

# User Manual **NanoLib**

Java ≥ JRE /JDK 11

# Contents

<b>1 Document aim and conventions.....</b>	<b>4</b>
<b>2 Before you start.....</b>	<b>5</b>
2.1 System and hardware requirements.....	5
2.2 Intended use and audience.....	6
2.3 Scope of delivery and warranty.....	6
<b>3 The NanoLib architecture.....</b>	<b>7</b>
3.1 User interface.....	7
3.2 NanoLib core.....	7
3.3 Communication libraries.....	7
<b>4 Getting started.....</b>	<b>8</b>
4.1 Prepare your system.....	8
4.2 Install the adapter driver for Windows.....	8
4.3 Connect your hardware.....	8
4.4 Load NanoLib.....	8
<b>5 Starting the example project.....</b>	<b>9</b>
<b>6 Classes / functions reference.....</b>	<b>10</b>
6.1 NanoLibAccessor.....	10
6.2 BusHardwareId.....	20
6.3 BusHardwareOptions.....	22
6.4 BusHwOptionsDefault.....	23
6.5 CanBaudRate.....	23
6.6 CanBus.....	23
6.7 CanOpenNmtService.....	23
6.8 CanOpenNmtState.....	23
6.9 EtherCATBus struct.....	24
6.10 EtherCATState struct.....	24
6.11 Ixxat.....	25
6.12 IxxatAdapterBusNumber.....	25
6.13 DeviceHandle.....	25
6.14 DeviceId.....	25
6.15 LogLevelConverter.....	27
6.16 ObjectDictionary.....	27
6.17 ObjectEntry.....	29
6.18 ObjectSubEntry.....	30
6.19 OdIndex.....	31
6.20 OdIndexVector.....	32
6.21 OdLibrary.....	32
6.22 OdTypesHelper.....	33
6.23 RESTfulBus struct.....	34
6.24 ProfinetDCP.....	34
6.25 ProfinetDevice struct.....	36
6.26 Result classes.....	36

6.26.1	ResultVoid.....	37
6.26.2	ResultInt.....	37
6.26.3	ResultString.....	38
6.26.4	ResultArrayByte.....	38
6.26.5	ResultArrayInt.....	39
6.26.6	ResultBusHwlds.....	39
6.26.7	ResultDeviceId.....	40
6.26.8	ResultDeviceIds.....	40
6.26.9	ResultDeviceHandle.....	41
6.26.10	ResultObjectDictionary.....	41
6.26.11	ResultConnectionState.....	42
6.26.12	ResultObjectEntry.....	42
6.26.13	ResultObjectSubEntry.....	43
6.26.14	ResultProfinetDevices.....	43
6.26.15	ResultSampleDataArray.....	44
6.26.16	ResultSamplerState.....	44
6.27	NicErrorCode.....	45
6.28	NicCallback.....	46
6.29	NicDataTransferCallback.....	46
6.30	NicScanBusCallback.....	46
6.31	NicLoggingCallback.....	46
6.32	SamplerInterface.....	46
6.33	SamplerConfiguration struct.....	48
6.34	SamplerNotify.....	48
6.35	SampleData struct.....	49
6.36	SampledValue struct.....	49
6.37	SamplerTrigger struct.....	49
6.38	Serial struct.....	49
6.39	SerialBaudRate struct.....	49
6.40	SerialParity struct.....	50
<b>7</b>	<b>Licenses.....</b>	<b>51</b>
<b>8</b>	<b>Imprint, contact, versions.....</b>	<b>52</b>

## 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 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) 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 setLogLevel(LogLevel level);
```

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

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

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

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

## 2 Before you start

Before you start using NanoLib, **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!

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

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

Version	Requirements (64-bit system mandatory)	Fieldbus adapters / cables
1.1.3	Windows 10 or 11 w/ <i>Visual Studio</i> <ul style="list-style-type: none"> <li>■ CANopen: <i>Ixxat</i> VCI driver (optional)</li> <li>■ EtherCat module / Profinet DCP: <i>Npcap</i> or <i>WinPcap</i></li> <li>■ RESTful module: <i>Npcap</i>, <i>WinPcap</i>, or admin permissions to communicate w/ Ethernet bootloaders</li> </ul> Linux w/ <i>Ubuntu 18 to 24</i> (all x64 and arm64) <ul style="list-style-type: none"> <li>■ <i>Ubuntu 24.04 LTS</i></li> <li>■ <i>Ubuntu 22.04 LTS</i></li> <li>■ <i>Ubuntu 20.04 LTS</i></li> <li>■ Profinet DCP: <code>CAP_NET_ADMIN</code> and <code>CAP_NET_RAW</code> capabilities</li> <li>■ CANopen: <i>Ixxat</i> ECI driver</li> <li>■ EtherCat: <code>CAP_NET_ADMIN</code>, <code>CAP_NET_RAW</code> and <code>CAP_SYS_NICE</code> capabilities</li> <li>■ RESTful: <code>CAP_NET_ADMIN</code> capability to communicate w/ Ethernet bootloaders (also recommended: <code>CAP_NET_RAW</code>)</li> </ul>	<ul style="list-style-type: none"> <li>■ EtherCAT: <i>Ethernet cable</i></li> <li>■ VCP / USB hub: <i>now uniform USB</i></li> <li>■ USB mass storage: <i>USB cable</i></li> <li>■ REST: <i>Ethernet cable</i></li> <li>■ CANopen: <i>IXXAT USB-to-CAN V2; Nanotec ZK-USB-CAN-1</i> <b>Note:</b> No IXXAT support for <i>Ubuntu</i> on <i>arm64</i>.</li> <li>■ Modbus RTU: <i>Nanotec ZK-USB-RS485-1</i> or equivalent <i>USB-RS485</i> adapter; <i>USB cable</i> via virtual comport (VCP)</li> <li>■ Modbus TCP: <i>Ethernet cable</i> according to product datasheet</li> </ul>
1.1.2	<ul style="list-style-type: none"> <li>■ If Python: version 3.7 to 3.12</li> </ul>	See above.
1.1.1	<ul style="list-style-type: none"> <li>■ If Java: JRE / JDK 11 or higher</li> </ul>	See above.
1.1.0		
1.0.0	Windows 10 w/ <i>Visual Studio</i> <ul style="list-style-type: none"> <li>■ CANopen: <i>Ixxat</i> VCI driver (optional)</li> <li>■ EtherCat module / Profinet DCP: <i>Npcap</i> or <i>WinPcap</i></li> <li>■ RESTful module: <i>Npcap</i>, <i>WinPcap</i>, or admin permissions to communicate w/ Ethernet bootloaders</li> </ul>	See above.
1.0.0	Linux w/ <i>Ubuntu</i>	See above.

Version	Requirements (64-bit system mandatory)	Fieldbus adapters / cables
0.8.0	<ul style="list-style-type: none"> <li>■ Profinet DCP: CAP_NET_ADMIN and CAP_NET_RAW capabilities</li> <li>■ CANopen: <i>ixxat</i> ECI driver</li> <li>■ EtherCat: CAP_NET_ADMIN, CAP_NET_RAW and CAP_SYS_NICE capabilities</li> <li>■ RESTful: CAP_NET_ADMIN capability to communicate w/ Ethernet bootloaders (also recommended: CAP_NET_RAW)</li> </ul>	<ul style="list-style-type: none"> <li>■ VCP / USB hub: <i>now uniform USB</i></li> <li>■ USB mass storage: <i>USB cable</i></li> <li>■ REST: <i>Ethernet cable</i></li> </ul>
0.7.1		<ul style="list-style-type: none"> <li>■ CANopen: <i>IXXAT USB-to-CAN V2; Nanotec ZK-USB-CAN-1</i></li> <li>■ Modbus RTU: <i>Nanotec ZK-USB-RS485-1 or equivalent USB-RS485 adapter; USB cable via virtual comport (VCP)</i></li> <li>■ Modbus TCP: <i>Ethernet cable according to product datasheet</i></li> </ul>

## 2.2 Intended use and audience

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

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

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

## 2.3 Scope of delivery and warranty

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

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

## 3 The NanoLib architecture

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

User interface (API)	NanoLib core	Communication libraries
Interface and helper classes which	Libraries which	Fieldbus-specific libraries which
<ul style="list-style-type: none"> <li>■ access you to your controller's OD (object dictionary)</li> <li>■ base on the NanoLib core functionalities.</li> </ul>	<ul style="list-style-type: none"> <li>■ implement the API functionality</li> <li>■ interact with bus libraries.</li> </ul>	<ul style="list-style-type: none"> <li>■ do interface between NanoLib core and bus hardware.</li> </ul>

### 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 . 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\_ethernetcat.dll*
- *nanolibm\_usbmmisc.dll*
- *nanolibm\_modbus.dll*
- *nanolibm\_restful-api.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 Java JRE / JDK 11 or higher. 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 adapter driver for Windows

Only after due driver installation, you may use the Ixxat USB-to-CAN V2 adapter. **Note:** Other supported adapters need no driver installation. Read the USB drives' product manual, to learn if / how to activate the virtual comport (VCP).

1. Download and install the Ixxat VCI 4 driver for Windows from [www.ixxat.com](http://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 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.

### 4.4 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:
  - **For quick-and easy basics:** See [Starting the example project](#).



## 5 Starting the example project

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

- *NanolibExample.java* (main file)
- *NanolibHelper.java* (helper class for wrapping the NanoLib accessor)
- *NanolibProfinetExample.java* (optional example for PROFINET devices)
- *NanolibSamplerExample.java* (optional example for sampler use)

An example to demonstrate the logging function is in the *loggingcallbackexample* folder and usable in Windows or Linux.

### In Windows via powershell etc.

1. In the command prompt: Change to the NanoLib directory:

```

` `` cmd
  cd <desired directory>\example
` ``

```

2. With *N.N.N* as current NanoLib version: Start the java example program containing the example jar:

```

` `` cmd
  java -jar nanolib-<example name>-N.N.N.jar
` ``

```

### In Linux via Bash

1. In the bash: Change to the NanoLib directory `` `cmd cd <desired directory>/example ` ``

```

` `` cmd
  cd <desired directory>/example
` ``

```

2. With *N.N.N* as current NanoLib version: Start the java example program containing the example jar

```

` `` cmd
  java -jar nanolib-<example name>-N.N.N.jar
` ``

```

The main example demonstrates the typical workflow for handling 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 controller's object dictionary (examples provided in code ).
6. Close the connection first to the device, then to the adapter.

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

### 6.1 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.

```
ResultBusHwIds listAvailableBusHardware ()
```

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

#### openBusHardwareWithProtocol ()

Use this function to connect bus hardware.

```
ResultVoid openBusHardwareWithProtocol (BusHardwareId busHwId,  
BusHardwareOptions busHwOpt)
```

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

#### isBusHardwareOpen ()

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

```
boolean isBusHardwareOpen (BusHardwareId busHardwareId)
```

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

### getProtocolSpecificAccessor ()

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

```
ResultVoid getProtocolSpecificAccessor (BusHardwareId busHwId)
```

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

### getProfinetDCP ()

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

```
ProfinetDCP getProfinetDCP ()
```

Returns	<u>ProfinetDCP</u>
---------	--------------------

### getSamplerInterface ()

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

```
SamplerInterface getSamplerInterface ()
```

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

### setBusState ()

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

```
ResultVoid setBusState (BusHardwareId busHwId, String state)
```

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

### scanDevices ()

Use this function to scan for devices in the network.

```
ResultDeviceIds scanDevices (BusHardwareId busHwId, NlcScanBusCallback 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>	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.

```
ResultDeviceHandle addDevice (DeviceId deviceId)
```

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

**connectDevice ()**

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

```
ResultVoid connectDevice (DeviceHandle deviceHandle)
```

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

**getDeviceName ()**

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

```
ResultString getDeviceName (DeviceHandle deviceHandle)
```

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

**getDeviceProductCode ()**

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

```
ResultInt getDeviceProductCode (DeviceHandle deviceHandle)
```

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

**getDeviceVendorId ()**

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

```
ResultInt getDeviceVendorId (DeviceHandle deviceHandle)
```

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

**getDeviceId ()**

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

```
ResultDeviceId getDeviceId (DeviceHandle deviceHandle)
```

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

**getDeviceIds ()**

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

```
ResultDeviceIds getDeviceIds ()
```

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

**getDeviceUid ()**

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

```
ResultArrayByte getDeviceUid (DeviceHandle deviceHandle)
```

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

**getDeviceSerialNumber ()**

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

```
ResultString getDeviceSerialNumber (DeviceHandle deviceHandle)
```

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

**getDeviceHardwareGroup ()**

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

```
ResultInt getDeviceHardwareGroup (DeviceHandle deviceHandle)
```

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

**getDeviceHardwareVersion ()**

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

```
ResultString getDeviceHardwareVersion (DeviceHandle deviceHandle)
```

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

**getDeviceFirmwareBuildId ()**

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

```
ResultString getDeviceFirmwareBuildId (DeviceHandle deviceHandle)
```

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

**getDeviceBootloaderVersion ()**

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

```
ResultInt getDeviceBootloaderVersion (DeviceHandle deviceHandle)
```

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

### getDeviceBootloaderBuildId ()

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

```
ResultString getDeviceBootloaderBuildId (DeviceHandle deviceHandle)
```

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

### rebootDevice ()

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

```
ResultVoid rebootDevice (DeviceHandle deviceHandle)
```

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

### getDeviceState ()

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

```
ResultString getDeviceState (DeviceHandle deviceHandle)
```

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

### setDeviceState ()

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

```
ResultVoid setDeviceState (DeviceHandle deviceHandle, String state)
```

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

### getConnectionState ()

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

```
ResultConnectionState getConnectionState (DeviceHandle deviceHandle)
```

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

### checkConnectionState ()

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

```
ResultConnectionState checkConnectionState (DeviceHandle deviceHandle)
```

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

### assignObjectDictionary ()

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

```
ResultObjectDictionary assignObjectDictionary (DeviceHandle deviceHandle,  
ObjectDictionary objectDictionary)
```

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

### autoAssignObjectDictionary ()

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

```
ResultObjectDictionary autoAssignObjectDictionary (DeviceHandle deviceHandle,  
String dictionariesLocationPath)
```

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

### getAssignedObjectDictionary ()

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

```
ResultObjectDictionary getAssignedObjectDictionary (DeviceHandle  
deviceHandle)
```

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

### getObjectDictionaryLibrary ()

This function returns an OdLibrary reference.

```
OdLibrary getObjectDictionaryLibrary ()
```

Returns	<i>OdLibrary</i> &	Opens the entire OD library and its object dictionaries.
---------	--------------------	----------------------------------------------------------

### setLoggingLevel ()

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

```
void setLoggingLevel (LogLevel 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).

```
public void setLoggingCallback(NlcLoggingCallback callback, LogModule logModule)
```

Parameters *\*callback*  
*logModule*

Sets a callback pointer.  
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.

```
void unsetLoggingCallback ()
```

### readNumber ()

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

```
ResultInt readNumber (DeviceHandle deviceHandle, OdIndex odIndex)
```

Parameters *deviceHandle*  
*odIndex*

Specifies what bus device NanoLib reads from.  
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.

```
ResultArrayInt readNumberArray (DeviceHandle deviceHandle, int index)
```

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

**readBytes ()**

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

```
ResultArrayByte readBytes (DeviceHandle deviceHandle, OdIndex odIndex)
```

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

**readString ()**

Use this function to read strings from the object directory.

```
ResultString readString (DeviceHandle deviceHandle, OdIndex odIndex)
```

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

**writeNumber ()**

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

```
ResultVoid writeNumber (DeviceHandle deviceHandle, long value, OdIndex odIndex, long bitLength)
```

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

**writeBytes ()**

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

```
ResultVoid writeBytes (DeviceHandle deviceHandle, ByteVector data, OdIndex odIndex)
```

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

### uploadFirmware ()

Use this function to update your controller firmware.

```
ResultVoid uploadFirmware (DeviceHandle deviceHandle, ByteVector fwData,
  NlcDataTransferCallback callback)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib updates.
	<i>fwData</i>	Array containing firmware data.
	<i>NlcDataTransferCallback</i>	A <a href="#">data progress</a> tracer.
Returns	<i>ResultVoid</i>	Confirms that a <a href="#">void function</a> has run.

### uploadFirmwareFromFile ()

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

```
ResultVoid uploadFirmwareFromFile (DeviceHandle deviceHandle, String
  absoluteFilePath, NlcDataTransferCallback callback)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib updates.
	<i>absoluteFilePath</i>	Path to file containing firmware data (string).
	<i>NlcDataTransferCallback</i>	A <a href="#">data progress</a> tracer.
Returns	<i>ResultVoid</i>	Confirms that a <a href="#">void function</a> has run.

### uploadBootloader ()

Use this function to update your controller bootloader.

```
ResultVoid uploadBootloader (DeviceHandle deviceHandle, ByteVector btData,
  NlcDataTransferCallback callback)
```

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

### uploadBootloaderFromFile ()

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

```
ResultVoid uploadBootloaderFromFile (DeviceHandle deviceHandle, String
  bootloaderAbsolutePath, NlcDataTransferCallback callback)
```

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

### uploadBootloaderFirmware ()

Use this function to update your controller bootloader and firmware.

```
ResultVoid uploadBootloaderFirmware (DeviceHandle deviceHandle, ByteVector
  btData, ByteVector fwData, NlcDataTransferCallback callback)
```

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

	<i>btData</i>	Array containing bootloader data.
	<i>fwData</i>	Array containing firmware data.
	<i>NlcDataTransferCallback</i>	A <a href="#">data progress</a> tracer.
Returns	<i>ResultVoid</i>	Confirms that a <a href="#">void function</a> has run.

### uploadBootloaderFirmwareFromFile ()

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

```
ResultVoid uploadBootloaderFirmwareFromFile (DeviceHandle deviceHandle, String
bootloaderAbsolutePath, String absoluteFilePath, NlcDataTransferCallback
callback)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib updates.
	<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 <a href="#">data progress</a> tracer.
Returns	<i>ResultVoid</i>	Confirms that a <a href="#">void function</a> has run.

### uploadNanoJ ()

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

```
ResultVoid uploadNanoJ(DeviceHandle deviceHandle, ByteVector vmmData,
NlcDataTransferCallback callback)
```

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

### uploadNanoJFromFile ()

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

```
ResultVoid uploadNanoJFromFile (DeviceHandle deviceHandle, String
absoluteFilePath, NlcDataTransferCallback callback)
```

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

### disconnectDevice ()

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

```
ResultVoid disconnectDevice (DeviceHandle deviceHandle)
```

Parameters	<i>deviceHandle</i>	Specifies what bus device NanoLib disconnects from.
Returns	<i>ResultVoid</i>	Confirms that a <a href="#">void function</a> has run.

**removeDevice ()**

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

```
ResultVoid removeDevice (DeviceHandle deviceHandle)
```

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

**closeBusHardware ()**

Use this function to disconnect from your fieldbus hardware.

```
ResultVoid closeBusHardware (BusHardwareId busHwId)
```

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

**6.2 BusHardwareId**

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

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

**BusHardwareId () [1/3]**

Constructor that creates a new bus hardware ID object.

```
BusHardwareId (String busHardware_, String protocol_, String hardwareSpecifier_, String name_)
```

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

**BusHardwareId () [2/3]**

Constructor that creates a new bus hardware ID object, with the option for an extra hardware specifier.

```
BusHardwareId (String busHardware_, String protocol_, String hardwareSpecifier_, String extraHardwareSpecifier_, String name_)
```

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

**BusHardwareId () [3/3]**

Constructor that copies an existing *busHardwareId*.

```
nlc::BusHardwareId::BusHardwareId (BusHardwareId const &)
```

```
BusHardwareId (BusHardwareId arg0)
```

Parameters *busHardwareId* Names the bus hardware ID to copy from.

**equals ()**

Compares a new bus hardware ID to existing ones.

```
boolean equals (BusHardwareId 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.

```
String getBusHardware ()
```

Returns *string*

**getHardwareSpecifier ()**

Reads out the bus hardware's specifier string (= network name etc.).

```
String getBusHardware ()
```

Returns *string*

**getExtraHardwareSpecifier ()**

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

```
String getExtraHardwareSpecifier ()
```

Returns *string*

**getName ()**

Reads out the bus hardware's friendly name.

```
String getName ()
```

Returns *string*

**getProtocol ()**

Reads out the bus protocol string.

```
String getProtocol ()
```

Returns *string*

### toString ()

Returns the bus hardware ID as a string.

```
String toString ()
```

Returns *string*

## 6.3 BusHardwareOptions

Find in this class, in a key-value list of strings, all options needed to open a bus hardware.

### BusHardwareOptions () [1/2]

Constructs a new bus hardware option object.

```
BusHardwareOptions ()
```

Use the function [addOption \(\)](#) to add key-value pairs.

### BusHardwareOptions () [2/2]

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

```
BusHardwareOptions (StringStringMap options)
```

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

### addOption ()

Creates additional keys and values.

```
void addOption (String key, String 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.

```
boolean equals (BusHardwareOptions other)
```

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

### getOptions ()

Reads out all added key-value pairs.

```
StringStringMap getOptions ()
```

Returns *string map*

**toString ()**

Returns all keys / values as a string.

```
String toString ()
```

Returns *string*

**6.4 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()
```

**6.5 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"
```

**6.6 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      ixxat = lxxat ()
```

**6.7 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"
```

**6.8 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"

```

## 6.9 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_TIME-
OUT = "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_AT-
TEMPTS = "5"

string CHANGE_NETWORK_STATE_AT-
TEMPTS_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"

```

## 6.10 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"

```

## 6.11 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 ()

```

## 6.12 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"

```

## 6.13 DeviceHandle

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

### DeviceHandle ()

```
DeviceHandle ()
```

### equals ()

Compares itself to a given device handle.

```
boolean equals (DeviceHandle other)
```

### toString ()

Returns a string representation of the device handle.

```
String toString ()
```

## 6.14 DeviceId

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

- Hardware adapter identifier
- Device identifier
- Description

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

**DeviceId () [1/3]**

Constructs a new device ID object.

```
DeviceId (BusHardwareId busHardwareId_, long deviceId_, String description_)
```

Parameters	<i>busHardwareId_</i>	Identifier of the bus.
	<i>deviceId_</i>	An index; subject to bus (= CANopen node ID etc.).
	<i>description_</i>	A description (may be empty); subject to bus.

**DeviceId () [2/3]**

Constructs a new device ID object with extended ID options.

```
DeviceId (BusHardwareId busHardwareId_, long deviceId_, String description_,  
ByteVector extraId_, String extraStringId_)
```

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

**DeviceId () [3/3]**

Constructs a copy of a device ID object.

```
DeviceId (DeviceId arg0)
```

Parameters	<i>deviceId_</i>	Device ID to copy from.
------------	------------------	-------------------------

**equals ()**

Compares new to existing objects.

```
boolean equals (DeviceId other)
```

Returns *boolean*

**getBusHardwareId ()**

Reads out the bus hardware ID.

```
BusHardwareId getBusHardwareId ()
```

Returns BusHardwareId

**getDescription ()**

Reads out the device description (maybe unused).

```
String getDescription ()
```

Returns *string*

**getDeviceId ()**

Reads out the device ID (maybe unused).

```
long getDeviceId ()
```

Returns *unsigned int*

**toString ()**

Returns the object as a string.

```
String toString ()
```

Returns *string*

**getExtraId ()**

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

```
ByteVector getExtraId ()
```

Returns *vector extrad\_*                      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).

```
String getExtraStringId ()
```

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

**6.15 LogLevelConverter**

This class returns your log level as a string.

```
static String toString (LogLevel logLevel)
```

**6.16 ObjectDictionary**

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

**getDeviceHandle ()**

```
ResultDeviceHandle getDeviceHandle ()
```

Returns *ResultDeviceHandle*

**getObject ()**

```
ResultObjectSubEntry getObject (OdIndex odIndex)
```

Returns *ResultObjectSubEntry*

**getObjectEntry ()**

```
ResultObjectEntry getObjectEntry (int index)
```

Returns [ResultObjectEntry](#)                      Informs on an object's properties.

**getXmlFileName ()**

```
getXmlFileName (self)
```

```
ResultString getXmlFileName ()
```

Returns [ResultString](#)                      Returns the XML file name as a string.

**readNumber ()**

```
ResultInt readNumber (OdIndex odIndex)
```

Returns [ResultInt](#)

**readNumberArray ()**

```
ResultArrayInt readNumberArray (int index)
```

Returns [ResultArrayInt](#)

**readString ()**

```
ResultString readString (OdIndex odIndex)
```

Returns [ResultString](#)

**readBytes ()**

```
ResultArrayByte readBytes (OdIndex odIndex)
```

Returns [ResultArrayByte](#)

**writeNumber ()**

```
ResultVoid writeNumber (OdIndex odIndex, long value)
```

Returns [ResultVoid](#)

**writeBytes ()**

```
ResultVoid writeBytes (OdIndex odIndex, ByteVector data)
```

Returns [ResultVoid](#)

**Related Links**

[OdIndex](#)

## 6.17 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.

```
String getName ()
```

### getPrivate ()

Checks if the object is private.

```
boolean getPrivate ()
```

### getIndex ()

Reads out the address of the object index.

```
int getIndex ()
```

### getDataType ()

Reads out the data type of the object.

```
ObjectEntryDataType getDataType ()
```

### getObjectCode ()

Reads out the object code:

<b>Null</b>	0x00
<b>Deftype</b>	0x05
<b>Defstruct</b>	0x06
<b>Var</b>	0x07
<b>Array</b>	0x08
<b>Record</b>	0x09

```
ObjectCode getObjectCode ()
```

### 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, ETHERNET, CANOPEN, VERIFY1020, UNKNOWN\_SAVEABLE\_TYPE

```
ObjectSaveable getObjectSaveable ()
```

### getMaxSubIndex ()

Reads out the number of subindices supported by this object.

```
short getMaxSubIndex ()
```

**getSubEntry ()**

```
ObjectSubEntry getSubEntry (short subIndex)
```

See also [ObjectSubEntry](#).

**6.18 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.

```
String getName ()
```

**getSubIndex ()**

Reads out the address of the subindex.

```
short getSubIndex ()
```

**getDataType ()**

Reads out the data type of the object.

```
ObjectEntryDataType getDataType ()
```

**getSdoAccess ()**

Checks if the subindex is accessible via SDO:

<b>ReadOnly</b>	1
<b>WriteOnly</b>	2
<b>ReadWrite</b>	3
<b>NoAccess</b>	0

```
ObjectSdoAccessAttribute getSdoAccess ()
```

**getPdoAccess ()**

Checks if the subindex is accessible/mappable via PDO:

<b>Tx</b>	1
<b>Rx</b>	2
<b>TxRx</b>	3
<b>No</b>	0

```
ObjectPdoAccessAttribute getPdoAccess ()
```

**getBitLength ()**

Checks the subindex length.

```
long getBitLength ()
```

**getDefaultValueAsNumeric ()**

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

```
ResultInt getDefaultValueAsNumeric (String key)
```

**getDefaultValueAsString ()**

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

```
ResultString getDefaultValueAsString (String key)
```

**getDefaultValues ()**

Reads out the default values of the subindex.

```
StringStringMap getDefaultValues ()
```

**readNumber ()**

Reads out the numeric actual value of the subindex.

```
ResultInt readNumber ()
```

**readString ()**

Reads out the string actual value of the subindex.

```
ResultString readString ()
```

**readBytes ()**

Reads out the actual value of the subindex in bytes.

```
ResultByteArray readBytes ()
```

**writeNumber ()**

Writes a numeric value in the subindex.

```
ResultVoid writeNumber (long value)
```

**writeBytes ()**

Writes a value in the subindex in bytes.

```
ResultVoid writeBytes (ByteVector data)
```

## 6.19 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.

**OdIndex ()**

Constructs a new `OdIndex` object.

```
OdIndex ()
```

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).

```
int getIndex ()
```

**getSubindex ()**

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

```
short getSubIndex ()
```

**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
```

```
String toString ()
```

Returns	<code>0xIIII:0xSS</code>	Default string representation
---------	--------------------------	-------------------------------

**6.20 OdIndexVector**

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

**6.21 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 ()**

```
long getObjectDictionaryCount ()
```

**getObjectDictionary ()**

```
ResultObjectDictionary getObjectDictionary (long odIndex)
```

Returns	<code>ResultObjectDictionary</code>
---------	-------------------------------------



**addObjectDictionaryFromFile ()**

```
ResultObjectDictionary addObjectDictionaryFromFile (String
  absoluteXmlFilePath)
```

Returns [ResultObjectDictionary](#)

**addObjectDictionary ()**

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

```
ResultObjectDictionary addObjectDictionary (ByteVector odXmlData, String
  xmlFilePath)
```

Returns [ResultObjectDictionary](#)

**6.22 OdTypesHelper****uintToObjectCode ()**

Converts unsigned integers to object code:

<b>Null</b>	0x00
<b>Deftype</b>	0x05
<b>Defstruct</b>	0x06
<b>Var</b>	0x07
<b>Array</b>	0x08
<b>Record</b>	0x09

```
static ObjectCode uintToObjectCode (long objectCode)
```

**isNumericDataType ()**

Informes if a data type is numeric or not.

```
static boolean isNumericDataType (ObjectEntryDataType dataType)
```

**isDefstructIndex ()**

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

```
static boolean isDefstructIndex (int typeNum)
```

**isDeftypeIndex ()**

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

```
static boolean isDeftypeIndex (int typeNum)
```

**isComplexDataType ()**

Informes if a data type is complex or not.

```
static boolean isComplexDataType (ObjectEntryDataType dataType)
```

### uintToObjectEntryDataType ()

Converts unsigned integers to OD data type.

```
static ObjectEntryDataType uintToObjectEntryDataType
```

### objectEntryDataTypeToString ()

Converts OD data type to string.

```
static String objectEntryDataTypeToString (ObjectEntryDataType odDataType)
```

### stringToObjectEntryDatatype ()

Converts string to OD data type if possible. Otherwise, returns UNKNOWN\_DATATYPE.

```
static ObjectEntryDataType stringToObjectEntryDatatype (String dataTypeString)
```

### objectEntryDataTypeBitLength ()

Informs on bit length of an object entry data type.

```
static long objectEntryDataTypeBitLength (ObjectEntryDataType dataType)
```

## 6.23 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

## 6.24 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).

```
long getScanTimeout ()
```

**setScanTimeout ()**

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

```
void setScanTimeout (long timeoutMsec)
```

**getResponseTimeout ()**

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

```
long getResponseTimeout ()
```

**setResponseTimeout ()**

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

```
void setResponseTimeout (long 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

```
ResultVoid isServiceAvailable (BusHardwareId busHardwareId)
```

Parameters	<i>BusHardwareId</i>	Hardware ID of Profinet DCP service to check.
Returns	<i>true</i>	Service is available.
	<i>false</i>	Service is unavailable.

**scanProfinetDevices ()**

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

```
ResultProfinetDevices scanProfinetDevices (BusHardwareId busHardwareId)
```

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

**setupProfinetDevice ()**

Establishes the following device settings:

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

```
ResultVoid setupProfinetDevice (BusHardwareId busHardwareId, ProfinetDevice profinetDevice, boolean savePermanent)
```

**resetProfinetDevice ()**

Stops the device and resets it to factory defaults.

```
ResultVoid resetProfinetDevice (BusHardwareId busHardwareId, ProfinetDevice profinetDevice)
```

### blinkProfinetDevice ()

Commands the Profinet device to start blinking its Profinet LED.

```
ResultVoid blinkProfinetDevice (BusHardwareId busHardwareId, ProfinetDevice
  profinetDevice)
```

### validateProfinetDeviceIp ()

Use this function to check device's IP address.

```
ResultVoid validateProfinetDeviceIp (BusHardwareId busHardwareId,
  ProfinetDevice profinetDevice)
```

Parameters	<i>BusHardwareId</i>	Specifies the hardware ID to check.
	<i>ProfinetDevice</i>	Specifies the <i>Profinet device</i> to validate.
Returns	<i>ResultVoid</i>	

## 6.25 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

## 6.26 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	<i>string</i>	errorString
	<i>NicErrorCode</i>	errorCode
	<i>uint32_t</i>	exErrorCode

Also, this class has the following public member functions:

### hasError ()

Reads out a function call's success.

```
boolean hasError ()
```

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

**getError ()**

Reads out the reason if a function call fails.

```
String getError ()
```

Returns *const string*

**getErrorCode ()**

Read the [NlcErrorCode](#).

```
NlcErrorCode getErrorCode ()
```

**getExErrorCode ()**

```
uint32_t getExErrorCode () const
```

```
long getExErrorCode ()
```

**6.26.1 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](#) and has the following public member functions:

**ResultVoid ()**

The following functions aid in defining the exact void result:

```
ResultVoid (String errorString_)
```

```
ResultVoid (NlcErrorCode errCode, String errorString_)
```

```
ResultVoid (NlcErrorCode errCode, long exErrCode, String errorString_)
```

```
ResultVoid (Result result)
```

**6.26.2 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.

```
long getResult ()
```

Returns *long*

## ResultInt ()

The following functions aid in defining the exact integer result:

```
ResultInt (long result_)
```

```
ResultInt (String errorString_)
```

```
ResultInt (NlcErrorCode errCode, String errorString_)
```

```
ResultInt (NlcErrorCode errCode, long exErrCode, String errorString_)
```

```
ResultInt (Result result)
```

### 6.26.3 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.

```
String getResult ()
```

Returns *const string*

## ResultString ()

The following functions aid in defining the exact string result:

```
ResultString (String message, boolean hasError_)
```

```
ResultString (NlcErrorCode errCode, String errorString_)
```

```
ResultString (NlcErrorCode errCode, long exErrCode, String errorString_)
```

```
ResultString (Result result)
```

### 6.26.4 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.

```
ByteVector getResult ()
```

Returns *const vector<uint8\_t>*

## ResultArrayByte ()

The following functions aid in defining the exact byte array result:

```
ResultArrayByte (ByteVector result_)
```

```
ResultArrayByte (String errorString_)
```

```
ResultArrayByte (NlcErrorCode errCode, String errorString_)
```

```
ResultArrayByte (NlcErrorCode errCode, long exErrCode, String errorString_)
```

```
ResultArrayByte (Result result)
```

### 6.26.5 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.

```
IntVector getResult ()
```

Returns `const vector<uint64_t>`

## ResultArrayInt ()

The following functions aid in defining the exact integer array result:

```
ResultArrayInt (IntVector result_)
```

```
ResultArrayInt (String errorString_)
```

```
ResultArrayInt (NlcErrorCode errCode, String errorString_)
```

```
ResultArrayInt (NlcErrorCode errCode, long exErrCode, String errorString_)
```

```
ResultArrayInt (Result result)
```

### 6.26.6 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.

```
BusHWIdVector getResult ()
```

Parameters `const`  
`vector<BusHardwareId>`

### ResultBusHwIds ()

The following functions aid in defining the exact bus-hardware-ID-array result:

```
ResultBusHwIds (BusHWIdVector result_)
```

```
ResultBusHwIds (String errorString_)
```

```
ResultBusHwIds (NlcErrorCode errCode, String errorString_)
```

```
ResultBusHwIds (NlcErrorCode errCode, long exErrCode, String errorString_)
```

```
ResultBusHwIds (Result result)
```

### 6.26.7 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.

```
DeviceId getResult ()
```

Returns *const vector<DeviceId>*

### ResultDeviceId ()

The following functions aid in defining the exact device ID result:

```
ResultDeviceId (DeviceId result_)
```

```
ResultDeviceId (String errorString_)
```

```
ResultDeviceId (NlcErrorCode errCode, String errorString_)
```

```
ResultDeviceId (NlcErrorCode errCode, long exErrCode, String errorString_)
```

```
ResultDeviceId (Result result)
```

### 6.26.8 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.

```
DeviceIdVector getResult ()
```

Returns *const vector<DeviceId>*



### ResultDeviceIds ()

The following functions aid in defining the exact device-ID-array result:

```
ResultDeviceIds (DeviceIdVector result_)
```

```
ResultDeviceIds (String errorString_)
```

```
ResultDeviceIds (NlcErrorCode errCode, String errorString_)
```

```
ResultDeviceIds (NlcErrorCode errCode, long exErrCode, String errorString_)
```

```
ResultDeviceIds (Result result)
```

### 6.26.9 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.

```
DeviceHandle getResult ()
```

Returns *DeviceHandle*

### ResultDeviceHandle ()

The following functions aid in defining the exact device handle result:

```
ResultDeviceHandle (DeviceHandle result_)
```

```
ResultDeviceHandle (String errorString_)
```

```
ResultDeviceHandle (NlcErrorCode errCode, String errorString_)
```

```
ResultDeviceHandle (NlcErrorCode errCode, long exErrCode, String errorString_)
```

```
ResultDeviceHandle (Result result)
```

### 6.26.10 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.

```
ObjectDictionary getResult ()
```

Returns *const*  
*vector<ObjectDictionary>*

### ResultObjectDictionary ()

The following functions aid in defining the exact object dictionary result:

```
ResultObjectDictionary (ObjectDictionary result_)
```

```
ResultObjectDictionary (String errorString_)
```

```
ResultObjectDictionary (NlcErrorCode errCode, String errorString_)
```

```
ResultObjectDictionary (NlcErrorCode errCode, long exErrCode, String  
errorString_)
```

```
ResultObjectDictionary (Result result)
```

### 6.26.11 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.

```
DeviceConnectionStateInfo getResult ()
```

Returns *DeviceConnectionStateInfo* Connected / Disconnected / ConnectedBootloader

### ResultConnectionState ()

The following functions aid in defining the exact connection state result:

```
ResultConnectionState (DeviceConnectionStateInfo result_)
```

```
ResultConnectionState (String errorString_)
```

```
ResultConnectionState (NlcErrorCode errCode, String errorString_)
```

```
ResultConnectionState (NlcErrorCode errCode, long exErrCode, String  
errorString_)
```

```
ResultConnectionState (Result result)
```

### 6.26.12 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.

```
ObjectEntry getResult ()
```

Returns *const ObjectEntry*

### ResultObjectEntry ()

The following functions aid in defining the exact object entry result:

```
ResultObjectEntry (ObjectEntry result_)
```

```
ResultObjectEntry (String errorString_)
```

```
ResultObjectEntry (NlcErrorCode errCode, String errorString_)
```

```
ResultObjectEntry (NlcErrorCode errCode, long exErrCode, String errorString_)
```

```
ResultObjectEntry (Result result)
```

### 6.26.13 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.

```
ObjectSubEntry getResult ()
```

Returns *const ObjectSubEntry*

### ResultObjectSubEntry ()

The following functions aid in defining the exact object sub-entry result:

```
ResultObjectSubEntry (ObjectSubEntry result_)
```

```
ResultObjectSubEntry (String errorString_)
```

```
ResultObjectSubEntry (NlcErrorCode errCode, String errorString_)
```

```
ResultObjectSubEntry (NlcErrorCode errCode, long exErrCode, String  
errorString_)
```

```
ResultObjectSubEntry (Result result)
```

### 6.26.14 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.

```
ProfinetDeviceVector getResult ()
```

### ResultProfinetDevices ()

The following functions aid in defining the exact Profinet devices.

```
ResultProfinetDevices (ProfinetDeviceVector profinetDevices)
```

```
ResultProfinetDevices (Result result)
```

```
ResultProfinetDevices (string errorText, NlcErrorCode errorCode)
```

### 6.26.15 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.

```
SampleDataVector getResult ()
```

### ResultSampleDataArray ()

The following functions aid in defining the exact Profinet devices.

```
ResultSampleDataArray (SampleDataVector dataArray)
```

```
ResultSampleDataArray (string errorDesc, NlcErrorCode errorCode, uint  
extendedErrorCode)
```

```
ResultSampleDataArray (string errorDesc, NlcErrorCode errorCode)
```

```
ResultSampleDataArray (Result result)
```

### 6.26.16 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.

```
SamplerState getResult ()
```

Returns	<i>SamplerState</i> >	Unconfigured / Configured / Ready / Running / Completed / Failed / Cancelled
---------	-----------------------	---------------------------------------------------------------------------------

## ResultSamplerState ()

The following functions aid in defining the exact sampler state.

```
ResultSamplerState (SamplerState state)
```

```
ResultSamplerState (String errorDesc, NlcErrorCode errorCode, long  
extendedErrorCode) de)
```

```
ResultSamplerState (ResultSamplerState other)
```

```
ResultSamplerState (Result result)
```

## 6.27 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	<b>C:</b> None. <b>D:</b> No error. <b>R:</b> The operation completed successfully.
GeneralError	<b>C:</b> Unspecified. <b>D:</b> Unspecified error. <b>R:</b> Failure that fits no other category.
BusUnavailable	<b>C:</b> Bus. <b>D:</b> Hardware bus not available. <b>R:</b> Bus inexistent, cut-off or defect.
CommunicationError	<b>C:</b> Communication. <b>D:</b> Communication unreliable. <b>R:</b> Unexpected data, wrong CRC, frame or parity errors, etc.
ProtocolError	<b>C:</b> Protocol. <b>D:</b> Protocol error. <b>R:</b> Response after unsupported protocol option, device report unsupported protocol, error in the protocol (say, SDO segment sync bit), etc. <b>R:</b> A response or device report to unsupported protocol (options) or to errors in protocol (say, SDO segment sync bit), etc. <b>R:</b> Unsupported protocol (options) or error in protocol (say, SDO segment sync bit), etc.
ODDoesNotExist	<b>C:</b> Object dictionary. <b>D:</b> OD address inexistent. <b>R:</b> No such address in the object dictionary.
ODInvalidAccess	<b>C:</b> Object dictionary. <b>D:</b> Access to OD address invalid. <b>R:</b> Attempt to write a read-only, or to read from a write-only, address.
ODTypeMismatch	<b>C:</b> Object dictionary. <b>D:</b> Type mismatch. <b>R:</b> Value unconverted to specified type, say, in an attempt to treat a string as a number.
OperationAborted	<b>C:</b> Application. <b>D:</b> Process aborted. <b>R:</b> Process cut by application request. Returns only on operation interrupt by callback function, say, from bus-scanning.
OperationNotSupported	<b>C:</b> Common. <b>D:</b> Process unsupported. <b>R:</b> No hardware bus / device support.
InvalidOperation	<b>C:</b> Common. <b>D:</b> Process incorrect in current context, or invalid with current argument. <b>R:</b> 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	<b>C:</b> Common. <b>D:</b> Argument invalid. <b>R:</b> Wrong logic or syntax.
AccessDenied	<b>C:</b> Common. <b>D:</b> Access is denied. <b>R:</b> Lack of rights or capabilities to perform the requested operation.
ResourceNotFound	<b>C:</b> Common. <b>D:</b> Specified item not found. <b>R:</b> Hardware bus, protocol, device, OD address on device, or file was not found.
ResourceUnavailable	<b>C:</b> Common. <b>D:</b> Specified item not found. <b>R:</b> busy, inexistent, cut-off or defect.
OutOfMemory	<b>C:</b> Common. <b>D:</b> Insufficient memory. <b>R:</b> Too little memory to process this command.
TimeOutError	<b>C:</b> Common. <b>D:</b> Process timed out. <b>R:</b> 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.

## 6.28 NlcCallback

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

### callback ()

```
ResultVoid callback ()
```

Returns [ResultVoid](#)

## 6.29 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 ()

```
ResultVoid callback (DataTransferInfo info, int data)
```

Returns [ResultVoid](#)

## 6.30 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 ()

```
ResultVoid callback (BusScanInfo info, DeviceIdVector devicesFound, int data)
```

Returns *ResultVoid*

## 6.31 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 (...)*.

```
void callback (String payload_str, String formatted_str, String logger_name,  
long log_level, java.math.BigInteger time_since_epoch, long thread_id)
```

## 6.32 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.

```
ResultVoid configure (DeviceHandle deviceHandle, SamplerConfiguration
  samplerConfiguration)
```

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

**getData ()**

Gets the sampled data.

```
ResultSampleDataArray getData (DeviceHandle deviceHandle)
```

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

**getLastError ()**

Gets a sampler's last error.

```
ResultVoid getLastError (DeviceHandle deviceHandle)
```

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

**getState ()**

Gets a sampler's status.

```
ResultSamplerState getState (DeviceHandle deviceHandle)
```

Returns	<u>ResultSamplerState</u>	Delivers the sampler state.
---------	---------------------------	-----------------------------

**start ()**

Starts a sampler.

```
ResultVoid start (DeviceHandle deviceHandle, SamplerNotify samplerNotify, long
  applicationData)
```

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

**stop ()**

Stops a sampler.

```
ResultVoid stop (DeviceHandle deviceHandle)
```

Parameters	[in] <i>deviceHandle</i>	Specifies what device to stop the sampler for.
------------	--------------------------	------------------------------------------------

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

### 6.33 SamplerConfiguration struct

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

#### Public attributes

std::vector <OdIndex>	<i>trackedAddresses</i>	Up to 12 OD addresses to be sampled.
uint32_t	<i>version</i>	A structure's version.
uint32_t	<i>durationMilliseconds</i>	Sampling duration in ms, from 1 to 65535
uint16_t	<i>periodMilliseconds</i>	Sampling period in ms.
uint16_t	<i>numberOfSamples</i>	Samples amount.
uint16_t	<i>preTriggerNumberOfSamples</i>	Samples pre-trigger amount.
bool	<i>usingSoftwareImplementation</i>	Use software implementation.
bool	<i>usingNewFWSamplerImplementation</i>	Use FW implementation for devices with a FW version v24xx or newer.
SamplerMode	<i>mode</i>	<i>Normal</i> , <i>repetitive</i> or <i>continuous</i> sampling.
SamplerTriggerCondition	<i>triggerCondition</i>	<b>Start trigger conditions:</b> 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, <u>OdIndex</u> , <i>triggerValue</i>
SamplerTrigger	<i>SamplerTrigger</i>	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
```

### 6.34 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.

```
void notify (ResultVoid lastError, SamplerState samplerState, SampleDataVector
  sampleDatas, long applicationData)
```

Parameters [in] *lastError* Reports the last error occurred while sampling.



[in] <i>samplerState</i>	Reports the sampler status at notification time: Unconfigured / Configured / Ready / Running / Completed / Failed / Cancelled.
[in] <i>sampleDatas</i>	Reports the sampled-data array.
[in] <i>applicationData</i>	Reports application-specific data.

### 6.35 SampleData struct

This struct contains the sampled data.

<i>uin64_t iterationNumber</i>	Starts at 0 and only increases in repetitive mode.
<i>std::vector&lt;SampledValues&gt;</i>	Contains the array of sampled values.

### 6.36 SampledValue struct

This struct contains the sampled values.

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

### 6.37 SamplerTrigger struct

This struct contains the trigger settings of the sampler.

<i>SamplerTriggerCondition condition</i>	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
<i>OdIndex</i>	The trigger's <u>OdIndex</u> (address).
<i>uin32_t value</i>	Condition value or bit number (starting from bit zero).

### 6.38 Serial struct

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

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

### 6.39 SerialBaudRate struct

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

<i>string</i>	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"

## 6.40 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"

## 7 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](https://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.

## 8 Imprint, contact, versions

© 2024 Nanotec Electronic GmbH & Co. KG | Kapellenstr. 6 | 85622 Feldkirchen | Germany | Tel. +49 (0) 89 900 686-0 | Fax +49 (0) 89 900 686-50 | info@nanotec.de | www.nanotec.com | All rights reserved. Error, omission, technical or content change possible without notice. Quoted brands /products are trademarks of their owners and to be treated as such. Original version.

Document	+ Added   > Changed   # Fixed	Product
1.0.3 <sup>2024.07</sup>	<ul style="list-style-type: none"> <li>&gt; NanoLib Core: Changed logging callback interface (LogLevel replaced by LogModule).</li> <li># NanoLib Logger: Separation between core and modules has been corrected.</li> <li># Modbus TCP: Fixed firmware update for FW4.</li> <li># EtherCAT: Fixed NanoJ program upload for Core5.</li> <li># EtherCAT: Fixed firmware update for Core5.</li> </ul>	1.1.3
1.0.2 <sup>2024.05</sup>	<ul style="list-style-type: none"> <li># Modbus RTU: Fixed timing issues with low baud rates during firmware update.</li> <li># RESTful: Fixed NanoJ program upload.</li> </ul>	1.1.2
1.0.1 <sup>2024.04</sup>	<ul style="list-style-type: none"> <li># NanoLib Modules Sampler: Correct reading of sampled boolean values.</li> </ul>	1.1.1
1.0.0 <sup>2024.02</sup>	<ul style="list-style-type: none"> <li>+ Java 11 support for all platforms.</li> <li>+ Python 3.11 /3.12 support for all platforms.</li> <li>+ New logging callback interface (see examples).</li> <li>+ Callback sinks for NanoLib Logger.</li> <li>&gt; Update logger to version 1.12.0.</li> <li>&gt; NanoLib Modules Sampler: Support now for Nanotec controller firmware v24xx.</li> <li>&gt; NanoLib Modules Sampler: Change in structure used for sampler configuration.</li> <li>&gt; NanoLib Modules Sampler: Continuous mode is synonymous with <i>endless</i>; the trigger condition is checked once; the number of samples must be 0.</li> <li>&gt; NanoLib Modules Sampler: Normal priority for the thread that collects data in firmware mode.</li> <li>&gt; NanoLib Modules Sampler: Rewritten algorithm to detect transition between <i>Ready &amp; Running state</i>.</li> <li># NanoLib Core: No more <i>Access Violation (0xC0000005)</i> on closing 2 or more devices using the same bus hardware.</li> <li># NanoLib Core: No more <i>Segmentation Fault</i> on attaching a PEAK adapter under Linux.</li> <li># NanoLib Modules Sampler: Correct sampled-values reading in firmware mode.</li> <li># NanoLib Modules Sampler: Correct configuration of 502X:04.</li> <li># NanoLib Modules Sampler: Correct mixing of buffers with channels.</li> <li># NanoLib-Canopen: Increased CAN timeouts for robustness and correct scanning at lower baudrates.</li> <li># NanoLib-Modbus: VCP detection algorithm for special devices (USB-DA-IO).</li> </ul>	1.1.0