *nanojUploadFromFile ()* now *uploadNanoJFromFile ()*. > *nanojUpload ()* now *uploadNanoJ ()*. > *objectDictionaryLibrary ()* now *getObjectDictionaryLibrary ()*. > NanoLib-CANopen: default settings used (*1000k* baudrate, Ixxat bus number *0*) if bus hardware options empty. > NanoLib-RESTful: admin permission obsolete for communication with Ethernet bootloaders under Windows if *npcap / winpcap* driver is available. # NanoLib-CANopen: bus hardware now opens crashless with empty options. # NanoLib-Common: *openBusHardwareWithProtocol ()* with no memory leak now.	1.0.0
1.0.2 2022.03	+ Python 3.10 / Linux ARM64 support. + USB mass storage / REST / Profinet DCP support. + *checkConnectionState ()*. + *getDeviceBootloaderVersion ()*. + *ResultProfinetDevices*. + *NlcErrorCode* (replaced *NanotecExceptions*). + NanoLib Modbus: VCP / USB hub unified to USB. # Modbus TCP scanning returns results. # Modbus TCP communication latency remains constant.	0.8.0
1.0.1 2021.11	+ More *ObjectEntryDataType* (complex and profile-specific). + *IOError* return if *connectDevice* and *scanDevices* find none. + Only 100 ms nominal timeout for CanOpen / Modbus. + *OdTypesHelper* class.	0.7.1

Document	+ Added │ > Changed │ # Fixed	Product
1.0.0 2021.06	Edition.	0.7.0