User Manual NanoLib

C#

Valid with product version 1.0.1

User Manual Version: 1.1.1

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

Underlined text marks a cross reference or hyperlink.

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

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

- Example 1: Select *File* > *New* > *Blank Document*. 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, you need to prepare your PC and inform yourself about the intended use and the library limitations.

2.1 System and hardware requirements

NOTICE

Malfunction from 32-bit operation!

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

► Follow valid OEM instructions.

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

Version	Requirements (64-bit system mandatory)	Fieldbus adapters / cables
0.7.1	■ Windows 10 w/ *Visual Studio* for *.NET* desktop: *VC++ runtimes x64*	■ CANopen: *IXXAT USB-to-CAN V2; Nanotec ZK-USB-CAN-1* ■ Modbus RTU: *Nanotec ZK-USB-RS485-1 or equivalent USB-RS485 adapter; USB cable via virtual comport (VCP)* ■ Modbus TCP: *Ethernet cable according to product datasheet*
0.8.0		■ VCP / USB hub: *now uniform USB* ■ USB mass storage: *USB cable* ■ REST: *Ethernet cable*
1.0.0	Windows 10 w/ *Visual Studio* ■ CANopen: *Ixxat* VCI driver (optional) ■ EtherCat module / Profinet DCP: *Npcap* or *WinPcap* ■ RESTful module: *Npcap, WinPcap,* or admin permissions to communicate w/ Ethernet bootloaders	■ EtherCAT: *Ethernet cable*
1.0.0	Linux w/ *Ubuntu* ■ Profinet DCP: `CAP_NET_ADMIN` and `CAP_NET_RAW` capabilities ■ CANopen: *Ixxat* ECI driver ■ EtherCat: `CAP_NET_ADMIN`, `CAP_NET_RAW` and `CAP_SYS_NICE` capabilities ■ RESTful: `CAP_NET_ADMIN` capability to communicate w/ Ethernet bootloaders (also recommended: `CAP_NET_RAW`)	■ EtherCAT: *Ethernet cable*

2.2 Intended use and audience

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

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

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

2.3 Scope of delivery and warranty

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

- Interface classes as source code (API)
- Libraries that facilitate communication by fieldbus: *nanolibm_canopen.dll*, *nanolibm_modbus.dll*, *nanolibm_restful-api.dll*, *nanolibm_usbmsc.dll* etc.
- Core functions as libraries in binary format: *nanolib_csharp*
- Example project: *NanolibExample.sln* (Visual Studio project) and *NanolibExample* (main file)

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 ■ grant access to your controller's OD (object dictionary) ■ are based on the NanoLib core functionalities.	Libraries which ■ implement the API functionality ■ interact with bus libraries.	Fieldbus-specific libraries which ■ serve as interface between NanoLib core and bus hardware.

3.1 User interface

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

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

3.2 NanoLib core

The NanoLib core comes with the library *nanolib_csharp.dll*. It implements the user interface functionality and is repsonsible for:

- Loading and managing the communication libraries.
- Providing the user interface functionalities in the NanoLibAccessor. This communication entry point defines a set of operations you can execute on the NanoLib core and communication libraries.

3.3 Communication libraries

The communication libraries provided by NanoLib (*nanolibm_canopen.dll, nanolibm_modbus.dll*) serve as hardware abstraction layer between core and controller. The core loads these libraries at startup time from the designated project folder and uses them to establish communication with the controller via the corresponding protocol.

4 Getting started

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

4.1 Prepare your system

Before installing the *Ixxat* driver, **do** prepare your PC along the operating system first. To prepare the PC along your Windows OS, install the latest *MS Visual Studio* for *.NET Desktop*.

1. Only then, install your Ixxat driver.
2. Connect the driver to the CAN-to-USB adapter.
3. Link all relevant devices to the adapter.
4. Only then, power up the devices.

4.2 Install the adapter driver for Windows

Only after due driver installation, you may use the Ixxat USB-to-CAN V2 adapter. **Note:** All other supported adapters do not require a driver installation Refer to the product manual of USB drives, to find out how to activate the virtual comport (VCP).

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

4.3 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. According to your region and needs: Download NanoLib from our website for either EMEA / APAC or AMERICA.
2. Unzip all files and folders from the NanoLib download package.

Select one option:

- **For quick-and easy basics:** See Starting the example project.
- **For advanced customizing in Windows:** See Creating your own 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 project *NanolibExample.sln* consists of:

- *Nanolib_Example.cs* (main file)
- *NanolibHelper.cs* (helper class for wrapping the NanoLib accessor)

In Windows with Visual Studio

1. Open the *NanolibExample.sln* file.
2. Open the *Nanolib_Example.cs* (main file).
3. Build the project (this will restore the nuget package).
4. Close and reopen Visual Studio.
5. Open the *Nanolib_Example.cs* again.
6. Compile and run the example code.

The example demonstrates the typical workflow for working with a controller:

1. Check the PC for connected hardware (adapters) and list them.
2. Establish connection to an adapter.
3. Scan the bus for connected controller devices.
4. Connect to a device.
5. Read/write from/to the controller's object dictionary (examples provided in code).
6. Close the connection first to the device, then to the adapter.

6 Creating your own project

Create, compile and run your own Windows project to use NanoLib.

6.1 Prepare the NuGet repository

You need a NuGet repository **before** unzipping NanoLib.

1. Create a folder for local repository, say, *C:\NugetRepo.*
2. Unzip all files and folders from *nanolib_csharp_win_###.zip.*
3. From that NanoLib unzip: Copy *Nanolib.####.nupkg* to the local repository.
4. Add the repository to *Visual Studio Tools > NuGet Package Manager > Package Sources > Add > Add your directory.*

6.2 Create a new project

Before creating a project, make *package.config* your default NuGet package format.

1. Open *Visual Studio > Tools > Options > NuGet Package Manager > General.*
2. In *Package Management:* Select *package.config* for default format.
3. Only now, go to *Open Visual Studio > Home.*
4. Select *Create new project.*
5. For project type: Select *Console App (.NET Framework) - C#* and *Next*.
6. Name the project, say, *NanolibTest* and set its location.
7. Select *Framework > .NET Framework 4.7.2.* and *Create.*
8. To add your *Nanolib NuGet Package:* Co-click your project > *Manage NuGet Packages... > Browse > Nanolib.*
9. Select the latest version and *Install.* **Note:** If you see no NanoLib package, prepare your NuGet repository (see above).
10. For a x64 target platform: Co-click your project > *Properties > Build > Platform target: x64.*

6.3 Build your project

Build your NanoLib project in MS Visual Studio.

1. Open the main ("Program.cs" in this example) and replace the text with the following code:

```
class Program
{
        static void Main(string[] args)
        {
            Nlc.NanoLibAccessor accessor = Nlc.Nanolib.getNanoLibAccessor();

        }
 }
```

2. Select *Build > Build solution.*
   → In the compile output window, there should be no error:

```
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

7 Classes / functions reference

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

ExampleFunction ()

Tells you briefly what the function does.

```
public BusHardwareId(string busHardware_, string protocol_, string
 hardwareSpecifier_, string name_)
```

Parameters	*param_a*	Additional comment if needed.
	param_b	
Returns	*ResultVoid*	Additional comment if needed.

7.1 NanoLibAccessor

Interface class used as entry point to the NanoLib. A typical workflow looks like this:

1. Start by scanning for hardware with NanoLibAccessor.listAvailableBusHardware ().
2. Set the communication settings with BusHardwareOptions ().
3. Open the hardware connection with NanoLibAccessor.openBusHardwareWithProtocol ().
4. Scan the bus for connected devices with NanoLibAccessor.scanDevices ().
5. Add a device with NanoLibAccessor.addDevice ().
6. Connect to the device with NanoLibAccessor.connectDevice ().
7. After finishing the operation, disconnect the device with NanoLibAccessor.disconnectDevice ().
8. Remove the device with NanoLibAccessor.removeDevice ().
9. Close the hardware connection with NanoLibAccessor.closeBusHardware ().
10. Familiarize yourself with the class's following public member functions:

listAvailableBusHardware ()

Use this function to list available fieldbus hardware.

```
virtual ResultBusHwIds listAvailableBusHardware ()
```

Returns	*ResultBusHwIds*	Delivers a fieldbus ID array.

openBusHardwareWithProtocol ()

Use this function to connect bus hardware.

```
virtual ResultVoid openBusHardwareWithProtocol (BusHardwareId busHwId,
 BusHardwareOptions 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.

```
virtual bool isBusHardwareOpen(BusHardwareId 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.

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

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

getProfinetDCP (isServiceAvailable)

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

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

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

Parameters	*busHwId*	Specifies the device to check *ProfinetDCP* availability for.
Returns	*ResultVoid*	Confirms that a void function has run.

getProfinetDCP (validateProfinetDeviceIp)

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

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

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

getSamplerInterface ()

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

```
virtual SamplerInterface getSamplerInterface()
```

Returns	*SamplerInterface*	Refers to the sampler interface class.

setBusState ()

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

```
virtual ResultVoid setBusState(BusHardwareId busHwId, string 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.

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

```
virtual ResultDeviceHandle addDevice (DeviceId 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*.

```
virtual ResultVoid connectDevice (Nlc.DeviceHandle deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall connect 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*.

```
virtual ResultString getDeviceName (Nlc.DeviceHandle deviceHandle)
```

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

getDeviceProductCode ()

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

```
virtual ResultInt getDeviceProductCode (Nlc.DeviceHandle deviceHandle)
```

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

getDeviceVendorId ()

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

```
virtual ResultInt getDeviceVendorId (Nlc.DeviceHandle deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall get 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.

```
virtual ResultDeviceId getDeviceId (Nlc.DeviceHandle deviceHandle)
```

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

getDeviceIds ()

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

```
virtual ResultDeviceIds getDeviceIds ()
```

Returns	*ResultDeviceIds*	Delivers a device ID list.

getDeviceUid ()

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

```
virtual ResultArrayByte getDeviceUid (Nlc.DeviceHandle deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall get 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 from the NanoLib internal list.

```
virtual ResultString getDeviceSerialNumber (Nlc.DeviceHandledeviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall get 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*.

```
virtual ResultString getDeviceHardwareGroup (Nlc.DeviceHandle deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall get 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*.

```
virtual ResultString getDeviceHardwareVersion (Nlc.DeviceHandle deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall get 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*.

```
virtual ResultString getDeviceFirmwareBuildId (Nlc.DeviceHandle deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall get 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 via *deviceHandle*.

```
virtual ResultInt getDeviceBootloaderVersion (Nlc.DeviceHandle deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall get 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 via *deviceHandle*.

```
virtual ResultString getDeviceBootloaderBuildId (Nlc.DeviceHandle
 deviceHandle)
```

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

rebootDevice ()

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

```
virtual ResultVoid nlc::NanoLibAccessor::rebootDevice (const DeviceHandle
 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.

```
virtual ResultString getDeviceState (Nlc.DeviceHandle deviceHandle)
```

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

setDeviceState ()

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

```
public virtual ResultVoid setDeviceState (Nlc.DeviceHandle deviceHandle,
 string state)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall set 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 return a specific device's last known connection state by *deviceHandle* (= *Disconnected, Connected, ConnectedBootloader*)

```
virtual ResultConnectionState getConnectionState (Nlc.DeviceHandle
 deviceHandle)
```

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

checkConnectionState ()

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

.

```
virtual ResultConnectionState checkConnectionState (Nlc.DeviceHandle
 deviceHandle)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall check 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**.

```
virtual ResultObjectDictionary assignObjectDictionary (Nlc.DeviceHandle
 deviceHandle, ObjectDictionary objectDictionary)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall assign 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.

```
virtual ResultObjectDictionary autoAssignObjectDictionary (Nlc.DeviceHandle
 deviceHandle, ObjectDictionary objectDictionary)
```

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*.

```
virtual ResultObjectDictionary getAssignedObjectDictionary (Nlc.DeviceHandle
 deviceHandle)
```

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

getObjectDictionaryLibrary ()

This function returns an OdLibrary reference.

```
virtual OdLibrary getObjectDictionaryLibrary ()
```

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*.

```
virtual void setLoggingLevel (LogLevel level)
```

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

readNumber ()

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

```
virtual ResultInt readNumber (Nlc.DeviceHandle deviceHandle, Nlc.OdIndex
 odIndex)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall read from.
	odIndex	Specifies the (sub-) index to read from.

Returns	*ResultInt*	Delivers an uninterpreted numeric value (can be signed, unsigned, fix16.16 bit values).

readNumberArray ()

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

```
virtual ResultArrayInt readNumberArray (Nlc.DeviceHandle deviceHandle, ushort
 index)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall read 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.

```
virtual ResultArrayByte nlc::NanoLibAccessor::readBytes (const DeviceHandle
 deviceHandle, const OdIndex odIndex)
```

```
virtual ResultArrayByte readBytes (Nlc.DeviceHandle deviceHandle, Nlc.OdIndex
 odIndex)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall read 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.

```
virtual ResultString readString (Nlc.DeviceHandle deviceHandle, Nlc.OdIndex
 odIndex)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall read 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.

```
virtual ResultVoid writeNumber (Nlc.DeviceHandle deviceHandle, long value,
 Nlc.OdIndex odIndex, uint bitLength)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall write 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.

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

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall write 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.

```
virtual ResultVoid uploadFirmware (Nlc.DeviceHandle deviceHandle, ByteVector
 fwData, NlcDataTransferCallback callback
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall update.
	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.

```
virtual ResultVoid uploadFirmwareFromFile (Nlc.DeviceHandle deviceHandle,
 string absoluteFilePath, NlcDataTransferCallback callback)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall update.
	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.

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

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall update.
	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.

```
virtual ResultVoid uploadBootloaderFromFile (Nlc.DeviceHandle deviceHandle,
 string bootloaderAbsoluteFilePath, NlcDataTransferCallback callback)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall update.

	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.

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

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall update.
	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.

```
virtual ResultVoid uploadBootloaderFirmwareFromFile (Nlc.DeviceHandle
 deviceHandle, string bootloaderAbsoluteFilePath, string absoluteFilePath,
 NlcDataTransferCallback callback)
```

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

uploadNanoJ ()

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

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

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

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall upload 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.

```
virtual ResultVoid uploadNanoJFromFile (Nlc.DeviceHandle deviceHandle, string
 absoluteFilePath, NlcDataTransferCallback allback)
```

Parameters	*deviceHandle*	Specifies what bus device NanoLib shall upload 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.

```
virtual ResultVoid disconnectDevice (Nlc.DeviceHandle deviceHandle)
```

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

removeDevice ()

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

```
virtual ResultVoid removeDevice (Nlc.DeviceHandle deviceHandle)
```

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

closeBusHardware ()

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

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

Parameters	*busHwId*	Specifies the fieldbus to close the connection to.
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

BusHardwareId ()

Creates a new bus hardware ID object.

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

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.

```
bool 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 (= MAC address etc.).

```
string getHardwareSpecifier ()
```

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 ()

Reads out the bus hardware ID as a string.

```
string toString ()
```

Returns	*string*

7.3 BusHardwareOptions

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

BusHardwareOptions () [1/2]

Creates a new bus hardware option object.

```
BusHardwareOptions ()
```

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

BusHardwareOptions () [2/2]

Creates 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`
	value	Example: `BAUD_RATE_1000K`

equals ()

Compares the BusHardwareOptions to existing ones.

```
bool 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 ()

Reads out all keys / values as a string.

```
string toString ()
```

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 DEFAULT_SHARED_LOCK_TIMEOUT_OPTION_NAME = "Shared Lock Timeout"	Timeout in milliseconds to acquire shared lock on the network (default = 250 ms).
const unsigned int SHARED_EXCLUSIVE_LOCK_TIMEOUT = "250"	
string READ_TIMEOUT_OPTION_NAME = "Timeout"	Timeout in milliseconds for a read operation (default = 700 ms).
const unsigned int DEFAULT_READ_TIMEOUT = "700"	
string WRITE_TIMEOUT_OPTION_NAME = "Timeout"	Timeout in milliseconds for a write operation (default = 200 ms).
const unsigned int DEFAULT_WRITE_TIMEOUT = "200"	
string READ_WRITE_ATTEMPTS_OPTION_NAME = "Read/Write Attempts"	Maximum read or write attempts (non-zero values only; default = 5).
const unsigned int DEFAULT_READ_WRITE_ATTEMPTS = "5"	
string CHANGE_NETWORK_STATE_ATTEMPTS_OPTION_NAME = "Change Network State Attempts"	Maximum number of attempts to alter the network state (non-zero values only; default = 10).
const unsigned int DEFAULT_CHANGE_NETWORK_STATE_ATTEMPTS = "10"	
string PDO_IO_ENABLED_OPTION_NAME = "PDO IO Enabled"	Enables or disables PDO processing for digital in- / outputs. ("True" or "False" only; default = "True").
string DEFAULT_PDO_IO_ENABLED = "True"	

7.10 EtherCATState Struct

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

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

7.11 Ixxat

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

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

7.12 IxxatAdapterBusNumber

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

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

7.13 DeviceHandle

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

DeviceHandle ()

```
DeviceHandle(DeviceHandle deviceHandle)
```

Returns *ResultVoid*

7.14 DeviceId

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

- Hardware adapter identifier
- Device identifier
- Description

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

DeviceId ()

Creates a new device ID object.

```
DeviceId (BusHardwareId busHardwareId_, uint deviceId_, string description_,
 const extraId_, string const extraStringId_)
```

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.

```
bool equals (DeviceId other)
```

Returns *boolean*

getBusHarwareId ()

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).

```
uint getDeviceId ()
```

Returns *unsigned int*

toString ()

Reads out the object as a string.

```
string toString ()
```

Returns *string*

getExtraId ()

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

```
string toString ()
```

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

getExtraStringId ()

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

```
string getExtraStringId ()
```

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

7.15 ObjectDictionary

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

getDeviceHandle ()

```
virtual ResultDeviceHandle getDeviceHandle ()
```

Returns *ResultDeviceHandle*

getObject ()

```
virtual ResultObjectSubEntry getObject (Nlc.OdIndex odIndex)
```

Returns *ResultObjectSubEntry*

getObjectEntry ()

```
virtual ResultObjectEntry getObjectEntry (ushort index)
```

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

getXmlFileName ()

```
virtual ResultString getXmlFileName () const
```

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

readNumber ()

```
virtual ResultInt readNumber (Nlc.OdIndex odIndex)
```

Returns	*ResultInt*

readNumberArray ()

```
virtual ResultArrayInt readNumberArray (ushort index)
```

Returns	*ResultArrayInt*

readString ()

```
virtual ResultString readString (Nlc.OdIndex odIndex)
```

Returns	*ResultString*

readBytes ()

```
virtual ResultArrayByte readBytes (Nlc.OdIndex odIndex)
```

Returns	*ResultArrayByte*

writeNumber ()

```
virtual ResultVoid writeNumber (Nlc.OdIndex odIndex, long value)
```

Returns	*ResultVoid*

writeBytes ()

```
virtual ResultVoid writeBytes (Nlc.OdIndex odIndex, ByteVector data)
```

Returns	*ResultVoid*

Related Links
OdIndex

7.16 ObjectEntry

This class represents an object entry of the object dictionary

The class has the following public member functions:

getName ()

Reads out the name of the object.

```
virtual string getName ()
```

getPrivate ()

Checks if the object is private.

```
virtual bool getPrivate ()
```

getIndex ()

Reads out the address of the object index.

```
virtual ushort getIndex ()
```

getDataType ()

Reads out the data type of the object.

```
virtual ObjectEntryDataType getDataType ()
```

getObjectCode ()

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

```
virtual ObjectCode getObjectCode ()
```

getObjectSaveable ()

Checks if the object is saveable.

```
virtual ObjectSaveable getObjectSaveable ()
```

getMaxSubIndex ()

Reads out the number of subindices supported by this object.

```
virtual byte getMaxSubIndex ()
```

getSubEntry ()

```
virtual ObjectSubEntry getSubEntry (byte subIndex)
```

See also ObjectSubEntry.

7.17 ObjectSubEntry

Class representing an object sub-entry (subindex) of the object dictionary and has the following public member functions:

getName ()

Reads out the name of the subindex.

```
virtual string getName ()
```

getSubIndex ()

Reads out the address of the subindex.

```
virtual byte getSubIndex ()
```

getDataType ()

Reads out the data type of the subindex.

```
virtual ObjectEntryDataType getDataType ()
```

getSdoAccess ()

Checks if the subindex is accessible via SDO.

```
virtual ObjectSdoAccessAttribute getSdoAccess ()
```

getPdoAccess ()

Checks if the subindex is accessible/mappable via PDO.

```
virtual ObjectPdoAccessAttribute getPdoAccess ()
```

getBitLength ()

Checks the subindex length.

```
virtual uint getBitLength ()
```

getDefaultValueAsNumeric ()

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

```
virtual ResultInt getDefaultValueAsNumeric (string key)
```

getDefaultValueAsString ()

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

```
virtual ResultString getDefaultValueAsString (string key)
```

getDefaultValues ()

Reads out the default values of the subindex.

```
virtual StringStringMap getDefaultValues ()
```

readNumber ()

Reads out the numeric actual value of the subindex.

```
virtual ResultInt readNumber ()
```

readString ()

Reads out the string actual value of the subindex.

```
virtual ResultString readString ()
```

readBytes ()

Reads out the actual value of the subindex in bytes.

```
virtual ResultArrayByte readBytes ()
```

writeNumber ()

Writes a numeric value in the subindex.

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

writeBytes ()

Writes a value in the subindex in bytes.

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

7.18 OdIndex

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

OdIndex ()

Creates a new OdIndex object.

```
OdIndex (ushort index, byte subIndex)
```

Parameters	*index*	From 0 to 65535 (0xFFFF) incl.
	subindex	From 0 to 255 (0xFF) incl.

getIndex ()

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

```
ushort Index { get; }
```

getSubindex ()

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

```
byte SubIndex { get; }
```

toString ()

Reads out the (sub-) index as a string. The string default *0xIIII:0xSS* reads as follows:

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

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

```
string ToString ()
```

Returns	*0xIIII:0xSS*	Default string representation

7.19 OdLibrary

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

getObjectDictionaryCount ()

```
virtual uint getObjectDictionaryCount ()
```

getObjectDictionary ()

```
virtual ResultObjectDictionary getObjectDictionary (uint odIndex)
```

addObjectDictionaryFromFile ()

```
virtual ResultObjectDictionary addObjectDictionaryFromFile (string
 absoluteXmlFilePath)
```

addObjectDictionary ()

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

```
virtual ResultObjectDictionary addObjectDictionary (ByteVector odXmlData)
```

7.20 OdTypesHelper

In addition to the following public member functions, this class contains custom data types. **Note:** To check your custom data types, open `public enum ObjectEntryDataType` in *ObjectEntryDataType.cs*.

uintToObjectCode ()

Converts unsigned integers to object code.

```
static ObjectCode uintToObjectCode (unsigned int objectCode)
```

isNumericDataType ()

Informs if a data type is numeric or not.

```
static bool isNumericDataType (ObjectEntryDataType dataType)
```

isDefstructIndex ()

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

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

isDeftypeIndex ()

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

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

isComplexDataType ()

Informs if a data type is complex or not.

```
static bool isComplexDataType (ObjectEntryDataType dataType)
```

uintToObjectEntryDataType ()

Converts unsigned integers to OD data type.

```
static ObjectEntryDataType uintToObjectEntryDataType (unsigned int
 objectDataType)
```

objectEntryDataTypeToString ()

Converts OD data type to string.

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

stringToObjectEntryDatatype ()

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

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

objectEntryDataTypeBitLength ()

Informs on bit length of an object entry data type.

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

7.21 RESTfulBus struct

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

const std::string	`CONNECT_TIMEOUT_OPTION_NAME` = "RESTful Connect Timeout"

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

7.22 ProfinetDCP

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

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

Under Linux, the calling application must have CAP_NET_ADMIN and CAP_NET_RAW capabilities. To enable: *sudo setcap 'cap_net_admin,cap_net_raw+eip' ./executable*
This class represents a Profinet DCP interface and has the following public member functions:

getScanTimeout ()

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

```
virtual uint getScanTimeout ()
```

setScanTimeout ()

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

```
virtual void setScanTimeout (uint timeoutMsec)
```

getResponseTimeout ()

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

```
virtual uint getResponseTimeout ()
```

setResponseTimeout ()

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

```
virtual void setResponseTimeout (uint timeoutMsec)
```

setupProfinetDevice ()

Establishes the following device settings:

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

```
virtual ResultVoid setupProfinetDevice (BusHardwareId busHardwareId,
 ProfinetDevice profinetDevice, bool savePermanent)
```

resetProfinetDevice ()

Stops the device and resets it to factory defaults.

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

blinkProfinetDevice ()

Commands the Profinet device to start blinking its Profinet LEDs.

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

7.23 ProfinetDevice

The Profinet device data, created from the *profinet_dcp.hpp* header file, have the following public attributes:

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

The MAC address is provided as array in the format: `macAddress = {0, 0, 0, 0, 0, 0};`. Whereas IP address, network mask and gateway are all interpreted as big endian hex numbers. For example:

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

7.24 Result classes

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

Protected attributes	*string*	errorString
	NlcErrorCode	errorCode
	uint32_t	exErrorCode

Also, this class has the following public member functions:

hasError ()

Reads out a function call's success.

```
bool hasError ()
```

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

getError ()

Reads out the reason if a function call fails.

```
bool getError ()
```

Returns *const string*

result ()

The following functions aid in defining the exact results:

```
Result (string errorString_)
```

```
Result (NlcErrorCode errCode, string errorString_)
```

```
Result (NlcErrorCode errCode, uint exErrCode, string errorString_)
```

```
Result (Result result)
```

getErrorCode () const

```
NlcErrorCode getErrorCode()
```

getExErrorCode () const

```
uint32_t getExErrorCode () const
```

```
uint getExErrorCode ()
```

7.24.1 ResultVoid

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

ResultVoid ()

The following functions aid in defining the exact void result:

```
ResultVoid (string errorString_)
```

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

```
ResultVoid (NlcErrorCode errCode, uint exErrCode, string errorString_)
```

```
ResultVoid (Result result)
```

7.24.2 ResultInt

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

getResult ()

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

```
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, uint exErrCode, string errorString_)
```

```
ResultInt (Result result)
```

7.24.3 ResultString

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

getResult ()

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

```
string getResult ()
```

Returns *const string*

ResultString ()

The following functions aid in defining the exact string result:

```
ResultString (string message, bool hasError_)
```

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

```
ResultString (NlcErrorCode errCode, uint exErrCode, string errorString_)
```

```
ResultString (Result result)
```

7.24.4 ResultArrayByte

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

getResult ()

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

```
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, uint exErrCode, string errorString_)
```

```
ResultArrayByte (Result result)
```

7.24.5 ResultArrayInt

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

getResult ()

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

```
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, uint exErrCode, string errorString_)
```

```
ResultArrayInt (Result result)
```

7.24.6 ResultBusHwIds

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

getResult ()

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

```
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, uint exErrCode, string errorString_)
```

```
ResultBusHwIds (Result result)
```

7.24.7 ResultDeviceId

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

getResult ()

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

```
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, uint exErrCode, string errorString_)
```

```
ResultDeviceId (Result result)
```

7.24.8 ResultDeviceIds

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

getResult ()

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

```
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, uint exErrCode, string errorString_)
```

```
ResultDeviceIds (Result result)
```

7.24.9 ResultDeviceHandle

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

getResult ()

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

```
Nlc.DeviceHandle getResult ()
```

Returns *DeviceHandle*

ResultDeviceHandle ()

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

```
ResultDeviceHandle (Nlc.DeviceHandle  result_)
```

```
ResultDeviceHandle (string errorString_)
```

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

```
ResultDeviceHandle (NlcErrorCode errCode, uint exErrCode, string errorString_)
```

```
ResultDeviceHandle (Result result)
```

7.24.10 ResultConnectionState

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

getResult ()

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

```
DeviceConnectionStateInfo getResult ()
```

Returns *DeviceHandle*

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, uint exErrCode, string
 errorString_)
```

```
ResultConnectionState (Result result)
```

7.24.11 ResultObjectDictionary

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

getResult ()

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

```
ObjectDictionary getResult ()
```

Returns *const vector<DeviceId>*

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, uint exErrCode, string
 errorString_)
```

```
ResultObjectDictionary (Result result)
```

7.24.12 ResultObjectEntry

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

getResult ()

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

```
ObjectEntry getResult ()
```

Returns *const vector<DeviceId>*

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, uint exErrCode, string errorString_)
```

```
ResultObjectEntry (Result result)
```

7.24.13 ResultObjectSubEntry

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

getResult ()

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

```
ObjectSubEntry getResult ()
```

Returns *const vector<DeviceId>*

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, uint exErrCode, string
 errorString_)
```

```
ResultObjectSubEntry (Result result)
```

7.25 NlcErrorCode

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

Error code	C: Category | D: Description | R: Reason
Success	**C:** None. **D:** No error. **R:** The operation completed successfully.
GeneralError	**C:** Unspecified. **D:** Unspecified error. **R:** Failure that fits no other category.
BusUnavailable	**C:** Bus. **D:** Hardware bus not available. **R:** Bus busy, inexistent, cut-off or defect.
CommunicationError	**C:** Communication. **D:** Communication unreliable. **R:** Unexpected data, wrong CRC, frame or parity errors, etc.
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)

Error code	C: Category | D: Description | R: Reason
	or to errors in protocol (say, SDO segment sync bit), etc. **R:** Unsupported protocol (options) or error in protocol (say, SDO segment sync bit), etc.
ODDoesNotExist	**C:** Object dictionary. **D:** OD address inexistent. **R:** No such address in the object dictionary.
ODInvalidAccess	**C:** Object dictionary. **D:** Access to OD address invalid. **R:** Attempt to write a read-only, or to read from a write-only, address.
ODTypeMismatch	**C:** Object dictionary. **D:** Type mismatch. **R:** Value unconverted to specified type, say, in an attempt to treat a string as a number.
OperationAborted	**C:** Application. **D:** Process aborted. **R:** Process cut by application request. Returns only on operation interrupt by callback function, say, from bus-scanning.
OperationNotSupported	**C:** Common. **D:** Process unsupported. **R:** No hardware bus / device support.
InvalidOperation	**C:** Common. **D:** Process incorrect in current context, or invalid with current argument. **R:** A reconnect attempt to already connected buses / devices. A disconnect attempt to already disconnected ones. A bootloader operation attempt in firmware mode or vice versa.
InvalidArguments	**C:** Common. **D:** Argument invalid. **R:** Wrong logic or syntax.
AccessDenied	**C:** Common. **D:** Access is denied. **R:** Lack of rights or capabilities to perform the requested operation.
ResourceNotFound	**C:** Common. **D:** Specified item not found. **R:** Hardware bus, protocol, device, OD address on device, or file was not found.
ResourceUnavailable	**C:** Common. **D:** Specified item not found. **R:** busy, inexistent, cut-off or defect.
OutOfMemory	**C:** Common. **D:** Insufficient memory. **R:** Too little memory to process this command.
TimeOutError	**C:** Common. **D:** Process timed out. **R:** Return after time-out expired. Timeout may be a device response time, a time to gain shared or exclusive resource access, or a time to switch the bus / device to a suitable state.

7.26 NlcCallback

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

callback ()

```
virtual ResultVoid callback ()
```

Returns *ResultVoid*

7.27 NlcDataTransferCallback

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

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

The main class itself has the following public member function:

callback ()

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

Returns *ResultVoid*

7.28 NlcScanBusCallback

Use this callback class for bus scanning.

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

The main class itself has the following public member function.

callback ()

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

Returns	*ResultVoid*	

7.29 SamplerInterface

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

configure ()

Configures a sampler.

```
virtual ResultVoid nlc::SamplerInterface::configure (const DeviceHandle
 deviceHandle, const SamplerConfiguration & samplerConfiguration)
```

Parameters	[in] *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.

```
virtual ResultSampleDataArray nlc::SamplerInterface::getData (const
 DeviceHandle 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 *samplerNotify* is active on *start ()*.

getLastError ()

Gets a sampler's last error.

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

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

getState ()

Gets a sampler's status.

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

Returns	*ResultSamplerState*	Delivers the sampler condition.

start ()

Starts a sampler.

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

Parameters	[in] *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.

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

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

7.30 SamplerConfiguration

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

Public attributes

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

Static public attributes

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

7.31 SamplerNotify

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

notify ()

Delivers a notification entry.

```
virtual void nlc::SamplerNotify::notify (const ResultVoid & lastError, const
 SamplerState samplerState, const std::vector <SampleData> & sampleDatas,
 int64_t applicationData)
```

Parameters	[in] *lastError*	Reports the last error occurred while sampling.
	[in] *samplerState*	Reports the sampler status at notification time.
	[in] *sampleDatas*	Reports the sampled-data array.
	[in] *applicationData*	Reports application-specific data.

7.32 Serial

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

:string	BAUD_RATE_OPTIONS_NAME = "serial baud rate"
SerialBaudRate	*baudRate* = SerialBaudRate ()
string	PARITY_OPTIONS_NAME = "serial parity"
SerialParity	*parity* = SerialParity ()

7.33 SerialBaudRate

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

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

7.34 SerialParity

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

string	NONE = "none"
string	ODD = "odd"

string	`EVEN` = "even"
string	`MARK` = "mark"
string	`SPACE` = "space"

8 Licenses

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

Creative Commons

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

CC BY 3.0

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

CC BY-ND 4.0

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

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

CC BY 3.0

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

CC BY-ND 4.0

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

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

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

9 Imprint, contact, versions

Nanotec Electronic GmbH & Co. KG │ Kapellenstraße 6 │ 85622 Feldkirchen │ Germany

Tel. +49 (0)89 900 686-0 │ Fax +49 (0)89 900 686-50 │ info@nanotec.de │ www.nanotec.com

Document	++ Added │ >> Changed │ ## Fixed	Product
1.0.0 2021.06	Edition	0.7.0
1.0.1 2021.11	■ **++** 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
1.0.2 2022.03	■ **++** USB mass storage / REST / Profinet DCP support added ■ ++ NanoLib Modbus: VCP / USB hub unified to USB ■ ++ *checkConnectionState ()* ■ ++ *getDeviceBootloaderVersion ()* ■ ++ *ResultProfinetDevices* ■ ++ *NlcErrorCode* (replaced *NanotecExceptions*) ■ ## Modbus TCP scanning returns results ■ ## Modbus TCP communication latency remains constant	0.8.0
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* ■ **>>** *firrmwareUploadFromFile* now *uploadFirmwareFromFile ()* ■ >> *firmwareUpload ()* now *uploadFirmware ()* ■ >> *bootloaderUploadFromFile ()* now *uploadBootloaderFromFile ()* ■ >> *bootloaderUpload ()* now *uploadBootloader ()* ■ >> *bootloaderFirmwareUploadFromFile ()* to *uploadBootloaderFirmwareFromFile ()* ■ >> *bootloaderFirmwareUpload ()* now *uploadBootloaderFirmware ()* ■ >> *nanojUploadFromFile ()* now *uploadNanoJFromFile ()* ■ >> *nanojUpload ()* now *uploadNanoJ ()* ■ >> *objectDictionaryLibrary ()* tnow *getObjectDictionaryLibrary ()* ■ >> *String_String_Map* now *StringStringMap* ■ >> Nanolib-Common: faster execution of *listAvailableBusHardware* and *openBusHardwareWithProtocol* with Ixxat adapter ■ >> 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

Document	++ Added | >> Changed | ## Fixed	Product
1.1.1 2022.09	■ ++ EtherCAT support	1.0.1 (B349)