Programming Reference Guide

netX Diagnostic and Remote Access

netXTransport Toolkit

V1.0

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC130704PRG02EN | Revision 2 | English | 2013-09 | Released | Public

Table of Contents

1 Introduction

1.1 About this Document

This manual describes the *netXTransport Toolkit*, the internal structure, handling and how to port the toolkit. The *netXTransport Toolkit* consists of C-source and header files allowing to abstract operating system dependent code. The outsourced operating system dependent code is reduced to a defined amount of functions and needs to be implemented by the user (developer).

netXTransport Toolkit

The '*netXTransport Toolkit*' provides a standard diagnostic interface for netX based systems via common physical connections (serial, USB, Ethernet) in combination with standard access functions (e.g. cifX API). The netXTransport communication principle is based on a client server model using the Hilscher Transport Protocol '*HilTransport*' (for more information refer to the *Hilscher Gesellschaft für Systemautomation mbH: Programming Reference Guide, netX Diagnostic and Remote Access, Fundamentals* documentation, reference [1]).

1.1.1 List of Revisions

Rev	Date	Name	Chapter	Revision
1	2013-07-08	SD	all	Created
2	2013-09-04	HH	5.2	Sequence of sub section according to Table 13.

Table 1: List of Revisions

1.1.2 Terms, Abbreviations and Definitions

Term	Description
AP (-task)	Application (-task) on top of the stack
ARP	Address Resolution Protocol
BOOTP	Bootstrap Protocol
DHCP	Dynamic Host Configuration Protocol
ICMP	Internet Control Message Protocol
IP	Internet Protocol
MSS	Maximum segment size (of TCP data), normally = 1460 byte on Ethernet (Maximum); MSS = MTU - sizeof(IP header) - sizeof (TCP header) = 1500 –20 –20 = 1460
MTU	Maximum Transmission Unit, normally 1500 byte = Data part of Ethernet frame
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Table 2: Terms, Abbreviations and Definitions

All variables, parameters, and data used in this manual have the LSB/MSB (“Intel”) data format. This corresponds to the convention of the Microsoft C Compiler.

All IP addresses in this document have host byte order.

1.2 References to Documents

This document refers to the following documents:

[1] Hilscher Gesellschaft für Systemautomation mbH: Programming Reference Guide, netX Diagnostic and Remote Access, **Fundamentals**, revision 2, english, 2011.

[2] Hilscher Gesellschaft für Systemautomation mbH: Programming Reference Guide, netX Diagnostic and Remote Access, **Host Device**, revision 3, english, 2013.

[3] Hilscher Gesellschaft für Systemautomation mbH: Programming Reference Guide, netX Diagnostic and Remote Access, **Target Device**, revision 2, english, 2011.

[4] Hilscher Gesellschaft für Systemautomation mbH: Programming Reference Guide, **CIFX API** (Application Programming Interface), revision 2, english, 2013.

Table 3: References to Documents

1.3 netXTransport Toolkit

1.3.1 netXTransport Folder Structure

Folder		Description	
netXTransport Toolkit			
Toolkit	Source	C-source and header files of the netXTransport Toolkit (Operating System independent source)	
	OSAbstraction	Example include and source file for OS specific implementation	
	User	Example source file for User function implementation	
	Doc / doxyfile	Miscellaneous files to create an internal doxygen documentation	
Examples	Linux	Example application for Linux. This example creates a library from the netXTransport toolkit and offers a demo application using this library	
		libnetXTransport	Library project for the netX Transport toolkit
		Connectors	Example source for a TCP connector including configuration
	Win32	Example Application for Windows (cifX API) This example creates a netXTransport DLL (Dynamic Link Library) from the netXTransport toolkit and offers a demo application which uses the DLL.	
		netXTransportDLL	Example project to create a netXTransport DLL
		Connectors	Example source for an TCP and serial (RS232/USB) connector

Table 4: Folder Structure

1.3.2 Features

Option:

Little Endian / Big Endian support (selectable via toolkit definition)

1.3.3 Restrictions

- Several functions must be implemented by the user, before being able to use the toolkit
- Multithreading support required for cyclic jobs
- Currently no customer API support (only cifX API incl. cifX-Marshaller/rcX-Packets)
- Target requires a running netXTransport Host-Application

1.4 Legal Notes

1.4.1 Copyright

1.4.2 Important Notes

The user manual, accompanying texts and the documentation were created for the use of the products by qualified experts, however, errors cannot be ruled out. For this reason, no guarantee can be made and neither juristic responsibility for erroneous information nor any liability can be assumed. Descriptions, accompanying texts and documentation included in the user manual do not present a guarantee nor any information about proper use as stipulated in the contract or a warranted feature. It cannot be ruled out that the user manual, the accompanying texts and the documentation do not correspond exactly to the described features, standards or other data of the delivered product. No warranty or guarantee regarding the correctness or accuracy of the information is assumed.

We reserve the right to change our products and their specification as well as related user manuals, accompanying texts and documentation at all times and without advance notice, without obligation to report the change. Changes will be included in future manuals and do not constitute any obligations. There is no entitlement to revisions of delivered documents. The manual delivered with the product applies.

Hilscher Gesellschaft für Systemautomation mbH is not liable under any circumstances for direct, indirect, incidental or follow-on damage or loss of earnings resulting from the use of the information contained in this publication.

1.4.3 Exclusion of Liability

The software was produced and tested with utmost care by Hilscher Gesellschaft für Systemautomation mbH and is made available as is. No warranty can be assumed for the performance and flawlessness of the software for all usage conditions and cases and for the results produced when utilized by the user. Liability for any damages that may result from the use of the hardware or software or related documents, is limited to cases of intent or grossly negligent violation of significant contractual obligations. Indemnity claims for the violation of significant contractual obligations are limited to damages that are foreseeable and typical for this type of contract.

It is strictly prohibited to use the software in the following areas:

- for military purposes or in weapon systems;
- for the design, construction, maintenance or operation of nuclear facilities;
- in air traffic control systems, air traffic or air traffic communication systems;
- in life support systems;
- in systems in which failures in the software could lead to personal injury or injuries leading to death.

We inform you that the software was not developed for use in dangerous environments requiring fail-proof control mechanisms. Use of the software in such an environment occurs at your own risk. No liability is assumed for damages or losses due to unauthorized use.

1.4.4 Export

The delivered product (including the technical data) is subject to export or import laws as well as the associated regulations of different counters, in particular those of Germany and the USA. The software may not be exported to countries where this is prohibited by the United States Export Administration Act and its additional provisions. You are obligated to comply with the regulations at your personal responsibility. We wish to inform you that you may require permission from state authorities to export, re-export or import the product.

2 Overview

2.1 netXTransport Toolkit

Goal of the '*netXTransport Toolkit*' is to provide a standard diagnostic interface for netX based systems via common physical connections (e.g. serial, USB, Ethernet) in combination with a variety of access functions (like cifX API functions, rcX data packages or custom defined API functions). To guarantee such a flexibility the '*netXTransport Toolkit*' offers appropriate interfaces to register additional connection and API types. This connection independent communication extends the field of application for runtime data access and data exchange between the host and the target.

The operating system independent design of the toolkit simplifies porting to different systems. A high flexibility, regarding physical connections, is given by the possibility of adding additional custom defined connector types during runtime. Furthermore the *netXTransport Toolkit* is capable to support different API interfaces. This feature enables support for different devices with different API interfaces.

The netXTransport communication is based on a server client concept using the *HilTransport Protocol*, and therefore the target requires a running netXTransport server application. For more information see reference [1].

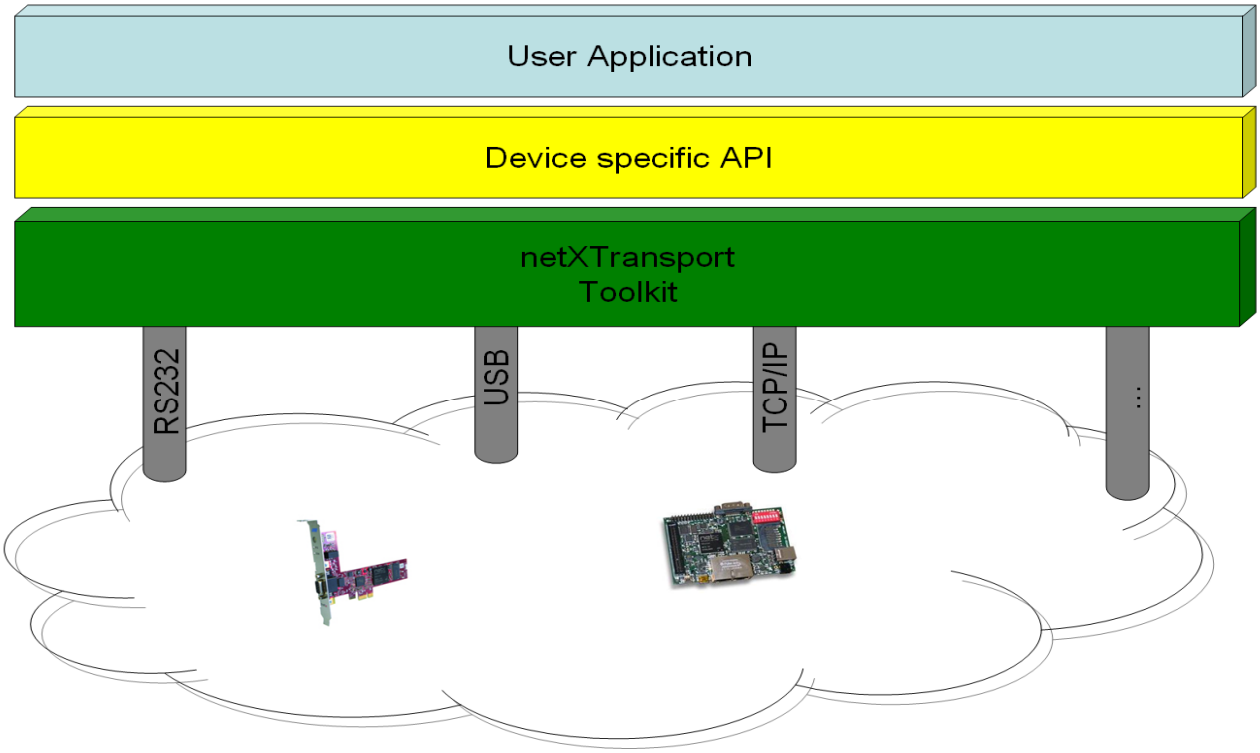

Figure 1: netXTransport Toolkit: Application Overview

2.2 netXTransport Application Overview

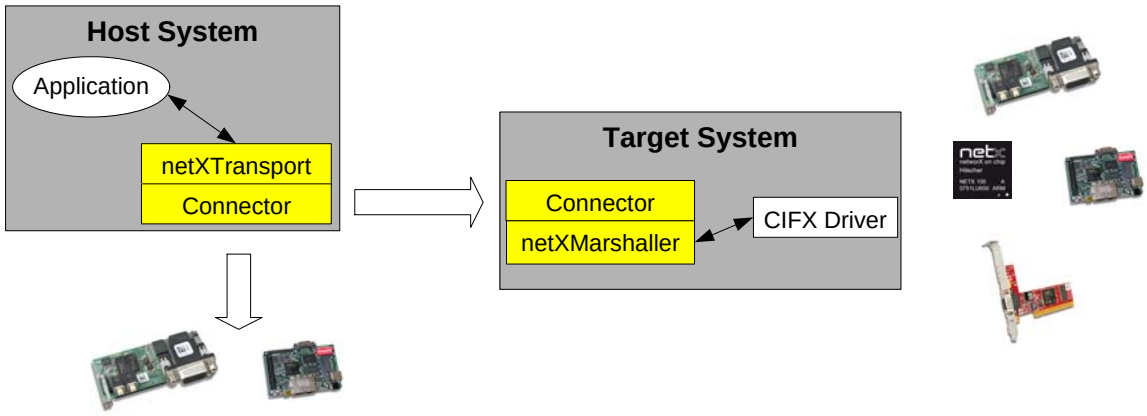

Figure 2: netXTransport Application Overview

2.3 netXTransport Toolkit Internal Layout

To provide the flexibility of handling different devices connected over different interfaces, the *netXTransport Toolkit* is divided in several layers (see Figure 3). The Translation Layer serves different API interfaces and manages the translation between a device specific API function call and the generic transport layer interface and vice versa. The transport layer coordinates the communication stream between a connector and the translation layer (TL) and handles the *HilTransport Protocol* related issues. The connector layer abstracts the different connector types and standardizes the handling and the data format.

Figure 3 shows the *netXTransport Toolkit* internal layout. The green marked block is hidden by the *netXTransport Toolkit*. The connector layer needs to be implemented by the user (for already implemented connectors or example connectors see *Table 4: Folder Structure* on page 6).

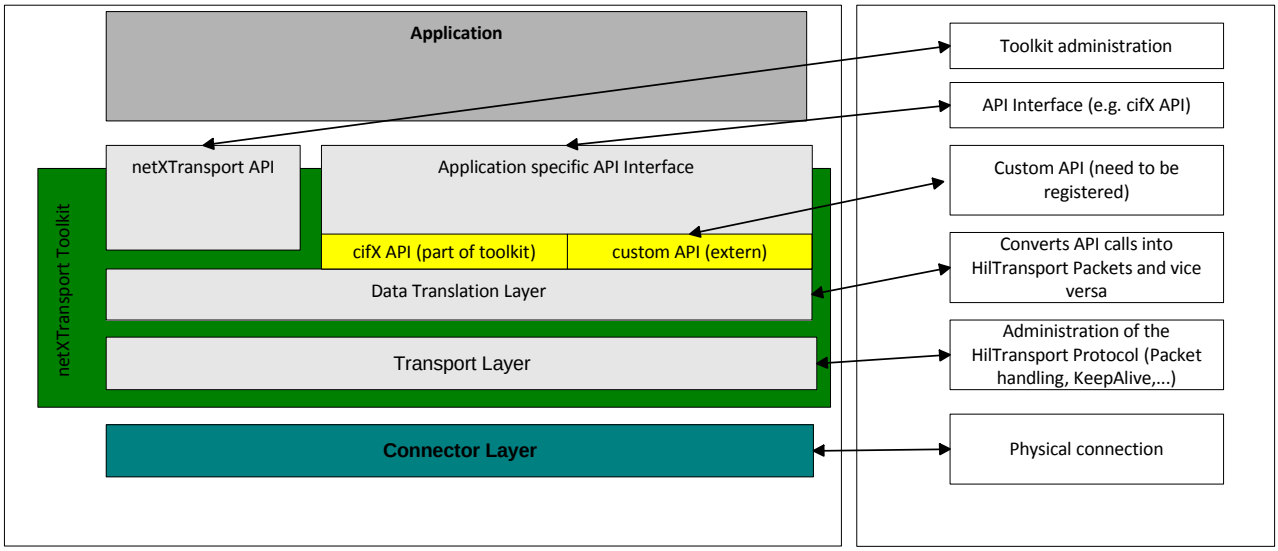

Figure 3: netXTransport Layer Model

For a detailed view of the dataflow see Figure 4 on page 12.

netXTransport API / netXTransport Layer

This module provides the core functionality, managing the initialization and the handling of all the underlying layers. The API interface of the netXTransport module serves the functions to add and initialize modules, like connector instances and different API types. For more information see section *Toolkit API* on page 30.

Data Translation Layer

As noted under *netXTransport* on page 9, the *netXTransport Toolkit* can support different devices with different API interfaces. This feature requires an abstraction layer hiding the device specific information and properties from the *netXTransport Layer*. The *Translation Layer* creates an API specific device, based on the netXTransport generic device and converts the device specific API calls into generic HilTransport packets and vice versa.

Transport Layer

The *Transport Layer* handles all communication related issues. Further it coordinates the communication stream between the *Connector Layer* and *Translation Layer*. It provides the *HilTransport Protocol* interpreter and manages the data send and receive requests. For more information see *Hilscher Transport Mechanism* in documentation reference [1].

Connector Layer (not part of the toolkit)

The *Connector Layer* handles the different registered connector types and controls the raw data in and out put. A *netXTransport Connector* is not part of the toolkit and needs to be implemented by the user. The *netXTransport Toolkit* provides a function to register a connector during runtime. For more information how to implement a *netXTransport Connector* see section *Creating a netXTransport Connector* on page 19.

Example connectors are available for Windows and Linux see *Table 4: Folder Structure* on page 6.

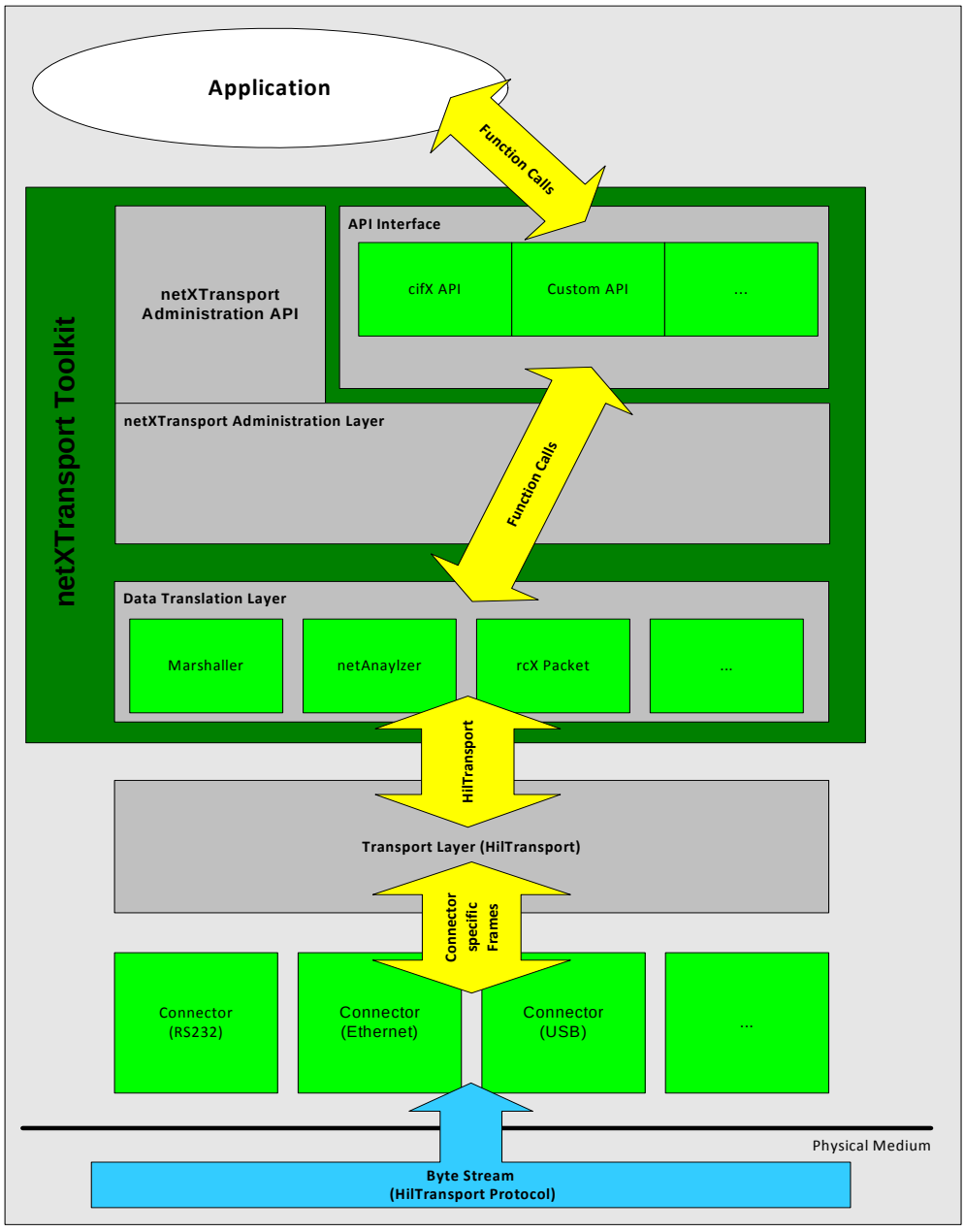

Figure 4: netXTransport Data Flow

2.4 netXTransport Additional Information

This section gives a short introduction to some terms used in this document.

2.4.1 netXTransport Generic Device

Independent of the device type all detected devices are abstracted by the *netXTransport Generic Device* (see section *NETX_TRANSPORT_GENERIC_DEVICE_T Structure* on page 54). This abstraction allows a unified device handling. Beneath the generic information, the NETX_TRANSPORT_DEVICE_T structure contains a reference to the device specific information generated by its corresponding Translation Layer module (see section *Translation Layer Information* on page 14).

2.4.2 netXTransport Device Grouping

The following figure shows how the *netXTransport Toolkit* internally categorizes the *netXTransport Generic Devices* in respect to their related types and interfaces.

The *netXTransport Device Group* is defined by the API type and therefore tied to a fixed translation layer (see section *NETX_TRANSPORT_DEV_GROUP_T Structure* on page 53). It contains the list of all devices matching the corresponding API type. The *netXTransport Device* structure (see section *NETX_TRANSPORT_GENERIC_DEVICE_T Structure* on page 54) contains, among other information, a link to the interface (see section *NETX_TRANSPORT_INTERFACE_T Structure* on page 55) it is related to. All of these components are gathered in the global *netXTransport Data Structure* (see section *NETX_TRANSPORT_DATA_T Structure* on page 53) and are administrated by the netXTransport layer.

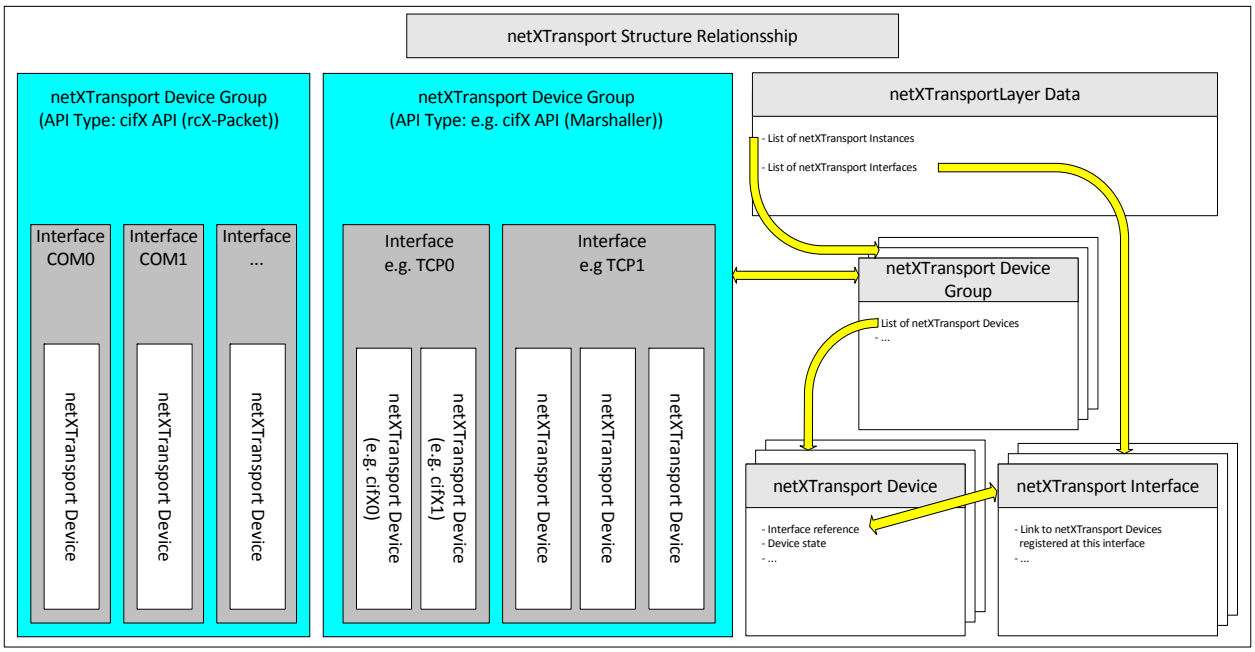

Figure 5: netXTransport Structure Relationship

2.4.3 Translation Layer Information

The *Translation Layer* needs to provide detailed knowledge of a device to be able to handle the translation between the device specific API and the generic *Transport Layer* (see section *netXTransport Toolkit Internal Layout*). The Translation Layer information is not specified and depends on the device type. However it consists always of two types:

- Interface related information (unique per connection) generated during the Device discovering process (see *TL_Probe()*, Figure 8 on page 67).
- Device related information generated during the device discovering process (see *TL_Scan()*, Figure 8 on page 67) (unique per device and might not differ from interface related information in case of single endpoint connection).

An example of connection related information can be found in the 'cifX API Marshaller Translation Layer' (*MARSHALLER_TL_T* defined in *TL_Marshaller.h* (netXTransport Toolkit)).

Device related information can be found cifX API Marshaller Translation Layer (MARSHALLER_ENDPOINT_DATA_T defined in TL_Marshaller.h (netXTransport Toolkit)).

2.4.4 Transport Layer Information

Contains all connection related information, like the connector type information, interface information and transaction information. For details refer to the *HIL_TRANSPORT_DATA_T* structure defined in *HilTransport.c* (netXTransport Toolkit).

2.4.5 Connection Types

The "Host to Target" connections can be differentiated in two types:

- Multiple Endpoint Connection (netXTransport server application is running on the target).

  The target may consist of more than one device (e.g. Target is a PC connected over TCP, sharing more than one device).
- Single Endpoint Connection

2.4.6 Device Naming Convention

Within the netXTransport toolkit all detected devices are differentiated by their name (see section *Internal Structures and Functions* on page 53).

The device name creation is a generic process and a combination of the interface name and its standard device name, where the standard device name is delivered by the device specific translation layer.

Device Name Length: max. 16 Characters

Name Delimitier: **"_"** (underline)

Device Name Structure:

[*Interface name*][*interface index*]_[*device standard name*][*device index*]

Example of a cifX Devices and Names:

- **RS323 connector** (e.g. single device / single endpoint connection):
  "COM0" + "cifX0" (device name) -> "COM0_cifX0"
  "COM1" + "cifX0" (device name) -> "COM1_cifX0"
- **TCP connector** (e.g. multiple device / multiple endpoint connection type):
  "TCP0" + "cifX0" (device name) -> "TCP0_cifX0"
  "TCP0" + "cifX1" (device name) -> "TCP0_cifX1"
  "TCP1" + "cifX0" (device name) -> "TCP1_cifX0"

3 How to port the netXTransport Toolkit

This is a short instruction on how to port the *netXTransport Toolkit* to an own system. In general the toolkit is independent of any operating system.

Some example implementations are available (Windows/Linux) showing the work necessary to port the toolkit to an own hardware platform (see section *Folder Structure* on page 6, '*netXTransport Toolkit Adaptation*').

Porting the netXTransport Toolkit consists of two basic steps

1. **Porting the Toolkit** (described in section *Porting the netXTransport Toolkit* on page 16)

   Available implementations can be found in the Folder *Structure* on page 6, '*netXTransport Toolkit Adaptation*'.
2. **Creating a Connector** (described in *Creating a netXTransport Connector* on page 19)

   Available implementations can be found in the *Folder Structure* on page 6, '*Connector*'.

3.1 Porting the netXTransport Toolkit

The netXTransport Toolkit consists of three parts:

- **Operating system independent code**

  The main part of the *netXTransport Toolkit* consists of operating system independent code. In case of porting the toolkit there is no need to modify this code (see section *Folder Structure* on page 6, '*netXTransport Toolkit*').

- **Operating system dependent code**

  The netXTransport requires a few functions which are operating system dependent (functions named *OS_xxx()*). These functions need to be implemented by the user. All required functions are declared in the header file *OS_Dependent.h* (see section *Folder Structure* on page 6, '*netXTransport Toolkit - Includes*'). For detailed information of the particular functions see section *OS Abstraction* on page 35.

- **User defined code (no restriction how to implement)**

  The netXTransport user defined functions (functions named *USER_xxx()*) need to be implemented by the user. For detailed information of the particular functions see section *USER Implemented Functions* on page 51.

3.1.1 Step-by-Step Guide

- **Copy the *netXTransport Toolkit* Source Folder to your project folder**
  (see section *Folder Structure* on page 6, '*netXTransport Toolkit*')
- **Implement the OS abstraction layer**
  1. Make a copy of the *OS_Template.c* (see section Folder Structure on page 6, '*\Adaptation\Template*') to your project folder and rename the file appropriate (e.g. OS_Linux.c for the Linux operating system).
  2. Implement the empty functions. You may take a look at '*\Adaptation\Win32\OS_Win32.c'* to see how this is done under Windows (or '*\Adaptation\Linux\OS_Linux.c'* for Linux). For detailed information of the particular functions see section *OS Abstraction* on page 35.
- **Implement the USER functions**
  1. Make a copy of the USER_Template.c (see section Folder Structure on page 6, '*\Adaptation\Template*') to your project folder and rename the file appropriate (e.g. USER_Linux.c for the Linux operating system).
  2. Implement the empty functions. You may take a look at '*\Adaptation\USER_Win32.c*' to see how this is done under Windows (or '*\Adaptation\USER_Linux.c'* for Linux). For detailed information of the particular functions see section *USER Implemented Functions* on page 51.
- **Create a project for the build environment**
  Add all C source and header files located in the project folder to your environment.
- **Build the netXTransport project**

Build Options:

Build Options (using C preprocessor definitions)	
Definition	**Description**
NETX_TRANSPORT_BIGENDIAN	Enables support for 'Big Endian'.

Table 5: Toolkit Build Options

3.1.2 Windows Example Folder Structure

Example Folder Structure of the *netXTransport Toolkit* ported to Windows:

Folder Structure	Description		
netXTransport.sln netXTransport_demo.vcproj	Microsoft Visual Studio project and solution files		
netXTransportDemo.c	Function demo source		
Connectors	RS232_Connetor.c	Windows RS232 and USB connector implementation	
	TCP connector	Windows TCP connector	
netXTransportDLL	Sub project to generate a Windows netXTransport DLL from the netX Transport toolkit		
	Toolkit	Operating system independent code (see section *Folder Structure* on page 6, '*netXTransport Toolkit*')	
	OSAbstraction	OS_Linux.c	Operating system dependent code
		OS_Includes.h	Operating system dependent Includes
		stdint.	Standard PC data type definitions
	User	User_win32.c	User function implementation

Table 6: netXTransport Toolkit Port: Example Folder Structure for Linux

Note: The netXTransport DLL expects a "Plugins" sub-directory containing the connector DLLs.
A sample "Plugins" directory with actual connector DLLs can be found in the example directory and must be copied to the directory containing the netXTransport DLL.

3.1.3 Linux Example Folder Structure

Example Folder Structure of the *netXTransport Toolkit* ported to Linux:

Folder Structure	Description		
Make and project files	Linux project files		
netXTransportDemo,c	Function demo source		
Connectors	TCP_Connetor.c	Linux TCP connector implementation	
	tcpconfig0	Configuration information for the connector	
libnetXTransport	Linux project files		
	Toolkit	Operating system independent code (see section *Folder Structure* on page 6, '*netXTransport Toolkit*')	
	OSAbstraction	OS_Linux.c	Operating system dependent code
		OS_Includes.h	Operating system dependent Includes
	User	User_Linux.c	User function implementation

Table 7: netXTransport Toolkit Port: Example Folder Structure for Linux

3.2 Creating a netXTransport Connector

Creating a compatible netXTransport Connector requires an implementation of a fixed interface according to the structure *NETX_CONNECTOR_T Structure* on page 29. The structure defines also the required function interface (see *NETX_CONNECTOR_FUNCTION_TABLE* netXTransport Toolkit, *ConnectorAPI.h*).

The following table shows the *netXTransport Connector* function pointer interface. The functions listed under '*Essential Function Interface*' are at least required. Functions listed under configuration interface are optional, since they are not called from the *netXTransport Toolkit*.

For detailed information how to create a netXTransport connector, refer to the *netX Diagnostic and Remote Access* manual – chapter 3, '*Creating a Custom Connector*' (documentation reference [2]).

Note: For a simple example of a *netXTransport Connector* implementation see the TCP connector for Linux (see section *Folder Structure* on page 6, '*\Connector\Linux*').

netXTransport Connector Function Interface:

Essential Function Interface	
Function Pointer definition	**Description**
PFN_NETXCON_GETIDENTIFIER	Return connector specific identifier (page 20).
PFN_NETXCON_OPEN	Open and initialize Connector (page 21).
PFN_NETXCON_CLOSE	Close and de-initialize Connector (page 21).
PFN_NETXCON_CREATEINTERFACE	Create and initialize interface (page 22).
PFN_NETXCON_INTF_START	Start interface (ready for communication) (page 22).
PFN_NETXCON_INTF_STOP	Stop interface (page 23).
PFN_NETXCON_INTF_SEND	Send data (page 23).
Configuration Interface (optional - access needs to be implemented by the user)	
Function Pointer definition	**Description**
PFN_NETXCON_INTF_GETINFORMATION	Return interface specific information (page 24).
PFN_NETXCON_GETINFORMATION	Return connector specific information (page 25).
PFN_NETXCON_GETCONFIG	Return connector specific configuration (page 25).
PFN_NETXCON_SETCONFIG	Set connector specific configuration (page 26).
PFN_NETXCON_CREATEDIALOG	Create configuration dialog (page 26).
PFN_NETXCON_ENDDIALOG	Close configuration dialog (page 27).

Table 8: netXTransport Connector Function Interface

3.2.1 netXTransport Connector Function Description

The following chapter describes the *netXTransport Connector* functions, notes implementation restrictions and offers useful hints.

Note: For a simple example of a *netXTransport Connector* implementation see the TCP connector for Linux (see section *Folder Structure* on page 6, '*\Connector\Linux*').

3.2.1.1 PFN_NETXCON_GETIDENTIFIER

Function returns a unique identifier.

For an example implementation see section *Folder Structure* on page 6, '*\Connector\Linux*'.

Function Call

```
int32_t pfnConGetIdentifier( char* szIdentifier, NETX_TRANSPORT_UUID_T* pvUUID);
```

Arguments

Argument	Data type	Description
szIdentifier	char*	Name of the interface.
pvUUID	NETX_TRANSPORT_UUID_T*	Returned pointer to UUID information. (For more information see netXTransport Toolkit, *ConnectorAPI.h*.)

Return Values

Return Values	
int32_t	0 on success (for possible errors see netXTransport_Errors.h)

3.2.1.2 PFN_NETXCON_OPEN

Function opens and initializes a connector. In case of a successful initialization it notifies the netXTransport layer of an interface creation (*pfnDevNotifyCallback*).

For an example implementation see section *Folder Structure* on page 6, '*Connector**Linux*'.

Note: The function needs to be thread-safe.

Function Call

```
int32_t pfnConOpen( PFN_NETXCON_INTERFACE_NOTIFY_CALLBACK pfnDevNotifyCallback
                    void*                                 pvUser);
```

Arguments

Argument	Data type	Description
pfnDevNotifyCallback	PFN_NETXCON_INTERFACE_NOTIFY_CALLBACK	netXTransport notify callback. Called when interface creation was successful. (For more information see netXTransport Toolkit, *ConnectorAPI.h*.)
pvUser	void*	User Parameter. Pass through to notify callback.

Return Values

Return Values	
int32_t	0 on success (for possible errors see netXTransport_Errors.h)

3.2.1.3 PFN_NETXCON_CLOSE

Function closes and de-initializes a connector.

For an example implementation see section *Folder Structure* on page 6, '*Connector**Linux*'.

Note: The function needs to be thread-safe.

Function Call

```
int32_t pfnConClose( void);
```

Arguments

Argument	Data type	Description
none		

Return Values

Return Values	
int32_t	0 on success (for possible errors see netXTransport_Errors.h)

3.2.1.4 PFN_NETXCON_CREATEINTERFACE

Function opens and initializes an interface.

For an example implementation see section *Folder Structure* on page 6, '*\Connector\Linux*'.

Function Call

```
void* pfnConCreateInterface( const char* szInterfaceName);
```

Arguments

Argument	Data type	Description
szInterfaceName	const char*	Name of the interface which should be created.

Return Values

Return Values	
void*	Handle to the created interface (used to identify the created interface and its related private resources)

3.2.1.5 PFN_NETXCON_INTF_START

Function starts all resources to be able to send and receive data.

For an example implementation see section *Folder Structure* on page 6, '*\Connector\Linux*'.

Function Call

```
int32_t pfnConIntfStart( void*                                pvInterface
                         PFN_NETXCON_DEVICE_RECEIVE_CALLBACK pfnReceiveData
                         void*                                pvUser);
```

Arguments

Argument	Data type	Description
pvInterface	void*	Handle to the Interface (see return value of PFN_NETXCON_CREATEINTERFACE)
pfnReceiveData	PFN_NETXCON_DEVICE_RECEIVE_CALLBACK	Receive data callback. Needs to be called in case of incoming data. (For more information see netXTransport Toolkit*, ConnectorAPI.h*.)
pvUser	void*	Pass through to PFN_NETXCON_DEVICE_RECEIVE_CALLBACK

Return Values

Return Values	
int32_t	0 on success (for possible errors see netXTransport_Errors.h)

3.2.1.6 PFN_NETXCON_INTF_STOP

Function stops an interface.

For an example implementation see section *Folder Structure* on page 6, '*\Connector\Linux*'.

Note: The function needs to be thread-safe.

Function Call

```
int32_t pfnConIntfStop( void* pvInterface);
```

Arguments

Argument	Data type	Description
pvInterface	void*	Handle to the Interface (see return value of PFN_NETXCON_CREATEINTERFACE)

Return Values

Return Values	
int32_t	0 on success (for possible errors see netXTransport_Errors.h)

3.2.1.7 PFN_NETXCON_INTF_SEND

Function sends the delivered data.

For an example implementation see section Folder Structure on page 6, '*\Connector\Linux*'.

Function Call

```
int32_t pfnConIntfSend( void* pvInterface, unsigned char* pabData, uint32_t ulDataLen);
```

Arguments

Argument	Data type	Description
pvInterface	void*	Handle to the Interface (see return value of PFN_NETXCON_CREATEINTERFACE)
pabData	unsigned char*	Pointer to the data to be sent.
ulDataLen	uint32_t	Length of the data pointed by pabData

Return Values

Return Values	
int32_t	0 on success (for possible errors see netXTransport_Errors.h)

3.2.1.8 PFN_NETXCON_INTF_GETINFORMATION

Function returns interface specific information.

Note: This function is not called from the *netXTransport Toolkit* and can be implemented for custom purposes.

For an example implementation see section *Folder Structure* on page 6, '*\Connector\Linux*'.

Function Call

```
int32_t pfnConIntfGetInformation( void*                 pvInterface,
                                  NETX_INTERFACE_INFO_E eCmd,
                                  uint32_t              ulSize,
                                  void*                 pvInterfaceInfo);
```

Arguments

Argument	Data type	Description
pvInterface	void*	Handle to the Interface (see return value of PFN_NETXCON_CREATEINTERFACE)
eCmd	NETX_INTERFACE_INFO_E	Command to execute.
ulSize	uint32_t	Size of buffer pointed by pvInterfaceInfo.
pvInterfaceInfo	void*	Pointer to receive buffer.

Return Values

Return Values	
int32_t	0 on success (for possible errors see netXTransport_Errors.h)

3.2.1.9 PFN_NETXCON_GETINFORMATION

Function returns connector specific information.

Note: This function is not called from the *netXTransport Toolkit* and can be implemented for custom purposes.

For an example implementation see section *Folder Structure* on page 6, '*\Connector\Linux*'.

Function Call

```
int32_t pfnConGetInformation( NETX_CONNECTOR_INFO_E eCmd,
                              uint32_t              ulSize,
                              void*                 pvConnectorInfo);
```

Arguments

Argument	Data type	Description
eCmd	NETX_CONNECTOR_INFO_E	Command to execute.
ulSize	uint32_t	Size of buffer pointed by pvConnectorInfo.
pvConnectorInfo	void*	Pointer to receive buffer.

Return Values

Return Values	
int32_t	0 on success (for possible errors see netXTransport_Errors.h)

3.2.1.10 PFN_NETXCON_GETCONFIG

Function returns the connector configuration.

Note: This function is not called from the *netXTransport Toolkit* and can be implemented for custom purposes.

For an example implementation see section *Folder Structure* on page 6, '*\Connector\Linux*'.

Function Call

```
int32_t pfnConGetConfig(NETX_CONNECTOR_CONFIG_CMD_E eCmd, void* pvConfig);
```

Arguments

Argument	Data type	Description
eCmd	NETX_CONNECTOR_CONFIG_CMD_E	Command to execute.
pvConfig	void*	Pointer to receive buffer.

Return Values

Return Values	
int32_t	0 on success (for possible errors see netXTransport_Errors.h)

3.2.1.11 PFN_NETXCON_SETCONFIG

Function sets the connector configuration.

Note: This function is not called from the *netXTransport Toolkit* and can be implemented for custom purposes.

For an example implementation see section *Folder Structure* on page 6, '*\Connector\Linux*'.

Function Call

```
int32_t pfnConSetConfig(NETX_CONNECTOR_CONFIG_CMD_E eCmd, const char* szConfig);
```

Arguments

Argument	Data type	Description
eCmd	NETX_CONNECTOR_CONFIG_CMD_E	Command to excecute.
szConfig	const char*	Pointer to configuration string.

Return Values

Return Values	
int32_t	0 on success (for possible errors see netXTransport_Errors.h)

3.2.1.12 PFN_NETXCON_CREATEDIALOG

Function opens a connector configuration dialog.

Note: This function is not called from the *netXTransport Toolkit* and can be implemented for custom purposes.

For an example implementation see section *Folder Structure* on page 6, '*\Connector\Linux*'.

Function Call

```
void* pfnConCreateDialog( void* hParentWnd, const char* szConfig);
```

Arguments

Argument	Data type	Description
hParentWnd	void*	Pointer to the parent window resource (handle to window)
szConfig	const char*	Pointer to configuration string.

Return Values

Return Values	
void*	Pointer to dialog resource

3.2.1.13 PFN_NETXCON_ENDDIALOG

Function closes the connector configuration dialog.

Note: This function is not called from the *netXTransport Toolkit* and can be implemented for custom purposes.

For an example implementation see section *Folder Structure* on page 6, '*\Connector\Linux*'.

Function Call

```
void pfnConEndDialog( void);
```

Arguments

Argument	Data type	Description
none		

Return Values

Return Values	
none	

4 Toolkit Structures

The following structures are used during toolkit initialization and connector registration.

4.1 TL_INIT_T Structure

The TL_INIT_T structure is required for registering a translation layer (see section *netXTransportInit()* on page 30).

Structure name: TL_INIT_T		
Element	**Type**	**Description**
pfnTLInit	PFN_TRANSLATIONLAYER_INIT	Pointer to initialization function of translation layer (see section *pfnTranslationLayerInit* on page 68)
pfnTLDeInit	PFN_TRANSLATIONLAYER_DEINIT	Pointer to de-initialization function of translation layer (see section *pfnTranslationLayerDeInit* on page 69)
pvData	void*	User data passed to *pfnTLInit/pfnTLDeInit*

Table 9: Toolkit Structures: TL_INIT_T

4.2 NETX_TL_INTERFACE_T Structure

The NETX_TL_INTERFACE_T structure defines a custom defined translation layer (see section *netXTransportAddTranslationLayer()* on page 33).

Structure name: NETX_TL_INTERFACE_T, PNETX_TL_INTERFACE_T		
Element	**Type**	**Description**
tList	- / -	Internal
usDataType	uin16_t	Data type of the Translation Layer definition
pfnProbe	PFN_TL_PROBE	see *PFN_TL_PROBE*
pfnRelease	PFN_TL_RELEASE	see *PFN_TL_RELEASE*
pfnScan	PFN_TL_SCAN	see *PFN_TL_SCAN*
pfnRestoreCon	PFN_TL_RESTORE_CON	see *PFN_TL_RESTORE_CON*
pfnReceive	PFN_TL_RECEIVE	see *PFN_TL_RECEIVE*
pfnMapFunctionTable	PFN_TL_MAPPFNTABLE	see *PFN_TL_MAPPFNTABLE*
hNetXTransportDevGroup	NXTHANDLE	Internal
hNetXTransport	NXTHANDLE	Internal

Table 10: Toolkit Structures: NETX_TL_INTERFACE_T

4.3 NETX_CONNECTOR_T Structure

The NETX_CONNECTOR_T structure is required for registering a *netXTransport Connector* (see *netXTransportAddConnector()* on page 32).

Structure name: NETX_CONNECTOR_T, PNETX_CONNECTOR_T		
Element	**Type**	**Description**
tFunctions	NETX_CONNECTOR_FUNCTION_TABLE	Connector function table see section *netXTransport Connector Function Description* on page 20
szConIdentifier	char*	Name of the connector
tConnectorUUID	NETX_TRANSPORT_UUID_T	UUID of the connector
ulConnectorInfoSize	uint32_t	Size of data pointed by *pvConnectorInfo*
pvConnectorInfo	void*	Private connector information

Table 11: Toolkit Structures: NETX_CONNECTOR_T

5 Toolkit Functions

The following chapter describes the *netXTransport Toolkit* functions, its usage and offers background information of the internal processes.

5.1 Toolkit API Functions

The following sections give an introduction to the netXTransport API functions.

Function	Description
netXTransportInit()	netXTransport Toolkit initialization (on page 30).
netXTransportDeinit()	netXTransport Toolkit de-initialization (on page 31).
netXTransportStart()	Starts netXTransport Toolkit (on page 31).
netXTransportStop()	Stops netXTransport Toolkit (on page 32).
netXTransportAddConnector()	Adds netXTransport Connector (on page 32).
netXTransportRemoveConnector()	Removes netXTransport Connector (on page 33).
netXTransportAddTranslationLayer()	Adds netXTransport Translation Layer (on page 33).
netXTransportRemoveTranslationLayer()	Removes netXTransport Translation Layer (on page 34).
netXTransportCyclicFunction()	Handles netXTransport cyclic jobs (on page 34).

Table 12: netXTransport API Interface

For detailed information of the internal processing of the different functions see section *netXTransport Internals* on page 53.

5.1.1 netXTransportInit()

Function initializes the *netXTransport Toolkit* and registers the required translation layers. For detailed information how to initialize *atTLInitTable* see section *netXTransport Translation Layer Initialization* on page 68.

Function Call

```
int32_t netXTransportInit( TL_INIT_T atTLInitTable[], uint32_t ulTableSize)
```

Arguments

Argument	Data type	Description
atTLInitTable	TL_INIT_T	Array of translation layer initialization structures (see section *TL_INIT_T Structure* on page 28).
ulTableSize	uint32_t	Size of table pointed by *atTLInitTable*.

Return Values

Return Values	
NXT_NO_ERROR on success	For possible errors see netXTransport_Errors.h.

5.1.2 netXTransportDeinit()

The function de-initializes the *netXTransport Toolkit* and frees all registered translation layers.

Function Call

```
void netXTransportDeinit( void)
```

Arguments

Argument	Data type	Description
none		

Return Values

None

5.1.3 netXTransportStart()

The function starts the *netXTransport Toolkit*. This process starts all registered connectors and schedules the device discovering process. Before Starting the toolkit make sure to have at least one registered connector (see *netXTransportAddConnector()*). For detailed information of the internals starting process refer to section *Starting the netXTransport Toolkit* on page 66

Function Call

```
int32_t netXTransportStart( PFN_DEVICEBROWSE_CALLBACK pfnBrwsCB, void* pvUser)
```

Arguments

Argument	Data type	Description
pfnBrwsCB	PFN_DEVICEBROWSE_CALLBACK	Optional: Callback called once if a device is discovered the first time. (For detailed information see *netXTransport.h* in *netXTransport Toolkit*)
pvUser	void*	User parameter passed through to discover callback.

Return Values

Return Values	
NXT_NO_ERROR on success	For possible errors see netXTransport_Errors.h.

5.1.4 netXTransportStop()

The function stops the *netXTransport Toolkit* (stops all registered connectors).

Function Call

```
void netXTransportStop( void)
```

Arguments

Argument	Data type	Description
none		

Return Values

None

5.1.5 netXTransportAddConnector()

The function registers a netXTransport connector. For an example usage see section *netXTransport Connector Registration* on page 79.

Function Call

```
int32_t netXTransportAddConnector( PNETX_CONNECTOR_T ptConnector)
```

Arguments

Argument	Data type	Description
ptConnector	PNETX_CONNECTOR_T	Pointer to netXTransport connector structure (see section *NETX_CONNECTOR_T Structure* on page 29)

Return Values

Return Values	
NXT_NO_ERROR on success	For possible errors see netXTransport_Errors.h.

5.1.6 netXTransportRemoveConnector()

The function removes a previously registered netXTransport connector by a call to *netXTransportAddConnector()*. For an example usage see section *netXTransport Connector Registration* on page 79.

Function Call

```
void netXTransportRemoveConnector( PNETX_CONNECTOR_T ptConnector)
```

Arguments

Argument	Data type	Description
ptConnector	PNETX_CONNECTOR_T	Pointer to netXTransport connector structure (see section *NETX_CONNECTOR_T Structure* on page 29).

Return Values

None

5.1.7 netXTransportAddTranslationLayer()

The function registers a netXTransport translation layer. For an example usage see section *Toolkit Initialization* on page 78.

Note: The function is only required when registering custom defined translation layers. The *netXTransport Toolkit* provides already implemented translation layers (e.g. cifX API Marshaller/rcX-Packet). For registering one of the already implemented types see section *netXTransport cifX API Translation Layer Initialization* on page 70.

Function Call

```
int32_t netXTransportAddTranslationLayer( PNETX_TL_INTERFACE_T ptTLInterface,
                                          NXTHANDLE*           phnetXTransportDevGroup);
```

Arguments

Argument	Data type	Description
ptTLInterface	PNETX_TL_INTERFACE_T	Pointer to netXTransport Translation Layer structure (see section Toolkit Structures: NETX_TL_INTERFACE_T on page 28).
phnetXTransportDevGroup	NXTHANDLE*	Returned handle to netXTransport device group.

Return Values

Return Values	
NXT_NO_ERROR on success	For possible errors see netXTransport_Errors.h.

5.1.8 netXTransportRemoveTranslationLayer()

The function removes a previously registered translation layer by a call to *netXTransportAddTranslationLayer()*.

Function Call

```
void netXTransportRemoveTranslationLayer( NXTHANDLE hnetXTransportDevGroup);
```

Arguments

Argument	Data type	Description
hnetXTransportDevGroup	NXTHANDLE	Handle to netXTransport device group (see section *netXTransportAddTranslationLayer()* on page 33)

Return Values

None

5.1.9 netXTransportCyclicFunction()

Function handles cyclic jobs of the *netXTransport Toolkit.* The function needs to be called cyclically by the user and therefore needs to be processed in a separate thread. The function must be called cyclically after starting the *netXTransport Toolkit* (*netXTransportStart()*) and must not be called after stopping the toolkit (*netXTransportStop()*).

Note: Function needs to be called in a separate thread. The recommended cycle is about 100ms.

Function Call

```
void netXTransportCyclicFunction (void);
```

Arguments

None

Return Values

None

5.2 OS Abstraction

The OS abstraction layer is introduced to allow the toolkit to run under several operating systems, without needing to change the toolkit components. The OS Abstraction needs to be implemented by the user.

OS Abstraction	
Memory Functions	
OS_Malloc	Allocate memory
OS_Memfree	Free allocated memory
OS_Memrealloc	Change size of an allocated memory block
OS_Memset	Set a memory area
OS_Memcpy	Copy a memory area
OS_Memcmp	Compare a memory area
Synchronisation Functions (Critical Section)	
OS_CreateLock	Create a lock object
OS_EnterLock	Enter a locked program region
OS_LeaveLock	Leave a locked program region
OS_DeleteLock	Delete a lock object
Synchronisation Functions (Semaphore)	
OS_CreateSemaphore	Create a Mutex (Mutual Exclusion) object
OS_PutSemaphore	Wait for a Mutex
OS_DeleteSemaphore	Release a Mutex
OS_WaitSemaphore	Delete a Mutex object
Synchronisation Functions (Event)	
OS_CreateEvent	Create an event object
OS_SetEvent	Set an event object into a signaled state
OS_ResetEvent	Reset an event object to a none signaled state
OS_DeleteEvent	Delete an event object
OS_WaitEvent	Wait for an event to be signaled
Timing Functions	
OS_GetMilliSecCounter	Get a millisecond counter value
OS_Sleep	Suspend a process for a given time
String Functions(Mutal Exclusion)	
OS_Strcmp	Copy a string
OS_Strlen	Get the length of a string
OS_Strncpy	Compare two strings
OS_Strnicmp	Compare two strings (case-insensitive)
OS_Strvsprintf	Compare two strings (case-insensitive)
OS_Strcspn	Compare two strings (case-insensitive)
OS_Strcat	Compare two strings (case-insensitive)

Table 13: OS Abstraction Functions

5.2.1 Memory Functions

Memory allocation and operation could differ between operating systems and even inside an operating system, depending on the mode the application/driver is running. The memory routines are included in the OS Abstraction to allow easy adaptation and modification.

5.2.1.1 OS_Memalloc

Memory allocation function.

Function Call

```
void* OS_Memalloc(uint32_t ulSize)
```

Arguments

Argument	Data type	Description
ulSize	uint32_t	Size in bytes to allocate

Return Values

A pointer to the allocated memory is returned. NULL indicates memory allocation failure.

5.2.1.2 OS_Memfree

Memory freeing function.

Function Call

```
void OS_Memfree(void* pvMem)
```

Arguments

Argument	Data type	Description
pvMem	void*	Memory block to free

5.2.1.3 OS_Memrealloc

Memory resize / reallocation Function.

Function Call

```
void* OS_Memrealloc(void* pvMem, uint32_t ulNewSize)
```

Arguments

Argument	Data type	Description
pvMem	void*	Memory block to resize
ulNewSize	uint32_t	New size of block in bytes

Return Values

A pointer to the reallocated memory is returned. NULL indicates memory reallocation failure.

5.2.1.4 OS_Memcpy

Copy function for non-overlapping memory block which copies one memory block to another memory block.

Function Call

```
void OS_Memcpy( void*    pvDest,
                void*    pvSrc,
                uint32_t ulSize)
```

Arguments

Argument	Data type	Description
pvDest	void*	Pointer to the destination memory block
pvSrc	void*	Pointer to the source memory block
ulSize	uint32_t	Size in bytes to copy

5.2.1.5 OS_Memmove

Move overlapping memory block from one block to another block.

Function Call

```
void OS_Memmove( void*    pvDest,
                 void*    pvSrc,
                 uint32_t ulSize)
```

Arguments

Argument	Data type	Description
pvDest	void*	Pointer to the destination memory block
pvSrc	void*	Pointer to the source memory block
ulSize	uint32_t	Size in bytes to move

5.2.1.6 OS_Memset

Initialize a memory block to a predefined value.

Function Call

```
void OS_Memset( void*    pvMem,
                uint8_t  bFill,
                uint32_t ulSize)
```

Arguments

Argument	Data type	Description
pvMem	void*	Pointer to the memory block to initialize
bFill	uint8_t	Filling byte
ulSize	uint32_t	Size in bytes to initialize

5.2.1.7 OS_Memcmp

Compare the content of two memory blocks.

Function Call

```
int OS_Memcmp(  void*           pvBuf1,
                void*           pvBuf2,
                uint32_t        ulSize)
```

Arguments

Argument	Data type	Description
pvBuf1	void*	First compare buffer
pvBuf2	void*	Second compare buffer
ulSize	uint32_t	Number of bytes to compare

Return Values

Return Values	
0	Memory contents equal
<0	pvBuf1 < pvBuf2
>0	pvBuf1 > pvBuf2

5.2.2 Synchronization / Locking / Timing

5.2.2.1 OS_CreateLock

Creates a new synchronization object (e.g. Critical Section).

Function Call

```
void* OS_CreateLock(void)
```

Arguments

Argument	Data type	Description
none		

Return Values

Return Values	
NULL	Object creation error
otherwise	Handle to a synchronization object

5.2.2.2 OS_DeleteLock

Delete a previously created synchronization object (e.g. Critical Section).

Function Call

```
void OS_DeleteLock(void* pvLock)
```

Arguments

Argument	Data type	Description
pvLock	void*	Synchronization object to delete

Return Values

None

5.2.2.3 OS_EnterLock

Lock the synchronization object for the current context. This call blocks until the lock has been acquired.

Function Call

```
void OS_EnterLock(void* pvLock)
```

Arguments

Argument	Data type	Description
pvLock	void*	Synchronization object to enter

Return Values

none

5.2.2.4 OS_LeaveLock

Unlock the synchronization object for the current context.

Function Call

```
void OS_LeaveLock(void* pvLock)
```

Arguments

Argument	Data type	Description
pvLock	void*	Synchronization object to leave

Return Values

None

5.2.2.5 OS_CreateSemaphore

Creates a new synchronization object.

Function Call

```
void* OS_CreateSemaphore(uint32_t ulInitialCount)
```

Arguments

Argument	Data type	Description
ulInitialCount	uint32_t	Maximum number of parallel access

Return Values

Return Values	
NULL	Object creation error
otherwise	Handle to a synchronization object

5.2.2.6 OS_DeleteSemaphore

Delete a previously created synchronization object.

Function Call

```
void OS_DeleteSemaphore(void* pvSem)
```

Arguments

Argument	Data type	Description
pvSem	void*	Synchronization object to delete

Return Values

None

5.2.2.7 OS_WaitSemaphore

Lock the synchronization object for the current context (decrements the semaphores reference counter).

Function Call

```
void OS_WaitSemaphore( void* pvSem, uint32_t ulTimeout)
```

Arguments

Argument	Data type	Description
pvSem	void*	Synchronization object to enter
ulTimeout	uint32_t	Time in ms to wait

Return Values

None

5.2.2.8 OS_PutSemaphore

Increments the semaphores reference counter.

Function Call

```
void OS_PutSemaphore( void* pvSem, uint32_t ulCount)
```

Arguments

Argument	Data type	Description
pvSem	void*	Synchronization object to leave
ulCount	uint32_t	Synchronization object to leave

Return Values

None

5.2.2.9 OS_Sleep

Delay execution of a program by the given time in milliseconds. This call is allowed to do a task switch, but can also be implemented as stall execution.

Function Call

```
void OS_Sleep(uint32_t ulSleepTimeMs)
```

Arguments

Argument	Data type	Description
ulSleepTimeMs	uint32_t	Time in ms to sleep

Return Values

None

5.2.2.10 OS_GetMilliSecCounter

Retrieve the free running millisecond counter of the operating system. The resolution influences the timeout monitoring accuracy.

Function Call

```
uint32_t OS_GetMilliSecCounter(void)
```

Arguments

Argument	Data type	Description
none		

Return Values

Actual value of the systems millisecond counter

5.2.3 Synchronization Functions (Event Handling)

Events are used to indicate changes in interrupt mode from the interrupt routine to the user functions.

5.2.3.1 OS_CreateEvent

Create a new, unnamed, automatic reset event.

Function Call

```
void* OS_CreateEvent(void)
```

Arguments

Argument	Data type	Description
none		

Return Values

Return Values	
NULL	Event creation error
otherwise	Handle to an event object

5.2.3.2 OS_DeleteEvent

Delete a previously created event.

Function Call

```
void OS_DeleteEvent(void* pvEvent)
```

Arguments

Argument	Data type	Description
pvEvent	void*	Event handle to delete

Return Values

Return Values	
none	

5.2.3.3 OS_SetEvent

Signal an event.

Function Call

```
void OS_SetEvent(void* pvEvent)
```

Arguments

Argument	Data type	Description
pvEvent	void*	Event handle to signal

Return Values

Return Values	
none	

5.2.3.4 OS_ClearEvent

Reset a signaled event.

Function Call

```
void OS_ResetEvent(void* pvEvent)
```

Arguments

Argument	Data type	Description
pvEvent	void*	Event handle to reset

Return Values

Return Values	
none	

5.2.3.5 OS_WaitEvent

Wait for the occurrence of a given event

Function Call

```
uint32_t OS_WaitEvent(    void*     pvEvent,
                          uint32_t  ulTimeout)
```

Arguments

Argument	Data type	Description
pvEvent	void*	Event handle to wait for being signaled
ulTimeout	uint32_t	Time in ms to wait for event

Return Values

Return Values	
CIFX_EVENT_SIGNALLED (0)	Event was signaled during wait
CIFX_EVENT_TIMEOUT (1)	Timeout waiting for event

5.2.4 String Functions

String operations are used inside the toolkit for the board/alias name handling and also for accessing ASCII strings inside the firmware information. The implementation should rely on ASCII / MBCS characters.

5.2.4.1 OS_Strncpy

Copy one string into another, considering the length of the destination buffer.

Function Call

```
char* OS_Strncpy(    char*          szDest,
                     const char*    szSource,
                     uint32_t       ulLen)
```

Arguments

Argument	Data type	Description
szDest	char*	Destination string buffer
szSource	const char*	Source string buffer
ulLen	uint32_t	Maximum length to copy

Return Values

Pointer to *szDest.*

5.2.4.2 OS_Strlen

Count the number of characters inside a string.

Function Call

```
int OS_Strlen(  const char* szText)
```

Arguments

Argument	Data type	Description
szText	const char*	String to determine length from

Return Values

Length of string in characters.

5.2.4.3 OS_Strcmp

Compare the content of two strings.

Function Call

```
int OS_Strcmp(  const char*     pszBuf1,
                const char*     pszBuf2)
```

Arguments

Argument	Data type	Description
pszBuf1	const char*	First compare string
pszBuf2	const char*	Second compare string

Return Values

Return Values	
0	String are equal
<0	pszBuf1 less than pszBuf2
>0	pszBuf1 greater than pszBuf2

5.2.4.4 OS_Strvsprintf

Write formatted string to a destination string buffer.

Function Call

```
int OS_Strvsprintf( char *szDest, uint32_t ulSize, const char *format, ...)
```

Arguments

Argument	Data type	Description
szDest	char*	Destination string buffer
ulSize	uint32_t	Size of destination string
format	const char*	Format string
...		Variable list of arguments

Return Values

Return Values	
szDest	Pointer to destination string

5.2.4.5 OS_Strcat

Concatenate two strings.

Function Call

```
int OS_Strcat(char* szDest, uint32_t ulDstSize, char* szSrc)
```

Arguments

Argument	Data type	Description
szDest	char*	Destination string buffer
ulDstSize	uint32_t	Size of destination string
szSrc	char*	Pointer to source string

Return Values

Return Values	
szDest	Pointer to destination string

5.3 USER Implemented Functions

USER functions are functions with target system dependencies like file functions, trace outputs or toolkit specific configurations. Because of the system independency of the toolkit only the user knows the target system and therefore he needs to implement target dependent part of these functions. The toolkit offers a user module containing the function bodies of the functions and an implementation example for some target systems are available.

USER Functions	
USER_TraceInitialize	Initialization for debug and error messages
USER_TraceDeInitialize	De-initialization of debug logging
USER_Trace	Generic function allows logging of debug message
USER_GetConnectorTimeout	Retrieves connector specific timeouts

Table 14: User Implementation Functions

5.3.1 USER_TraceInitialize

This function is called from the toolkit to prepare resources required for "Debug" and "Error" trace messages. Depending on the implementation these trace messages can be written to a file, to a hardware interface, to a screen or somewhere else. It is up to the implementation where the trace can be seen.

Function Call

```
void USER_TraceInitialize(NETX_TRANSPORT_DATA_T* ptnetXTransportInst)
```

Arguments

Argument	Data type	Description
ptnetXTransportInst	NETX_TRANSPORT_DATA_T*	Pointer to netXTransport instance

5.3.2 USER_TraceDeInitialize

This function is called from the toolkit to release previous acquired resources by calling *USER_TraceInitialize().*

Function Call

```
void USER_Trace(NETX_TRANSPORT_DATA_T* ptnetXTransportInst)
```

Arguments

Argument	Data type	Description
ptnetXTransportInst	NETX_TRANSPORT_DATA_T*	Pointer to netXTransport instance

5.3.3 USER_Trace

This function is called from the toolkit to output "Debug" and "Error" messages. The amount of outputs can be controller through a global variable "*g_ulTraceLevel*".

Available Trace Settings are:

```
/*****************************************************************************/
/*! Trace level definitions                                                  */
/*****************************************************************************/
#define TRACE_LEVEL_DEBUG   0x00000001    /*!< enables debug trace   */
#define TRACE_LEVEL_INFO    0x00000002    /*!< enables info trace    */
#define TRACE_LEVEL_WARNING 0x00000004    /*!< enables warning trace */
#define TRACE_LEVEL_ERROR   0x00000008    /*!< enables error trace   */
```

Function Call

```
void USER_Trace(NETX_TRANSPORT_DATA_T     ptnetXTransportInst,
                uint32_t                  ulTraceLevel,
                const char*               szFormat,
                ...)
```

Arguments

Argument	Data type	Description
ptnetXTransportInst	PDEVICEINSTANCE	Device instance the trace is made for
ulTraceLevel	uint32_t	Trace level the message is output for
szFormat	string	Printf style format string
…		Variable argument list for printf

5.3.4 USER_GetConnectorTimeout

This function is called from the toolkit to retrieve the connector specific timeouts.

Function Call

```
void USER_GetConnectorTimeout( PNETX_CONNECTOR_T ptConnector, uint32_t* pulTimeout)
```

Arguments

Argument	Data type	Description
ptConnector	PNETX_CONNECTOR_T	Pointer to netXTransport connector
pulTimeout	uint32_t	Pointer to buffer for returned timeout

6 netXTransport Internals

6.1 Internal Structures and Functions

The following structures and functions are for internal usage and for the translation layer implementation (see section *netXTransport Translation Layer Initialization* on page 68). The offered function interface provides access to the *netXTransport Generic Devices* (see section *netXTransport Generic Device* on page 13) and its related resources like *Translation Layer Information* (see section *Translation Layer Information* on page 14) and *Transport Layer Information* (see section *Transport Layer Information* on page 14).

6.1.1 NETX_TRANSPORT_DATA_T Structure

The structure holds the global information of all underlying layers (see section *netXTransport Device Grouping* page 13).

Structure name: NETX_TRANSPORT_DATA_T, PNETX_TRANSPORT_DATA_T		
Element	**Type**	**Description**
tConnectors	struct CONNECTOR_LIST	List of registered connectors
tNetXTransportInterface	struct NETXTRANSPORT_IF_LIST	List of created interfaces
tNetXTransportDevGroupList	struct DEV_GROUP_LIST	List of registered Translation Layers
pvLogFile	void*	Pointer to 'USER_Trace' resources
ptTLInitTable	PTL_INIT_T	Translation Layer (de-)initialization table
ulInitTableSize	uint32_t	Size of table pointed by PTL_INIT_T
eState	NXT_NETXTRANSPORT_STATE_E	Current toolkit state
pvnetXTransportLock	void*	Global synchronization lock
pfnDeviceDiscoverCB	PFN_DEVICEBROWSE_CALLBACK	Callback, called during device discovering process
pvDeviceDiscoverUserParam	void*	User parameter PFN_DEVICEBROWSE_CALLBACK (passed through to pfnDeviceDiscoverCB)

Table 15: Toolkit Structures: NETX_TRANSPORT_DATA_T

6.1.2 NETX_TRANSPORT_DEV_GROUP_T Structure

The structure holds all information required to define the *netXTransport Device Group* (see section *netXTransport Device Grouping* page 13).

Structure name: NETX_TRANSPORT_DEV_GROUP_T, PNETX_TRANSPORT_DEV_GROUP_T		
Element	**Type**	**Description**
ptTLInterface	PNETX_TL_INTERFACE_T	Pointer to the corresponding Translation Layer (see section *NETX_TL_INTERFACE_T Structure* on page 28)
tGenericDeviceList	GEN_DEVICE_LIST	List of all devices matching the device type of ptTLInterface

Table 16: Toolkit Structures: NETX_TRANSPORT_DEV_GROUP_T

6.1.3 NETX_TL_INTERFACE_T Structure

The structure defines the generic Translation Layer interface and its resources.

Structure name: NETX_TL_INTERFACE_T, PNETX_TRANSPORT_DEVICE_T		
Element	**Type**	**Description**
tList	- / -	Internal
usDataType	uint16_t	Data Type
pfnProbe	PFN_TL_PROBE	Called on connection establishment
pfnRelease	PFN_TL_RELEASE	Called when releasing Translation Layer
pfnScan	PFN_TL_SCAN	Called during endpoint scan
pfnReceive	PFN_TL_RECEIVE	Called in case of receive data
pfnMapFunctionTable	PFN_TL_MAPPFNTABLE	Translation Layer specific function pointer table
hNetXTransport	NXTHANDLE	Internal

Table 17: Toolkit Structures: NETX_TL_INTERFACE_T

6.1.4 NETX_TRANSPORT_GENERIC_DEVICE_T Structure

The structure defines the "*Generic netXTransport Device"* and its resources (see section *netXTransport Additional Information* on page 13).

Structure name: NETX_TRANSPORT_GENERIC_DEVICE_T, PNETX_TRANSPORT_GENERIC_DEVICE_T		
Element	**Type**	**Description**
tList	- / -	Internal
szDeviceName	char	Device Name (see device naming, *netXTransport Additional Information* on page 13)
ulDeviceIdentifier	uint32_t	netXTransport unique device identifier
hTransport	void*	Handle to its Transport Layer Instance
eState	NXT_DEVICE_STATE_E	Device current state
ulRefCounter	uint32_t	Device reference counter
pvTLDeviceData	void*	Pointer to device specific information structure (information generated by the Translation Layer, see section *Translation Layer Information* on page 14)
hNetXTransportDeviceHandle	NXTHANDLE	Handle to itself
pvDeviceLock	void*	Device synchronization lock
pvEvent	void*	Device specific events (state change)
pvInterface	void*	Pointer to its related interface

Table 18: Toolkit Structures: NETX_TRANSPORT_GENERIC_DEVICE_T

6.1.5 NETX_TRANSPORT_INTERFACE_T Structure

The structure defines the netXTransport interface related resources (see *Device Grouping*, netXTransport Additional Information on page 13).

Structure name: NETX_TRANSPORT_INTERFACE_T, PNETX_TRANSPORT_INTERFACE_T		
Element	**Type**	**Description**
tList	- / -	Internal
szDeviceName	char	Interface name
hTransport	NXTHANDLE	Handle to its Transport Layer Instance
pvUser	void*	Internal
pvConnector	void*	Pointer to private connector data
eInterfaceState	NXT_INTERFACE_STATE_E	Current interface state
tDeviceListRef	struct PDEVICE_LIST	List of all device related to this interface
ulDeviceCounter	uint32_t	Number of devices listed in tDeviceListRef
ulActiveTransfers	uint32_t	Number of active transfers
ulActiveConnections	uint32_t	Number of active connections
pvInterfaceLock	void*	Interface lock (for synchronization purposes)

Table 19: Toolkit Structures: NETX_TRANSPORT_INTERFACE_T

6.1.6 netXTransportGetHandleToDeviceGroup

The function returns a netXTransport handle to the requested translation layer, specified by the data type *usDataType*.

Function Call

```
NXTHANDLE netXTransportGetHandleToDeviceGroup( uint16_t usDataType)
```

Arguments

Argument	Data type	Description
usDataType	uint16_t	Type of the requested translation layer (e.g. cifX API Marshaller = 0x200, rcX-Packet = 0x100)

Return Values

Return Values	
int32_t	0 on success (for possible errors see netXTransport_Errors.h)

6.1.7 netXTransportGetDeviceCount

The function returns the number of registered devices of the *netXTransport Device Group* (see section netXTransport Device *Grouping* on page 13).

Function Call

```
int32_t netXTransportGetDeviceCount( NXTHANDLE hNetXTransportDevGroup)
```

Arguments

Argument	Data type	Description
hNetXTransportDevGroup	NXTHANDLE	Handle to the netXTransport device group (see section *netXTransportGetHandleTo* on page 55)

Return Values

Return Values	
int32_t	Number of detected devices

6.1.8 netXTransportGetTLFctTable

The function returns a pointer to the netXTransport translation layer function pointer table information structure *pfnDriverFunc*.

Function Call

```
int32_t netXTransportGetTLFctTable( NXTHANDLE                hNetXTransportDevGroup,
                                    FUNCTION_POINTER_TABLE_T* pfnDriverFunc);
```

Arguments

Argument	Data type	Description
hNetXTransportDevGroup	NXTHANDLE	Handle to the netXTransport device group (see section *netXTransportGetHandleTo* on page 55)
pfnDriverFunc	FUNCTION_POINTER_TABLE_T*	Translation layer dependent (see *TranslationLayer.h*, netXTransport Toolkit)

Return Values

Return Values	
int32_t	0 on success (for possible errors see *netXTransport_Errors.h*)

6.1.9 netXTransportGetTLFctTableByDevice

The function returns a pointer to the netXTransport translation layer function pointer table information structure *pfnDriverFunc*.

Function Call

```
int32_t netXTransportGetTLFctTableByDevice( NXTHANDLE hnetXTransportDeviceHandle,
                                            FUNCTION_POINTER_TABLE_T* pfnDriverFunc)
```

Arguments

Argument	Data type	Description
hnetXTransportDeviceHandle	NXTHANDLE	Handle to the netXTransport device (see section *netXTransportGetHandleTo* on page 55)
pfnDriverFunc	FUNCTION_POINTER_TABLE_T*	Translation layer dependent (see *TranslationLayer.h*, netXTransport Toolkit)

Return Values

Return Values	
int32_t	0 on success (for possible errors *see netXTransport_Errors.h*)

6.1.10 netXTransportGetTLInfo

The function returns a pointer to the translation layer private interface resources. For more information see section *Translation Layer Information* on page 14.

Function Call

```
NXTHANDLE netXTransportGetTLInfo( NXTHANDLE hnetXTransportDeviceHandle)
```

Arguments

Argument	Data type	Description
hnetXTransportDeviceHandle	NXTHANDLE	Handle to the *netXTransport Generic Device* (see section *netXTransportGetDeviceHandle* on page 58 or *netXTransportGetDeviceHandleByNumber* on page 59)

Return Values

Return Values	
void*	Pointer to the translation layer specific private endpoint information

6.1.11 netXTransportGetTLDeviceData

The function returns a pointer to the translation layer private device information. For more information see *Translation Layer Information* on page 14.

Function Call

```
void* netXTransportGetTLDeviceData( NXTHANDLE hnetXTransportDeviceHandle)
```

Arguments

Argument	Data type	Description
hnetXTransportDeviceHandle	NXTHANDLE	Handle to the *netXTransport Generic Device* (see section *netXTransportGetDeviceHandle* on page 58 or *netXTransportGetDeviceHandleByNumber* on page 59)

Return Values

Return Values	
void*	Pointer to the translation layer specific private device information

6.1.12 netXTransportGetDeviceHandle

The function returns a handle to the *netXTransport Generic Device* (see section *netXTransport Generic Device* on page 13) specified by name.

Function Call

```
int32_t netXTransportGetDeviceHandle( char*      szDeviceName,
                                      NXTHANDLE* hnetXTransportDeviceHandle)
```

Arguments

Argument	Data type	Description
szDeviceName	char*	Name of the requested device
hnetXTransportDeviceHandle	NXTHANDLE*	Pointer to the handle

Return Values

Return Values	
int32_t	0 on success (for possible errors see *netXTransport_Errors.h*)

6.1.13 netXTransportGetDeviceHandleByNumber

The function returns a handle to the *netXTransport Generic Device* (see section *netXTransport Generic Device* on page 13) specified by number. The number is absolute in relation to the number of devices registered at the corresponding *netXTransport Device Group* specified by *hnetXTransportDevGroup* handle.

Function Call

```
NXTHANDLE netXTransportGetDeviceHandleByNumber( NXTHANDLE hnetXTransportDevGroup,
                                                uint32_t  ulDeviceNo)
```

Arguments

Argument	Data type	Description
hnetXTransportDevGroup	NXTHANDLE	Handle to the netXTransport device group (see section *netXTransportGetHandleTo* on page 55)
ulDeviceNo	uint32_t	Number of the device to be retrieved

Return Values

Return Values	
NXTHANDLE	Handle to the requested netXTransport device handle

6.1.14 netXTransportGetDeviceName

The function returns the name of the netXTransport device specified by the netXTransport device handle (see section *Device Naming* Convention on page 15).

Function Call

```
void netXTransportGetDeviceName( NXTHANDLE hNetxTransportDeviceHandle,
                                 char*     szDeviceName,
                                 uint32_t  ulStrlen)
```

Arguments

Argument	Data type	Description
hNetXTransportDeviceHandle	NXTHANDLE	Handle to the *netXTransport Generic Device* (see section *netXTransportGetDeviceHandle* on page 58 or *netXTransportGetDeviceHandleByNumber* on page 59)
szDeviceName	char*	Pointer to the buffer for device name
ulStrlen	uint32_t	Size of the buffer pointed by szDeviceName

Return Values

None

6.1.15 netXTransportAttachDevice

The function needs to be called before accessing the device resources maintained by the netXTransport toolkit (e.g. Translation Layer information via *netXTransportGetTLDeviceData* on page 58) and increments the devices reference counter.

If the functions returns 0 the device is not available (see section *netXTransport Device State Change* on page 75).

To leave the resources call *netXTransportDetachDevice()*.

Function Call

```
int netXTransportAttachDevice( NXTHANDLE hnetXTransportDeviceHandle)
```

Arguments

Argument	Data type	Description
hNetXTransportDeviceHandle	NXTHANDLE	Handle to the *netXTransport Generic Device* (see section *netXTransportGetDeviceHandle* on page 58 or *netXTransportGetDeviceHandleByNumber* on page 59)

Return Values

Return Values	
int	!= 0 if the function succeeds (the device is accessible)

6.1.16 netXTransportDetachDevice

This function decrements the devices reference counter and needs to be called to detach from the device previous attached by call to *netXTransportAttachDevice().*

Function Call

```
void netXTransportDetachDevice( NXTHANDLE hNetXTransportDeviceHandle)
```

Arguments

Argument	Data type	Description
hNetXTransportDeviceHandle	NXTHANDLE	Handle to the *netXTransport Generic Device* (see section *netXTransportGetDeviceHandle* on page 58 or *netXTransportGetDeviceHandleByNumber* on page 59)

Return Values

None

6.1.17 netXTransportCheckInterfaceState

The function checks the interfaces state.

- state = 1 Interface active and accessible interface the function returns 1
- state = 0 Interface in-active and not accessible

Function Call

```
int netXTransportCheckInterfaceState(NXTHANDLE hnetXTransportDevGroup)
```

Arguments

Argument	Data type	Description
hnetXTransportDevGroup	NXTHANDLE	Handle to the netXTransport device group (see section *netXTransportGetHandleTo* on page 55)

Return Values

Return Values	
int	!= 0 if the function succeeds and the device is accessible

6.1.18 netXTransportActivateConnection

The function activates a connection and starts "Keep Alive" background processing on the related interface. It needs to be called if "Keep Alive" should be activated.

The background services can be stopped by calling *netXTransportDeActivateConnection().*

Function Call

```
void netXTransportActivateConnection( NXTHANDLE hNetXTransportDeviceHandle)
```

Arguments

Argument	Data type	Description
hNetXTransportDeviceHandle	NXTHANDLE	Handle to the *netXTransport Generic Device* (see section *netXTransportGetDeviceHandle* on page 58 or *netXTransportGetDeviceHandleByNumber* on page 59)

Return Values

None

6.1.19 netXTransportDeActivateConnection

The function stops the background processing previously started with *netXTransportActivateConnection().*

Function Call

```
void netXTransportDeActivateConnection( NXTHANDLE hNetXTransportDeviceHandle)
```

Arguments

Argument	Data type	Description
hNetXTransportDeviceHandle	NXTHANDLE	Handle to the *netXTransport Generic Device* (see section *netXTransportGetDeviceHandle* on page 58 or *netXTransportGetDeviceHandleByNumber* on page 59)

Return Values

None

6.1.20 netXTransportScheduleReconnect

Depending on the connection type (see *Connection Types* on page 14), the function schedules either a device or interface reconnect. A reconnect is processed by internally calling *netXTransportDisconnectDevice()* and *netXTransportReconnectDevice()*.

Function Call

```
int32_t netXTransportScheduleReconnect( NXTHANDLE hNetXTransportDeviceHandle,
                                        uint32_t  ulTimeout)
```

Arguments

Argument	Data type	Description
hNetXTransportDeviceHandle	NXTHANDLE	Handle to the *netXTransport Generic Device* (see section *netXTransportGetDeviceHandle* on page 58 or *netXTransportGetDeviceHandleByNumber* on page 59)
ulTimeout	uint32_t	Timeout for reconnect

Return Values

Return Values	
int	!= 0 if the function succeeds the device is accessible

6.1.21 netXTransportDisconnectDevice

Depending on the connection type (see section *Connection Types* on page 14), the function interrupts a device or interface related connection and cleans up all resources.

For more information of the device state change see section *netXTransport Device Handling* on page 73.

Function Call

```
int32_t netXTransportDisconnectDevice( NXTHANDLE hNetXTransportDeviceHandle)
```

Arguments

Argument	Data type	Description
hNetXTransportDeviceHandle	NXTHANDLE	Handle to the *netXTransport Generic Device* (see section netXTransportGetDeviceHandle on page 58 or netXTransportGetDeviceHandleByNumber on page 59)

Return Values

Return Values	
int32_t	0 on success (for possible errors see netXTransport_Errors.h)

6.1.22 netXTransportReconnectDevice

Depending on the connection type (see section *Connection Types* on page 14), the function tries to reconnect a device or interface related connection.

For more information of the device state change see section *netXTransport Device Handling* on page 73.

Function Call

```
int32_t netXTransportReconnectDevice( NXTHANDLE hNetXTransportDeviceHandle)
```

Arguments

Argument	Data type	Description
hNetXTransportDeviceHandle	NXTHANDLE	Handle to the *netXTransport Generic Device* (see section *netXTransportGetDeviceHandle* on page 58 or *netXTransportGetDeviceHandleByNumber* on page 59)

Return Values

Return Values	
int32_t	0 on success (for possible errors see *netXTransport_Errors.h*)

6.1.23 netXTransportRegisterReconnect

The function registers for the reconnect event. The event is triggered if a device becomes reattached after removal.

Function Call

```
void netXTransportRegisterReconnect( NXTHANDLE hNetXTransportDeviceHandle)
```

Arguments

Argument	Data type	Description
hNetXTransportDeviceHandle	NXTHANDLE	Handle to the *netXTransport Generic Device* (see section *netXTransportGetDeviceHandle* on page 58 or *netXTransportGetDeviceHandleByNumber* on page 59)

Return Values

None

6.1.24 netXTransportDeRegisterReconnect

The function de-registers a previously registered reconnect by calling *netXTransportRegisterReconnect()*.

Function Call

```
void netXTransportDeRegisterReconnect( NXTHANDLE hNetXTransportDeviceHandle)
```

Arguments

Argument	Data type	Description
hNetXTransportDeviceHandle	NXTHANDLE	Handle to the *netXTransport Generic Device* (see section *netXTransportGetDeviceHandle* on page 58 or *netXTransportGetDeviceHandleByNumber* on page 59)

Return Values

None

6.2 Generic Startup Process

Figure 7 shows netXTransport initialization and the target API usage, using the example of the cifX API.

The netXTransport initialization and startup consist of four steps:

- **Toolkit initialization** (see section *netXTransportInit()* on page 30)
  Initializes internal resources and registers given Translation Layers
- **Connector registration** (see section *netXTransportAddConnector()* on page 32)
  Registers the given connectors
- **Starting the netXTransport Toolkit** (see section *netXTransportStart()* on page 31)
  Opens all registered connectors and schedules the device discovering process.
- **Starting thread for the netXTransport cyclic jobs** (see section *netXTransportCyclicFunction()* on page 34)
  Starts netXTransport cyclic jobs

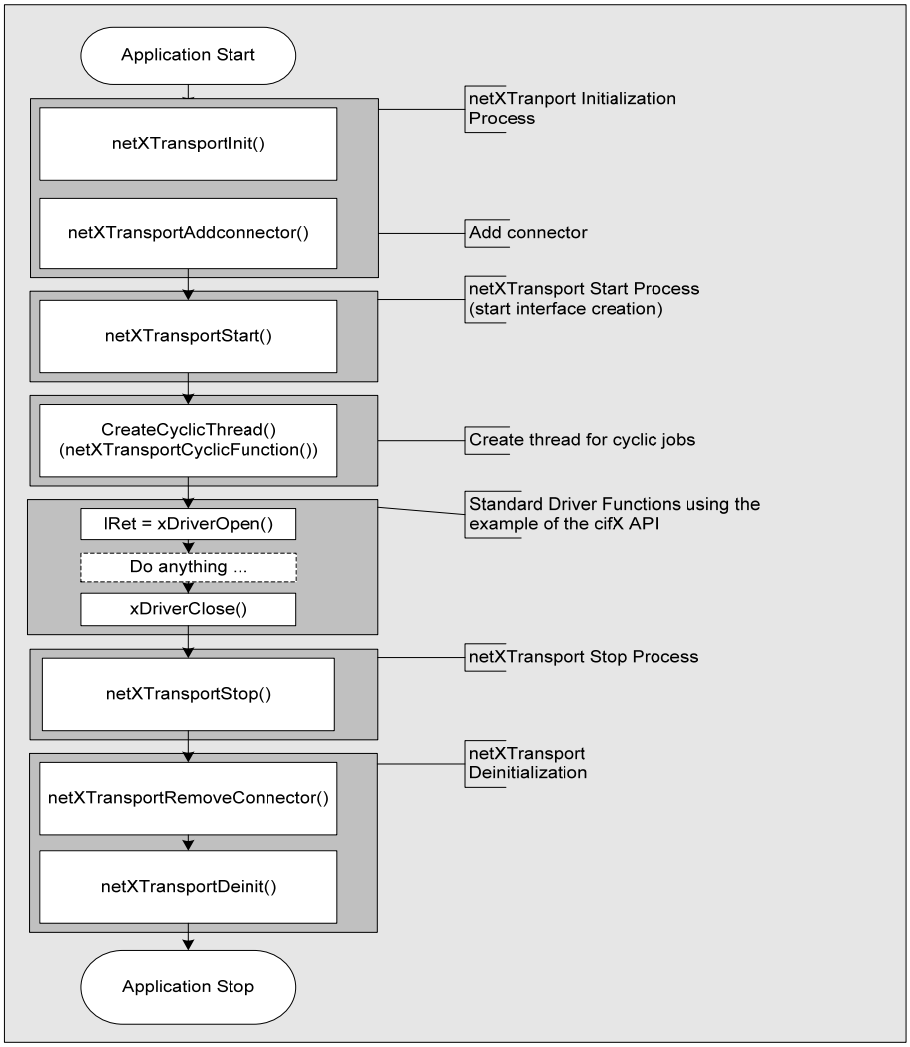

Figure 6: netXTransport: Global Initialization Process

6.2.1 Starting the netXTransport Toolkit

The netXTransport start process (see section *netXTransportStart()* on page 31) opens all connectors and creates the related interfaces.

Next step is to initialize the connection (transport layer initialization) and scanning for available devices (translation layer initialization).

During the translation layer initialization all device specific information becomes generated (see section *Translation Layer Information* on page 14).

For every detected device a netXTransport generic device will be created which contains the links to the connection related information and the device specific information, created by the translation layer.

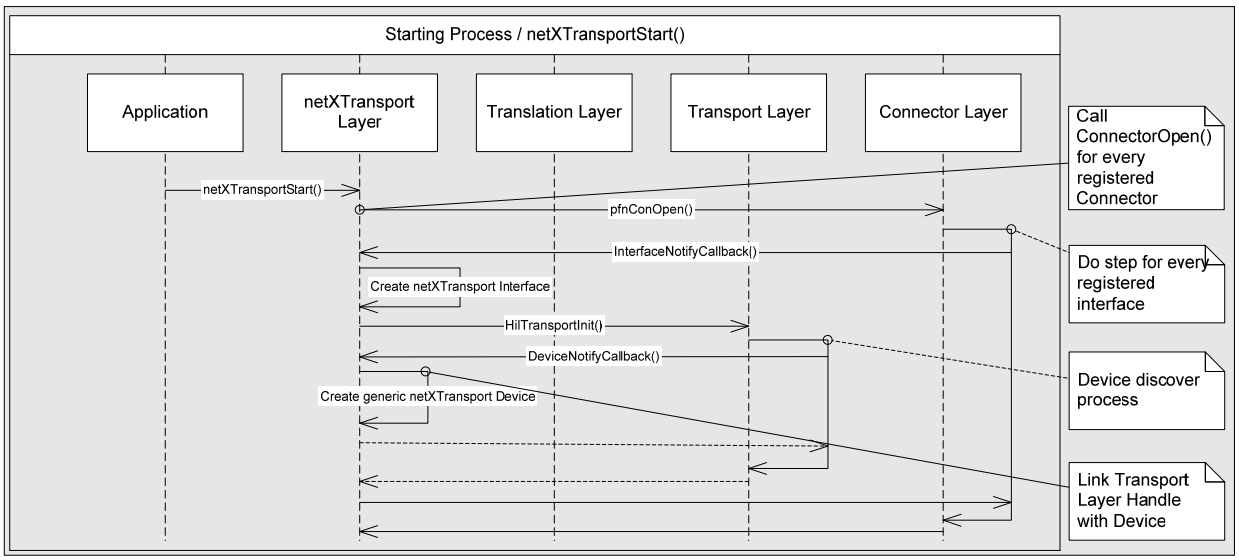

Figure 7: netXTransport: Start Process

netXTransport Start Process:

- The netXTransport module iterates over all registered connectors and calls the connector open function (see section *PFN_NETXCON_OPEN* on page 21)
- The connector prepares the required resources for all registered interfaces.
  On success, the *InterfaceNotify* callback signals a new interface.
- The netXTransport layer creates the *netXTransport Interface* (see section *Toolkit Structures: NETX_TRANSPORT_INTERFACE_T* on page 55)
- The netXTransport layer calls *HilTransportInit()* which first checks if there is a target at the newly created interface. In case of a available target, the function calls the *DeviceNotify* callback for every detected device and delivers the device related information.

  For detailed information of the *HilTransportInit()* function see Figure 8 on page 67.
- The netXTransport layer creates a *netXTransport Generic Device* for each device notified by *HilTransportInit()* (see section *NETX_TRANSPORT_GENERIC_DEVICE_T Structure* on page 54).

6.2.1.1 Processing inside HilTransportInit()

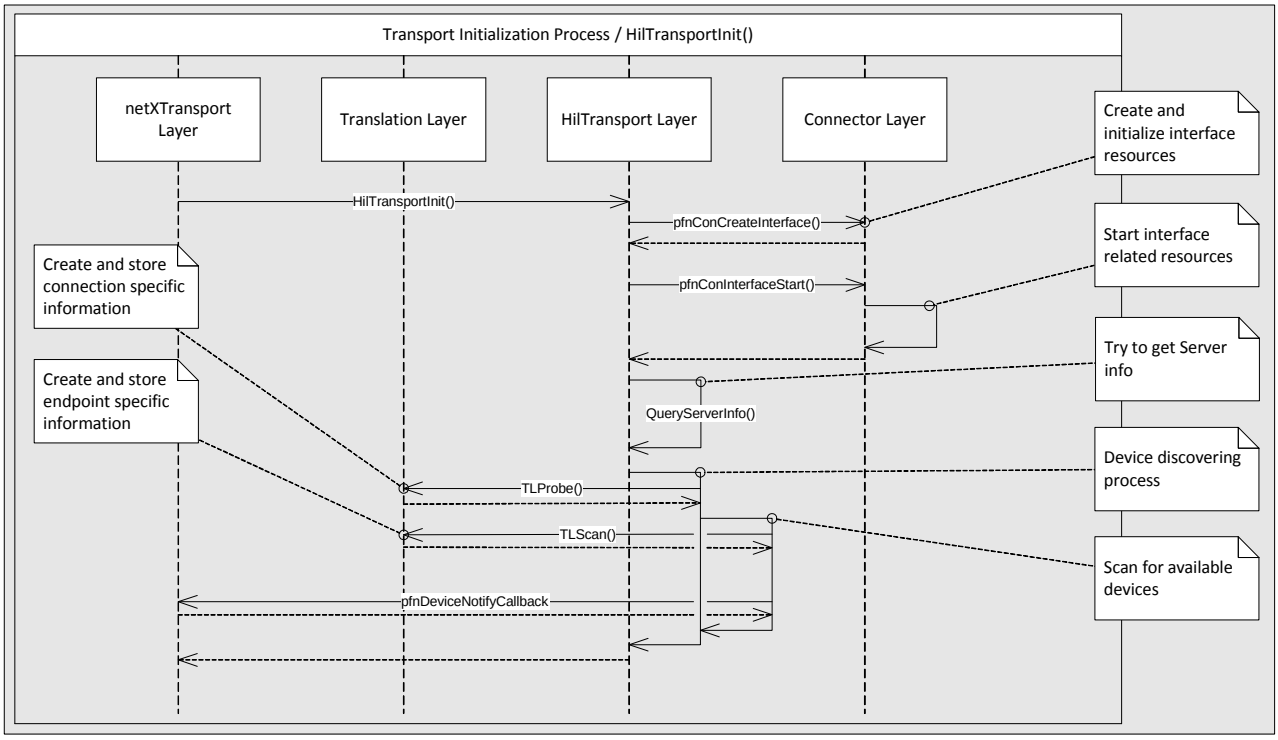

Figure 8: netXTransport: Transport Initialization Sequence

HilTransport() Processing:

- Creating a connector interface (see section *PFN_NETXCON_CREATEINTERFACE* on page 22)
- Starting the interface (see section *PFN_NETXCON_INTF_START* on page 22)
- Try retrieving the basic netXTransport server information (QueryServerInfo()) see *Hilscher Gesellschaft für Systemautomation mbH: Programming Reference* Guide, netX Diagnostic and Remote Access, Fundamentals).
- Device Discovering Process:
  - Using *TL_Probe()* to iterate over all registered translation layers and probe if there is a responsive target (send appropriate messages and wait for response). If no target is available, return with an error and skip this interface. In case of a valid response, store the target specific information (see section *Translation Layer Information* on page 14).
  - *TL_Scan()* read detailed endpoint information by iteration over all devices of a target and create their translation layer specific device information structure (see section *Translation Layer Information* on page 14).

    Return a handle to the device specific information for all successful generated devices.
  - Inform the netXTransportLayer about a new device by calling the *pfnDeviceNotifiyCallback()* function and passing the device information handle.

Note: *TL_Probe()* and *TL_Scan()* are translation layer specific functions and need to be implemented by the provider of a specific translation layer.
The two integrated *Marshaller* and *rcX-Packet Layer* are examples of implemented translation layers.

6.3 netXTransport Translation Layer Initialization

The Translation Layer initialization is done within the toolkit initialization function *netXTransportInit()* (see section *netXTransportInit()* on page 30). The passed *TL_INIT* structure provides a function pointer interface for the initialization (see section *pfnTranslationLayerInit* on page 68) and de-initialization (see section *pfnTranslationLayerDeInit* on page *69*).

The netXTransport Toolkit provides two already implemented Translation Layer types

- cifX API, based on rcX-Packet (Type = 0x100)
- cifX API, based on Marshaller (Type = 0x200)

The initialization and de-initialization process for the above noted Translation Layers (0x100/0x200) is hidden by the toolkit. To register one of the provided Translation Layers see section *netXTransport cifX API Translation Layer Initialization* on page 70.

6.3.1 pfnTranslationLayerInit

The function is called from the toolkit during initialization. Within this function all custom initialization work needs to be done. The translation layer can be registered using *netXTransportAddTranslationLayer()* (see section *netXTransportAddTranslationLayer()* on page 33). For an example refer to the already implemented solution *cifX_Marshaller_Init()* of the cifX API translation layer.

Note: If one or both of the already implemented layers (see section *netXTransport Translation Layer Initialization* on page 68) should be registered call the appropriate initialization function (see section *netXTransport cifX API Translation Layer Initialization* on page 70).

Function Call

```
int32_t pfnTranslationLayerInit( void* pvUserParam)
```

Arguments

Argument	Data type	Description
pvUserParam	void*	User parameter

Return Values

Return Values	
int32_t	0 on success (for possible errors see *netXTransport_Errors.h*)

6.3.2 pfnTranslationLayerDeInit

The function is called from the toolkit during de-initialization. Within this function the translation layer will be de-registered using *netXTransportRemoveTranslationLayer() (see* section *netXTransportRemoveTranslationLayer() on page 34).* and all custom de-initialization processes needs to be done.

For an already implemented solution see *cifX_Marshaller_DeInit()* of the cifX API translation layer.

Note: If one or both of the already implemented layers (see section *netXTransport Translation Layer Initialization* on page 68) should be de-registered call the appropriate de-initialization function (see section *netXTransport cifX API Translation Layer Initialization* on page 70).

Function Call

```
int32_t pfnTranslationLayerDeInit( void* pvUserParam)
```

Arguments

Argument	Data type	Description
pvUserParam	void*	User parameter

Return Values

Return Values	
int32_t	0 on success (for possible errors see *netXTransport_Errors.h*)

6.3.3 netXTransport cifX API Translation Layer Initialization

The netXTransport toolkit provides two already implemented Translation Layer Types

- cifX API, based on rcX-Packet (Type = 0x100)
- cifX API, based on Marshaller (Type = 0x200)

Depending on the requirements, one or both of the above noted translation layer types can be registered by calling the appropriate initialization function:

cifX API Translation Layer (Initialization)	
Function	**Description**
cifX_Marshaller_Init	Initialization of the Marshaller Layer
rcXPacket_Init	Initialization of the rcX-Packet Layer

To de-initialize a previous registered layer call the appropriate de-initialization function:

cifX API Translation Layer (De-Initialization)	
Function	**Description**
cifX_Marshaller_DeInit	De-Initialization of the Marshaller Layer
rcXPacket_DeInit	De-Initialization of the rcX-Packet Layer

These functions need to be called within the translation layer initialization/de- initialization function.

The functions are specified in the netXTransport initialization structure *TL_INIT_T* which is delivered when *netXTransportInit()* is called (see section *netXTransportInit()* on page 30).

For an example can be found under *Usage and Examples* on page 78.

6.3.3.1 cifX_Marshaller_Init

The function initializes and registers the cifX API Marshaller translation layer.

Function Call

```
int32_t cifX_Marshaller_Init( void* pvParam);
```

Arguments

Argument	Data type	Description
pvParam	void*	User parameter (currently not used)

Return Values

Return Values	
int32_t	0 on success (for possible errors see *netXTransport_Errors.h*)

6.3.3.2 rcXPacket_Init

The function initializes and registers the cifX API rcX-Packet Translation layer.

Function Call

```
int32_t rcXPacket_Init( void* pvParam);
```

Arguments

Argument	Data type	Description
pvParam	void*	User parameter (currently not used)

Return Values

Return Values	
int32_t	0 on success (for possible errors see *netXTransport_Errors.h*)

6.3.3.3 cifX_Marshaller_DeInit

The function de-registers and de-initializes the cifX API Marshaller translation layer.

Function Call

```
int32_t cifX_Marshaller_DeInit( void* pvParam);
```

Arguments

Argument	Data type	Description
pvParam	void*	User parameter (currently not used)

Return Values

Return Values	
int32_t	0 on success (for possible errors see *netXTransport_Errors.h*)

6.3.3.4 rcXPacket_DeInit

The function de-registers and de-initializes the cifX API rcX-Packet translation layer.

Function Call

```
int32_t rcXPacket_DeInit( void* pvParam);
```

Arguments

Argument	Data type	Description
pvParam	void*	User parameter (currently not used)

Return Values

Return Values	
int32_t	0 on success (for possible errors see *netXTransport_Errors.h*)

6.4 netXTransport Device Handling

The following chapter gives an overview of the lifecycle of a *netXTransport Generic Device* (see section *netXTransport Generic Device on page* 13).

netXTransport: Device State Change Diagram:

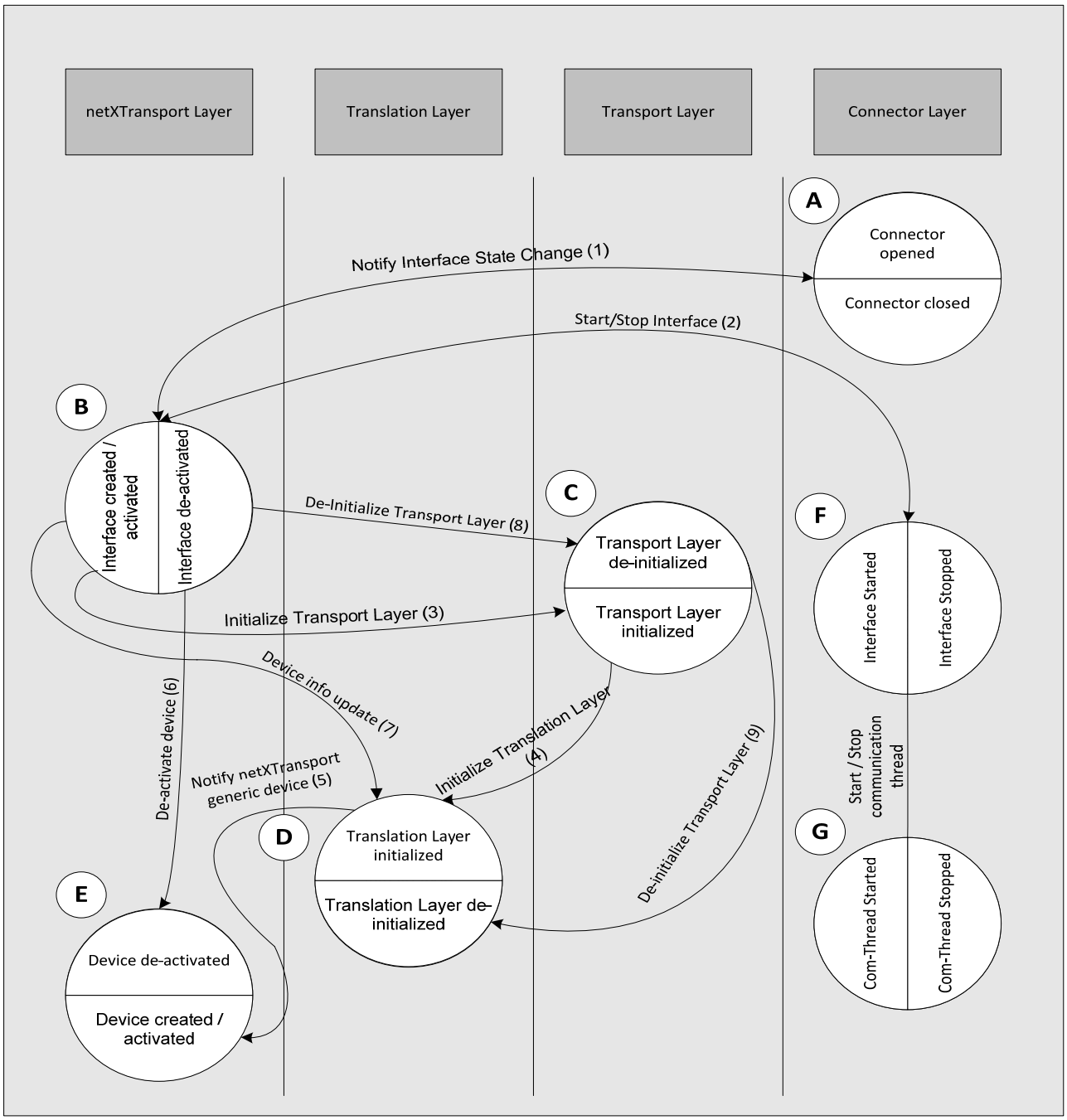

Figure 9: netXTransport: Device State Change

6.4.1 netXTransport Device Enumeration

- State A:
  The connector (Connector Layer) notifies a newly created interface, scheduled from the connector creation of the *netXTransportStart()* function.

  Note: Depending on the connector and its configuration one connector can provide more than one interface.
- State B:
  A new interface will be created.
- State F/G:
  Start the new created interface (communication thread activation)
- State C/D:
  Initialize Transport Layer and Translation Layer.
  - Search for a supported Translation Layer (see *TL_Probe()*, Figure 8 on page 67
  - Scan for all available devices (see *TL_Scan()*, Figure 8 on page 67)
- State E:
  In case of successful initialization at least one newly created device becomes notified to the netXTransport layer. Since the device is new, the netXTransport layer creates a new generic device and activates the device.

6.4.2 netXTransport Device State Change

Under several circumstances it is possible that a device changes its state from active to inactive and vice versa. In case of an inactive state the device will not be responsive anymore and the related resources (see *Translation Layer Information* and *Transport Layer Information*) are not valid any more.

The *netXTransport Toolkit* provides dedicated functions to check the device state and to lock its removal (see section *netXTransportAttachDevice* on page 60 and *netXTransportDetachDevice* on page 60).

Examples for initiating a device state change are:

- Connection Error (e.g. due to connection quality)
- Connection Timeout (e.g. device was not responsible for specified amount of time)
- None responsive device (e.g. hardware reset of the device)

Note: The netXTransport toolkit never deletes detached interfaces or devices, because a running application may hold valid interface pointers to such a device. Therefore devices which are disappeared and appeared again identified by their unique device identification number and will be re-activated.

In general a device state change can be triggered from two sides

- Connector triggered (depends on the connection type where a connector needs to be capable to detect connection interrupts)
- Manually triggered (an application triggers a device hardware reset or a timeout occurs on the device connection which is monitored and scheduled by the *netXTransport Toolkit*))

6.4.2.1 Connector triggered state change

Since the connector is not aware of any device information, connector triggered state changes belong always to an interface. Thus all devices, related to the changed interface, needs to be updated.

Connector triggered state change:

- Step 1:
  Connector notifies new interface state
- Step 6:
  In case of an interface removal the netXTransport layer updates all of the related device states and de-initializes the related Transport layer (see step 8+9) and returns.
- Step 3 - 5:
  If an interface reappears, the whole initialization process (step 3 - 5), becomes repeated and the device data will be updated.

6.4.2.2 Manually triggered state changes

In case of manually triggered state change the netXTransport Layer differentiate between the connection types (Multiple/Single device connection, see section *netXTransport* Additional Information on page 13).

State Changes on a Multiple Device Connection

To prevent influences of connections to other devices on a *Multiple Connection Interface*, the netXTransport layer only schedules an update of the device Translation Layer information.

- Step 6:
  In case of a device removal only the device state becomes updated.
- Step 7+5:
  In case of a reconnection, a Translation Layer update is scheduled to update the device specific information.

State Changes on a Single Endpoint Connection

- State B:
  In case of a removal the netXTransport Layer simulates an interface removal which first updates the device state (step 6), de-initializes the netXTransport Layer (step 8+9) and interface becomes closed (step 2).
- State B:
  If the device should be reconnected, the netXTransport Layer restarts the related interface (see Step 2) and the whole initialization process is repeated (Step 3 - 5).

6.4.2.3 Reconnect Process on Device Hardware Reset

Example of a reconnect process in case of a device hardware reset.

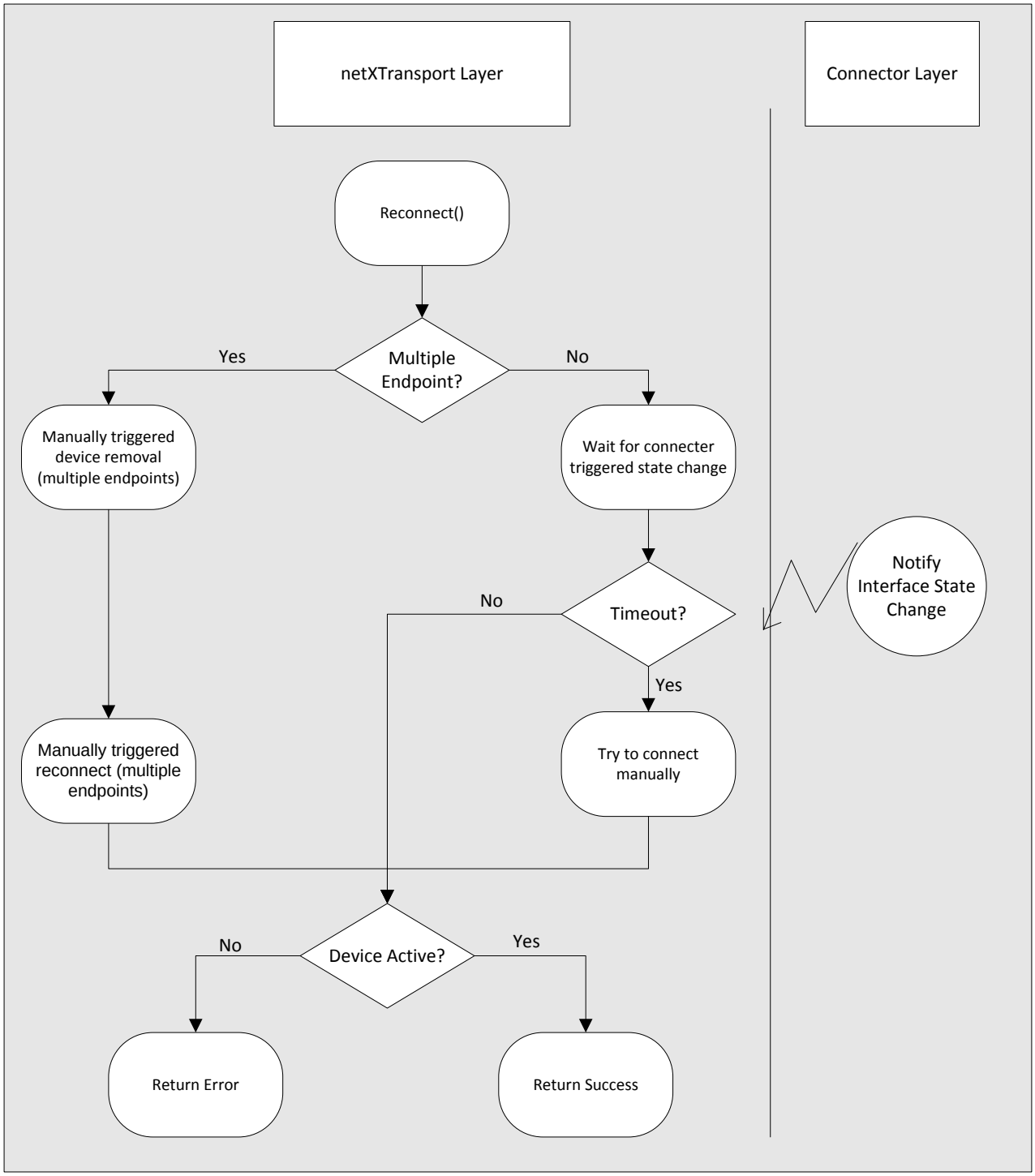

Figure 10: netXTransport: Reset Triggered Reconnect Process

7 Usage and Examples

For a complete example implementation for Windows and Linux see *'Examples'* in *Table 4: Folder Structure* on page 6.

The netXTransport initialization and startup consist of four steps:

- **Toolkit initialization** (see section *netXTransportInit()* on page 30)
  Initializes internal resources and registers given Translation Layers
- **Connector registration** (see section *netXTransportAddConnector()* on page 32)
  Registers the given connectors
- **Starting the netXTransport Toolkit** (see section *netXTransportStart()* on page 31)
  Opens all registered connectors and schedules the device discovering process.
- **Starting thread for the netXTransport cyclic jobs** (see section *netXTransportCyclicFunction()* on page 34)
  Starts netXTransport cyclic jobs

7.1 Toolkit Initialization

```
int32_t TLLayerInit( void* pvParam)
{
  int32_t lRet = NXT_NO_ERROR;

  /* since the cifX API Translation Layer is part of the toolkit, use the provided */
  /* toolkit functions to initialize and register cifX Translation Layer init and  */
  /* register Marshaller Layer */
  if (NXT_NO_ERROR == (lRet = cifX_Marshaller_Init( pvParam)))
  {
    /* init and register rcX-Packet Layer */
    lRet = rcXPacket_Init( pvParam);
  }
  return lRet;
}

void TLLayerDeInit( void* pvParam)
{
  cifX_Marshaller_DeInit( pvParam);
  rcXPacket_DeInit( pvParam);
}

int main( void)
{
  TL_INIT_T tDataLayerInit;
  int32_t   lRet = NXT_NO_ERROR;

  /* setup the netXTransport initialization structure */
  /* (Translation-Layer initialization                */
  tDataLayerInit.pfnTLInit   = TLLayerInit;
  tDataLayerInit.pfnTLDeInit = TLLayerDeInit;
  tDataLayerInit.pvData      = NULL;

  /* initialize the netXTransport toolkit and register the Translation-Layer */
  if (NXT_NO_ERROR != (lRet = netXTransportInit( &tDataLayerInit,
                                                 sizeof(tDataLayerInit))))
  {
    printf("\nError during toolkit initialization. lRet=0x%X\n", lRet);
  } else
  {
    printf("The netXTransport Toolkit is successfully initialized\n");
  }
  netXTransportDeInit();
}
```

7.2 netXTransport Connector Registration

In case of a successful connector creation (see section *Creating a netXTransport Connector* on page 19), the connector needs to be registered at the *netXTransport Toolkit* via *netXTransportAddConnector()*.

The following example code shows how to register and to de-register a custom connector.

Note: For an example implementation of the *Custom_netXConxxx()* connector functions refer to the already implemented connector for Linux (see *Connector*, *Table 4: Folder Structure* on page 6).

```
NETX_CONNECTOR_T s_tRS232Connector;

int main( void)
{
  TL_INIT_T tDataLayerInit;
  int32_t   lRet = NXT_NO_ERROR;

  /* setup the netXTransport initialization structure */
  /* (Translation-Layer initialization                */
  tDataLayerInit.pfnTLInit   = TLLayerInit;
  tDataLayerInit.pfnTLDeInit = TLLayerDeInit;
  tDataLayerInit.pvData      = NULL;

  /* setup the netXTransport generic connector interface structure */
  s_tRS232Connector.tFunctions.pfnConGetIdentifier      = Custom_netXConGetIdentifier;
  s_tRS232Connector.tFunctions.pfnConOpen               = Custom_netXConOpen;
  s_tRS232Connector.tFunctions.pfnConClose              = Custom_netXConClose;
  s_tRS232Connector.tFunctions.pfnConCreateInterface    = Custom_netXConCreateInterface;
  s_tRS232Connector.tFunctions.pfnConIntfStart          = Custom_netXConStartInterface;
  s_tRS232Connector.tFunctions.pfnConIntfStop           = Custom_netXConStopInterface;
  s_tRS232Connector.tFunctions.pfnConIntfSend           = Custom_netXConSendInterface;
  s_tRS232Connector.tFunctions.pfnConIntfGetInformation = Custom_netXConGetConfig;

  if (NXT_NO_ERROR != (lRet = netXTransportInit( &tDataLayerInit,
                                                 sizeof(tDataLayerInit))))
  {
    printf("\nError during toolkit initialization. lRet=0x%X\n", lRet);

  } else if (NXT_NO_ERROR != (lRet = netXTransportAddConnector(&s_tRS232Connector)))
  {
    printf("\nError during RS232-Connector initialization. lRet=0x%X\n", lRet);
  } else
  {
    printf("The connector is successfully registered\n");
    netXTransportRemoveConnector(&s_tRS232Connector);
  }
  netXTransportDeInit();

  return 0;
}
```

7.3 cifX API Example

```
int main( void)
{
  TL_INIT_T tDataLayerInit;
  int32_t   lRet = NXT_NO_ERROR;

  /* setup the netXTransport initialization structure */
  /* (Translation-Layer initialization                */
  tDataLayerInit.pfnTLInit   = TLLayerInit;
  tDataLayerInit.pfnTLDeInit = TLLayerDeInit;
  tDataLayerInit.pvData      = NULL;

  if (NXT_NO_ERROR != (lRet = netXTransportInit( &tDataLayerInit,
                                                 sizeof(tDataLayerInit))))
  {
    printf("\nError during toolkit initialization. lRet=0x%X\n", lRet);

  } else if (NXT_NO_ERROR != (lRet = netXTransportAddConnector(&s_tRS232Connector)))
  {
    printf("\nError during RS232-Connector initialization. lRet=0x%X\n", lRet);

  } else if (NXT_NO_ERROR != (lRet = netXTransportStart( NULL, NULL)))
  {
    printf("\nError while trying to start. lRet=0x%X\n", lRet);

  } else
  {
    /* TODO: create thread and call netXTransportCyclicFunction() */

    /* Start the cifX-API */
    /* open the driver    */
    if (CIFX_NO_ERROR == (lRet = xDriverOpen( &hDriver)))
    {
      if (CIFX_NO_ERROR == (lRet = xSysdeviceOpen( hDriver, szBoardName, &hSysdevice)))
      {
        /* do anything... */
        xSysdeviceClose( hSysdevice);
      } else
      {
        printf( "Error opening system channel! (lRet = 0x%X)\n", lRet);
      }
      xDriverClose();
    } else
    {
      printf( "Error open the cifX-Driver! (lRet = 0x%X)\n", lRet);
    }
  }
  /* TODO: stop cyclic thread (netXTransportCyclicFunction()) */
  netXTransportStop();
  netXTransportRemoveConnector(&s_tRS232Connector);
  netXTransportDeInit();
}
```

8 Error Codes

Value	Symbol	Description
0x00000000	NXT_NO_ERROR	No error
0x800Fxxxx		
0x800F0001	NXT_INVALID_POINTER	Invalid pointer (e.g. NULL) passed
0x800F0002	NXT_INVALID_HANDLE	Invalid handle passed
0x800F0003	NXT_INVALID_PARAMETER	Invalid parameter
0x800F0004	NXT_INVALID_COMMAND	Invalid command
0x800F0005	NXT_INVALID_BUFFERSIZE	Invalid buffer size
0x800F0006	NXT_INVALID_ACCESS_SIZE	Invalid access size
0x800F0007	NXT_FUNCTION_FAILED	Function failed
0x800F0008	NXT_FILE_OPEN_FAILED	File could not be opened
0x800F0009	NXT_FILE_SIZE_ZERO	File size is zero
0x800F000A	NXT_FILE_READ_ERROR	File read error
0x800F000B	NXT_FILE_WRITE_ERROR	File write error
0x800F000C	NXT_FUNCTION_NOT_AVAILABLE	Function not available
0x800F000D	NXT_BUFFER_TOO_SHORT	Buffer to short
0x800F000E	NXT_RESOURCE_NOT_FOUND	Resource not found / not available
0x800F000F	NXT_NO_MEMORY	Not enough memory
0x800Fxxxx		
0x80100001	NXT_TOOLKIT_INIT_ERROR	Toolkit initialization error
0x80100002	NXT_TL_INIT_ERROR	Translation Layer initialization error
0x80100003	NXT_CONN_INIT_ERROR	Connector Layer initialization error
0x80100004	NXT_HIL_INIT_ERROR	Transport Layer initialization error
0x80100005	NXT_DUPLICATE_CONN_IDENTIFIER	Duplicate connector identifier found
0x80100006	NXT_DUPLICATE_CONN_UUID	Duplicate connector UUID found
0x80100007	NXT_DUPLICATE_TL_DATATYPE	Duplicate Translation Layer for one Data-Type
0x80100008	NXT_SEND_TIMEOUT	Send timeout
0x80100009	NXT_RECV_TIMEOUT	Receive timeout
0x8010000A	NXT_TRANSACTION_CANCELLED	Transaction cancelled
0x8010000B	NXT_UNSUPPORTED_DEVICE	Unsupported device
0x8002xxx		

Table 20: Error Codes

9 Appendix

9.1 List of Tables

9.2 List of Figures

9.3 Contacts

Headquarters

Germany
Hilscher Gesellschaft für
Systemautomation mbH
Rheinstrasse 15
65795 Hattersheim
Phone: +49 (0) 6190 9907-0
Fax: +49 (0) 6190 9907-50
E-Mail: info@hilscher.com

Support
Phone: +49 (0) 6190 9907-99
E-Mail: de.support@hilscher.com

Subsidiaries

China
Hilscher Systemautomation (Shanghai) Co. Ltd.
200010 Shanghai
Phone: +86 (0) 21-6355-5161
E-Mail: info@hilscher.cn

Support
Phone: +86 (0) 21-6355-5161
E-Mail: cn.support@hilscher.com

France
Hilscher France S.a.r.l.
69500 Bron
Phone: +33 (0) 4 72 37 98 40
E-Mail: info@hilscher.fr

Support
Phone: +33 (0) 4 72 37 98 40
E-Mail: fr.support@hilscher.com

India
Hilscher India Pvt. Ltd.
New Delhi - 110 065
Phone: +91 11 26915430
E-Mail: info@hilscher.in

Italy
Hilscher Italia S.r.l.
20090 Vimodrone (MI)
Phone: +39 02 25007068
E-Mail: info@hilscher.it

Support
Phone: +39 02 25007068
E-Mail: it.support@hilscher.com

Japan
Hilscher Japan KK
Tokyo, 160-0022
Phone: +81 (0) 3-5362-0521
E-Mail: info@hilscher.jp

Support
Phone: +81 (0) 3-5362-0521
E-Mail: jp.support@hilscher.com

Korea
Hilscher Korea Inc.
Seongnam, Gyeonggi, 463-400
Phone: +82 (0) 31-789-3715
E-Mail: info@hilscher.kr

Switzerland
Hilscher Swiss GmbH
4500 Solothurn
Phone: +41 (0) 32 623 6633
E-Mail: info@hilscher.ch

Support
Phone: +49 (0) 6190 9907-99
E-Mail: ch.support@hilscher.com

USA
Hilscher North America, Inc.
Lisle, IL 60532
Phone: +1 630-505-5301
E-Mail: info@hilscher.us

Support
Phone: +1 630-505-5301
E-Mail: us.support@hilscher.com