Programming Reference Guide

netX Diagnostic and Remote Access

Host Device

V0.9.6.x

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC100407PR03EN | Revision 3 | English | 2013-09 | Released | Public

Table of Contents

© Hilscher, 2010-2011

1 Introduction

1.1 About this Document

The '*netX Diagnostic and Remote Access'* services define a communication protocol for accessing netX based target systems and remote workstations containing a netX based hardware.

The basic overview and technical fundamentals are specified in the '*netX Diagnostic and Remote Access - Fundamentals*' manual.

The host application accessing the target does not need to know anything about the underlying protocol and can still access the default cifX driver API. Hilscher offers some standard communication interfaces, but offers also a way for customer to integrate their own interface by attaching themselves to the communication protocol (*Hilscher Transport Mechanism*).

1.2 List of Revisions

Rev	Date	Name	Chapter	Revision
1	2010-04-16	RM, PL, SS	all	Created
2	2011-06-30	HH		Graphics optimized
3	2013-09-04	SD	3	Section *Creating a Custom Connector* revised, expanded and example code added.

Table 1: List of Revisions

© Hilscher, 2010-2011

1.3 Terms, Abbreviations and Definitions

Term	Description
API	Application Programming Interface
AP (task)	Application (task) on top of a communication stack
cifX	Communication Interface based on netX
comX	Communication Module based on netX
CDC	Communication Device Class (USB terminology)
CMD	Command
DPM	Dual-Port Memory
FW	Firmware
LSB	Least Significant Bit or Byte
MBX	Mailbox
MSB	Most Significant Bit or Byte
ODM	Online Data Manager
OS	Operating System
PLC	Programmable Logic Controller
rcX	Real-time Operating System running on the netX chip
SHM	SHared Memory (virtual DPM equivalent)

Table 2: Terms, Abbreviations and Definitions

1.4 References

For further information about DPM and SHM handling, please refer to the following specifications:

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

[2] Hilscher Gesellschaft für Systemautomation mbH: netX Dual-Port Memory Interface for netX based Products, revision 12, english, 2013.

[3] Hilscher Gesellschaft für Systemautomation mbH: Programming Reference Guide, CIFX API, revision 2, english, 2013.

Table 3: References

1.5 Intended Audience

This specification is for internal use by hardware, firmware and software developers, by testers, support personnel and project managers.

© Hilscher, 2010-2011

1.6 Legal Notes

1.6.1 Copyright

© 2010-2011 Hilscher Gesellschaft für Systemautomation mbH

All rights reserved.

The images, photographs and texts in the accompanying material (manual, accompanying texts, documentation, etc.) are protected by German and international copyright law as well as international trade and protection provisions. You are not authorized to duplicate these in whole or in part using technical or mechanical methods (printing, photocopying or other methods), to manipulate or transfer using electronic systems without prior written consent. You are not permitted to make changes to copyright notices, markings, trademarks or ownership declarations. The included diagrams do not take the patent situation into account. The company names and product descriptions included in this document may be trademarks or brands of the respective owners and may be trademarked or patented. Any form of further use requires the explicit consent of the respective rights owner.

1.6.2 Important Notes

The 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 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 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 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.

1.6.3 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.

1.6.4 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.

© Hilscher, 2010-2011

2 netX Marshaller Context

The netX Marshaller is a software component that allows accessing netX based target systems via different physical interfaces in order to perform

- Diagnostics
- I/O data monitoring
- System management (channel init, reset device, etc.)
- Firmware and configuration download
- Run-time data access (speed depends on the connection)

The component acts as a client on the host system, supporting the Hilscher transport protocol and services (see '*netX Diagnostic and Remote Access - Fundamentals*' manual).

Overview:

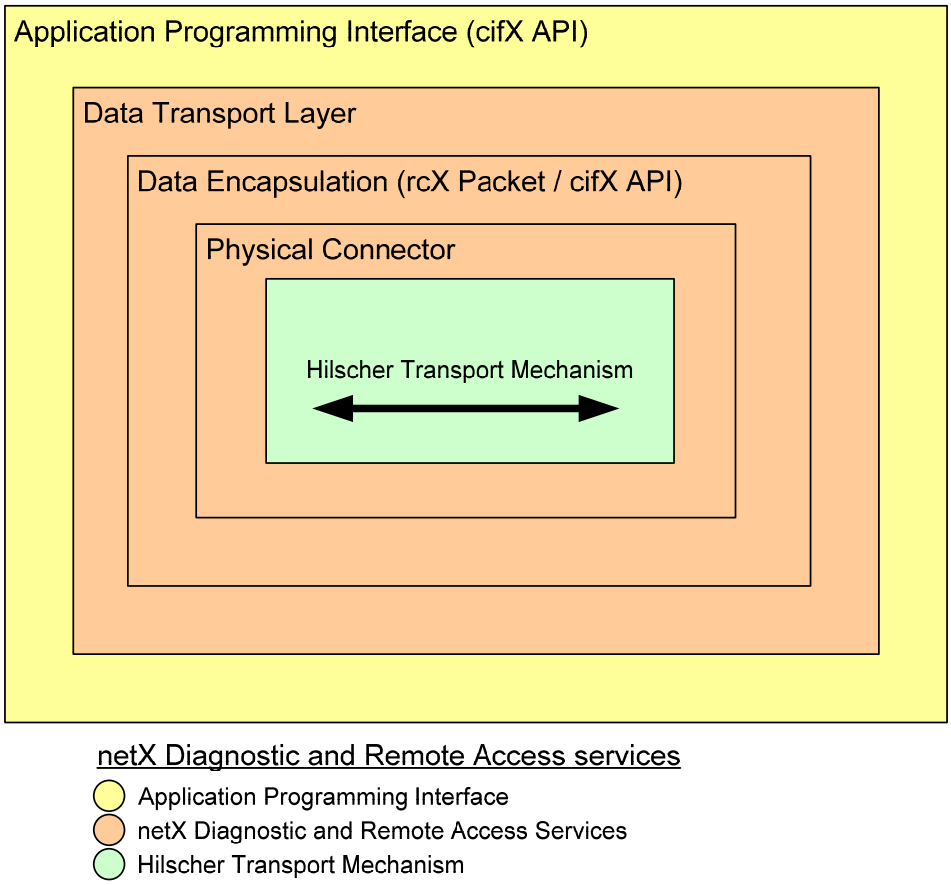

Figure 1: Overview Structure

© Hilscher, 2010-2011

The component is provided as several Windows DLLs (Dynamic Link Library). A main DLL (netXDiagRemote.dll), containing the '*netXMarshaller*' function and a number of underlying DLL '*Connector*' DLLs (e.g. RS232Connector.dll) providing the physical connection functions.

A user application can use the netXDiagRemote.dll to access a netX target via the defined cifX API functions.

The '*netX Diagnostic and Remote Access*' services providing different target access possibilities:

- via standard rcX packets
- via cifX API functions calls which are executed on the target

The type of access is controlled from the target, which defines the possible access. Both communication types are using the same outer framing (Hilscher Transport Mechanism). Inside, the frame contains either a standard rcX packet or a cifX API function call.

Encoding and decoding of the remote access data packets is transparently done on the host and the target system, inside the '*netXMarshaller*' and transport layers. When using the cifX API remotely the underlying host marshaller component will encode the request and send them to the target device. The target decodes the request and executes them locally.

This makes access to the device independent from the underlying physical connection. The exact encoding of the data is defined in '*netX Diagnostic and Remote Access - Fundamentals*'.

© Hilscher, 2010-2011

2.1 General Software Structure

The '*netXMarshaller*' consists of three components.

Component	Description
netXMarshaller	Manages all physical connections and offers decoding and encoding functions for the possible transport types.
Transport	The transport layer encodes the cifX API calls to the correct transport format before the data are passed to the connector. Response data are encoded and delivered to the application.
Connector	Connects the marshaller main module to a physical interface and provides a byte streaming interface for incoming and outgoing data. **Note:** Connectors need to be written during the porting process or when using a proprietary interface

Table 4: netXMarshaller Components

The following picture gives an overview of the internal structure of the netXMarshaller component showing only the major function blocks.

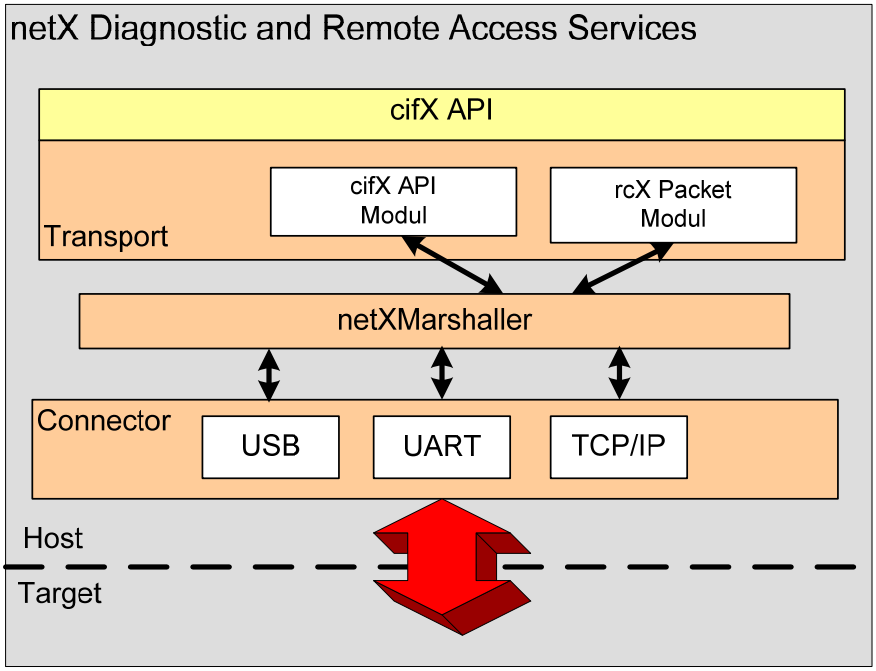

Figure 2: netXMarshaller Structure

© Hilscher, 2010-2011

2.2 Implementation

The interface needs to be useable for kinds of user application which are using the standard cifX API, so it is defined to be independent of the used physical interface, transport and data type. Before the communication can be start, the interface negotiates the supported data type of the device.

Depends on the result of the supported data type a special data type will be used for the data transfer.

The corresponding Physical Layer, which supports a special kind of physical interfaces, is used to transfer data. The “Hilscher Transport Header” realizes a standard for the way of transfer.

Host Side

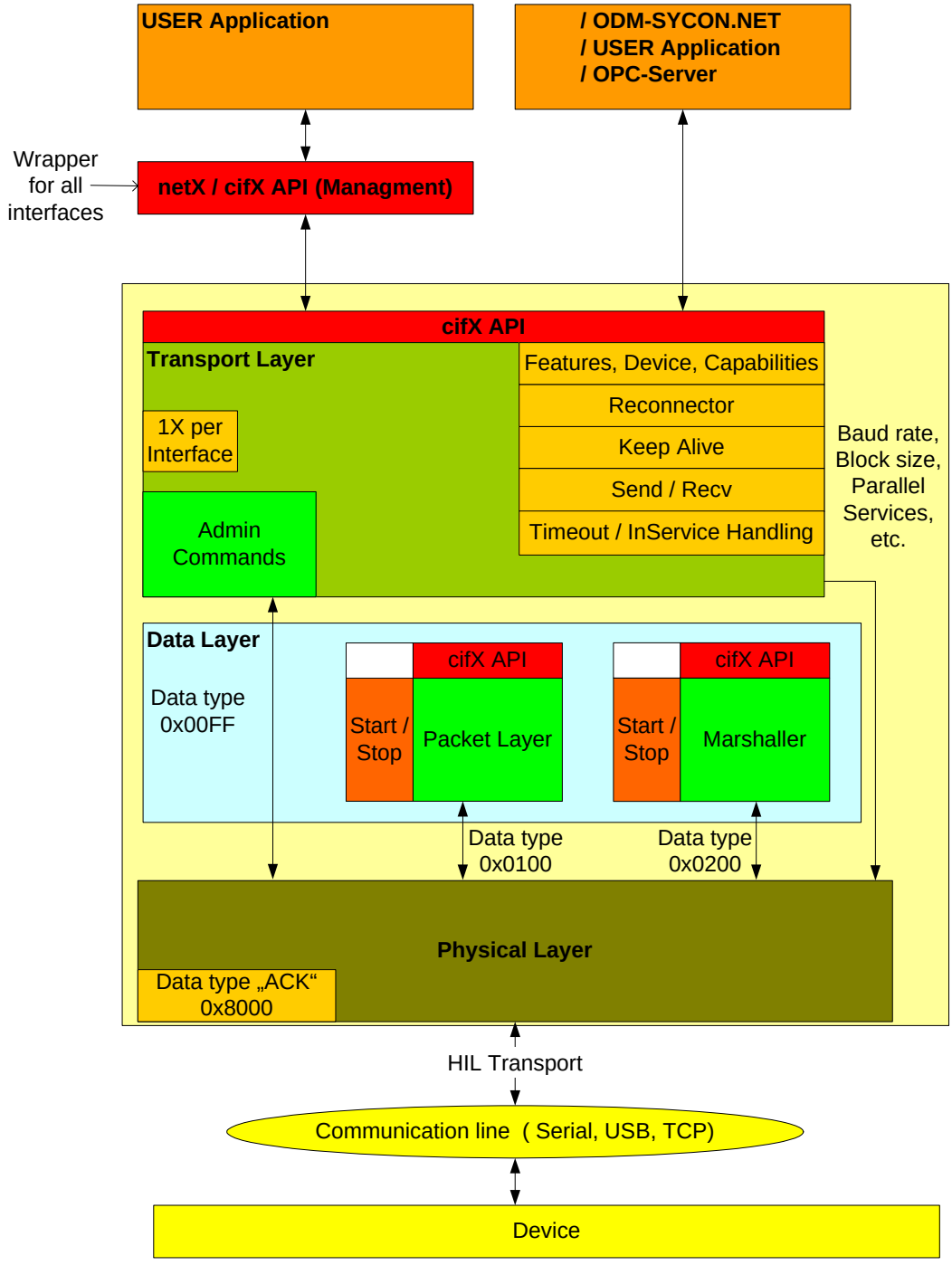

Figure 3: Layout Overview

© Hilscher, 2010-2011

2.2.1 DeviceHandler

The DeviceHandler manage the different kinds of physical interfaces. Depends on the available physical interfaces the DeviceHandler prepare the Physical Layer of a connection.

2.2.2 Transport Layer

This layer managed the connection for one physical interface and supports the cifX API to the application side. Before the communication can be start this layer checks the available data types (communication standards).

The following state machine shows the separate steps of negotiation of usable data types.

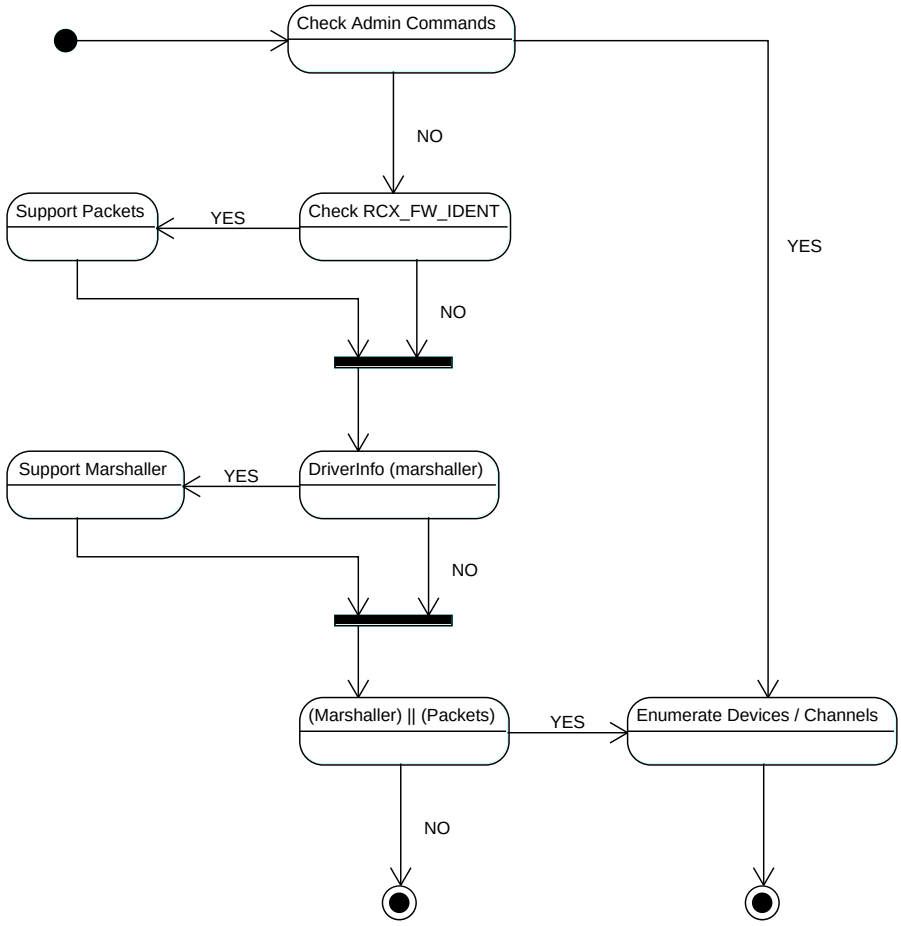

Figure 4: State Machine negotiate supported Data Type

All settings and device information for each connected device are managed by this layer. This Layer also includes the “Keep Alive mechanism”, “reConnector” als well as data queues for the data transfer.

The “reConnector” is used after a connection was falling down. The “reConnector” will be use every time the “User-Application” wants to send data, after the connection was falling down. Additional this layer manages the lower layers for each device.

The “Keep Alive mechanism is use every time the “Keep Alive mechanism” will be supported.

The major task of this layer is the support of the CifX - API for the user application. Additional, this layer includes the “Admin Command” standard for the connection handling to a device as well as the data handling of the send and received data.

© Hilscher, 2010-2011

2.2.3 Data Layer

Depends on the Device-Side supported data type, Marshaller-, rcX-Packets are used. This layer passes the packets to the “Physical Layer”, which sends the packets conformable to the transport settings.

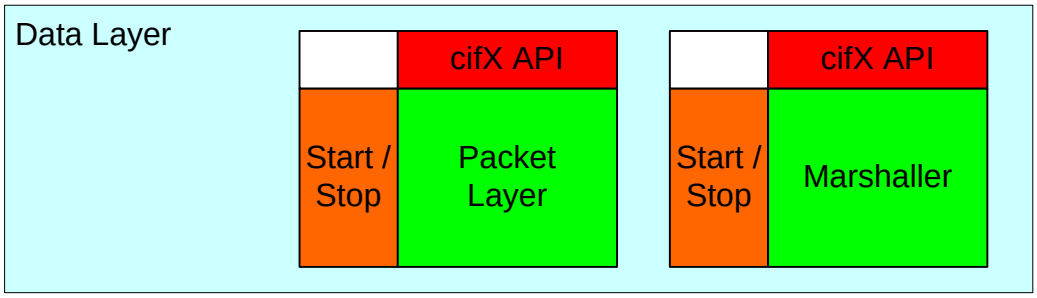

Figure 5: Packet Layer Overview of usable Packet Types

All supported packet types includes inititalization and de-initialization functionality this allowed separate to control the packet types. The packet types support also the cifX API this simplify the calls for the Transport Layer.

© Hilscher, 2010-2011

2.2.4 Physical Layer

The Physical Layer supervises the physical interface for the data transfer to the device. After the “Transport Layer” got some information from the device, the “Physical Layer” should be configured from the “Transport Layer”. The layer registers the open connection to the device by the upper layers.

The Physical Layer should be configuring, before a connection will be open, from the Transport Layer. Before the Physical Layer got a new configuration for the connections, al connections will be open by using of default values.

Each special implementation of the Physical Layer includes the Acknowledge routines of the transport kind Marshaller.

© Hilscher, 2010-2011

2.3 Layout and corresponding between the Layers

The layout is inspirational by the '*Open Systems Interconnection Reference Model*'. All the different layers are used to realize different duties and responsibilities by the handling and communication to an interface also a device. About the possibility to change the implementation of the “Transport Layer” another interface will be supported. If a separate wrapper is used the “Application Layer” can be abstract to a real cifX API. On this layout the reduction of using is that only one instance should be used for on interface.

This means that only one application should communicate by using this layout at one point of time.

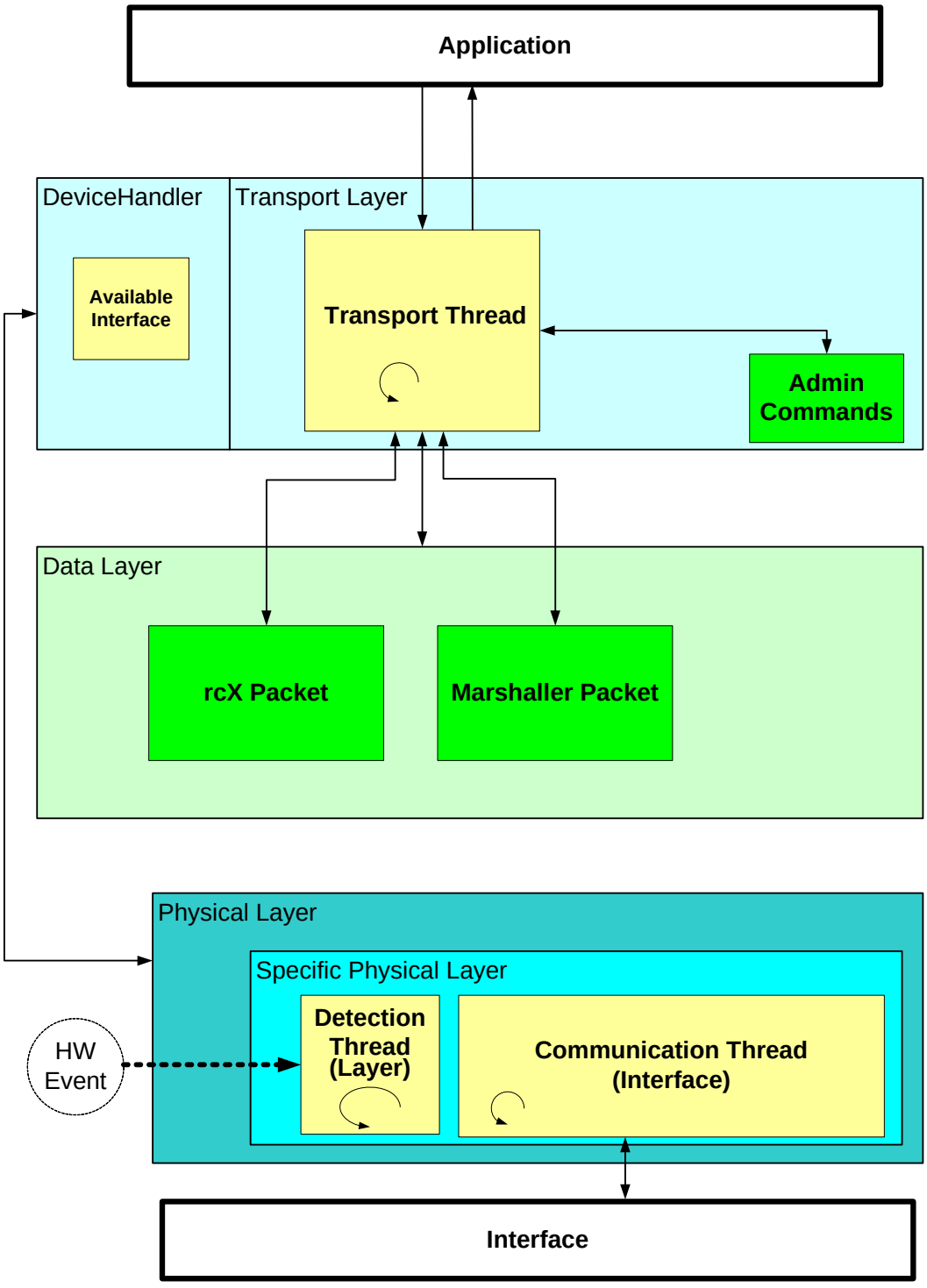

Figure 6: Relationships between the Layers

© Hilscher, 2010-2011

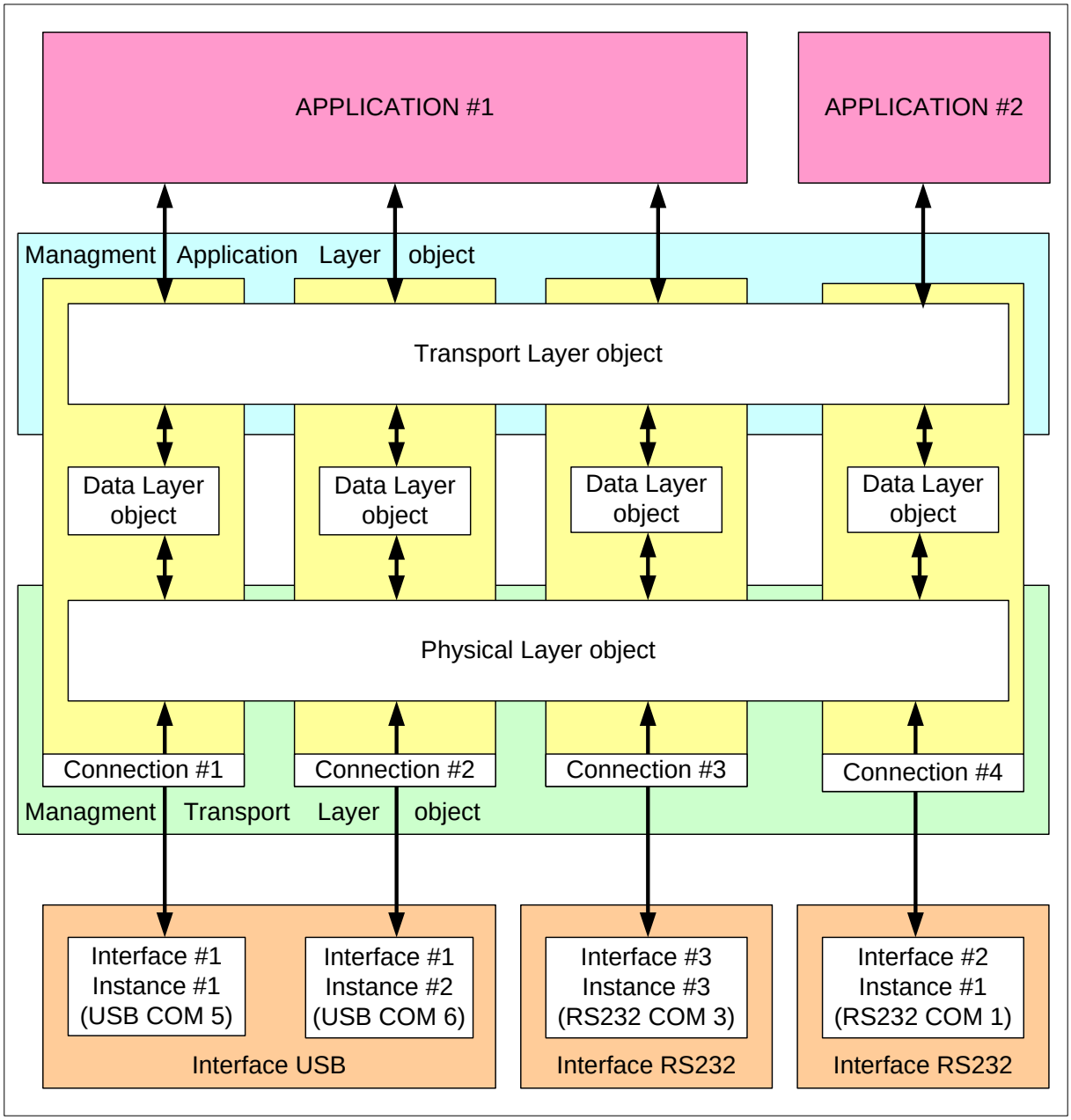

Figure 7: Overview parallel supported Connections

© Hilscher, 2010-2011

2.4 Layer Interaction on Connection Establishment

The figure below illustrates the interaction between the different layers of the netXMarshaller during the establishment of a device connection.

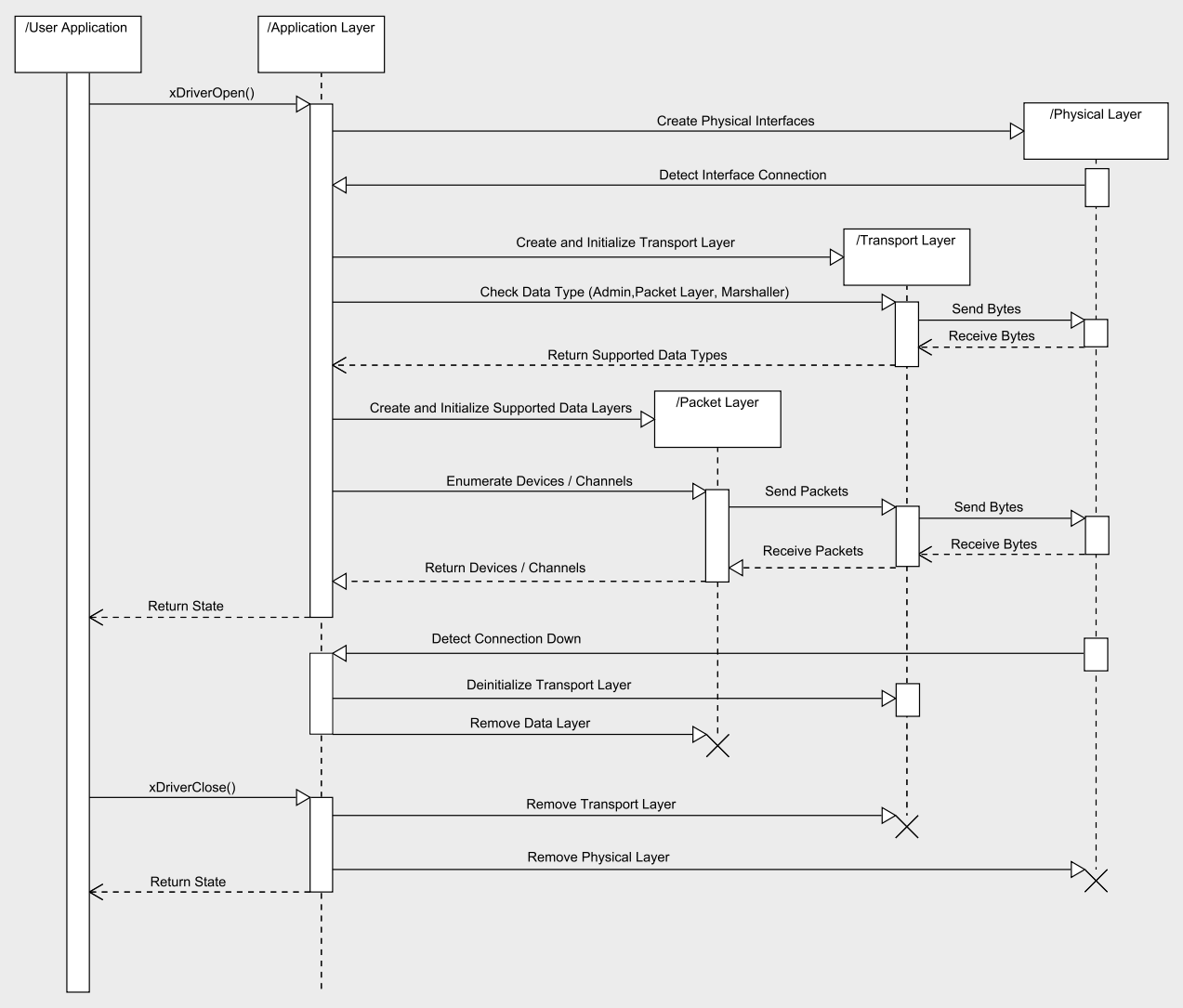

Figure 8: Layer Interaction on Connection Establishment

© Hilscher, 2010-2011

2.5 Interface Monitor

The marshaller main module provides a mechanism, which monitors the utilization of the interfaces provided by the connector modules. If the interface is not used by an application for a period of time, the interface monitor initiate the disconnection process. Subsequent accesses to the interface causes a re-establishment of the connection. The figure below illustrates this mechanism.

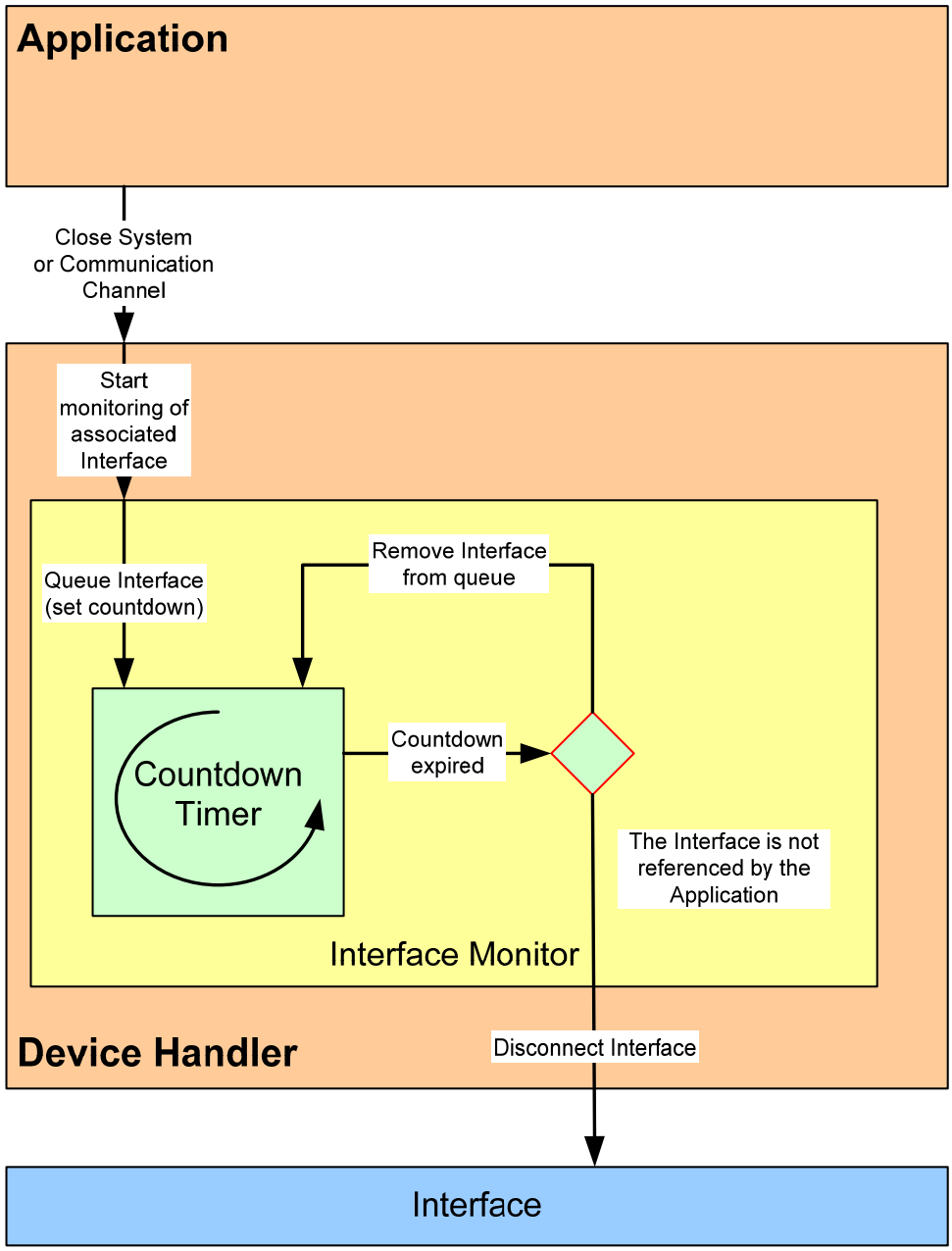

Figure 9: Interface Monitor

© Hilscher, 2010-2011

3 Creating a Custom Connector

The host implementation of the '*netX Diagnostic and Remote Access*' services allows to add custom connector modules for additional physical interfaces.

The following chapter describes the available resources (source code and interfaces) and the creation of a custom connector.

3.1 Directory Structure

Directory		Contents
Documentation		Documentation
netXConnector		Development directory for connector modules
	ConnectorAPI	API definition files ConnectorAPI.def (DLL export definitions) ConnectorAPI.h (API definition file) ConnectorAPI.cpp (Example DLL interface function module) ConnectorDLL.cpp (Example DLL main module) ConnectorConfig.cpp (Configuration class) ConnectorConfig.h (Configuration class header) netXConnectorErrors.mc (Message compiler file for default connector API errors)
	RS232Connector	A working USB/RS232 connector example
	TCPConnector	A working TCP connector example
	Documentation	Doxygen documentation of the Connector API

Table 5: Connector API - Directory Structure

© Hilscher, 2010-2011

3.2 Functions

This section describes the public functions of the physical connector, which forms the interface to the transport layer.

The connector function interface can be divided into mandatory and optional function types.

The mandatory functions are at least required for an operational netXTransport Connector. The optional functions are for configuration purposes, the content and scope depend on custom needs.

Essential Function Interface	
Function Pointer definition	**Description**
PFN_NETXCON_GETIDENTIFIER	Return connector specific identifier (page 21).
PFN_NETXCON_OPEN	Open and initialize Connector (page 22).
PFN_NETXCON_CLOSE	Close and de-initialize Connector (page 23).
PFN_NETXCON_CREATEINTERFACE	Create and initialize interface (page 24).
PFN_NETXCON_INTF_START	Start interface (ready for communication) (page 25).
PFN_NETXCON_INTF_STOP	Stop interface (page 26).
PFN_NETXCON_INTF_SEND	Send data (page 27).
Configuration Interface (optional - access needs to be implemented by the user)	
Function Pointer definition	**Description**
PFN_NETXCON_INTF_GETINFORMATION	Return interface specific information (page 34).
PFN_NETXCON_GETINFORMATION	Return connector specific information (page 30).
PFN_NETXCON_GETCONFIG	Return connector specific configuration (page 32).
PFN_NETXCON_SETCONFIG	Set connector specific configuration (page 33).
PFN_NETXCON_CREATEDIALOG	Create configuration dialog (page 35).
PFN_NETXCON_ENDDIALOG	Close configuration dialog (page 36).

© Hilscher, 2010-2011

The following figure shows the calling sequence of the particular connector functions and its internal processing using the example of the netXTransport Toolkit.

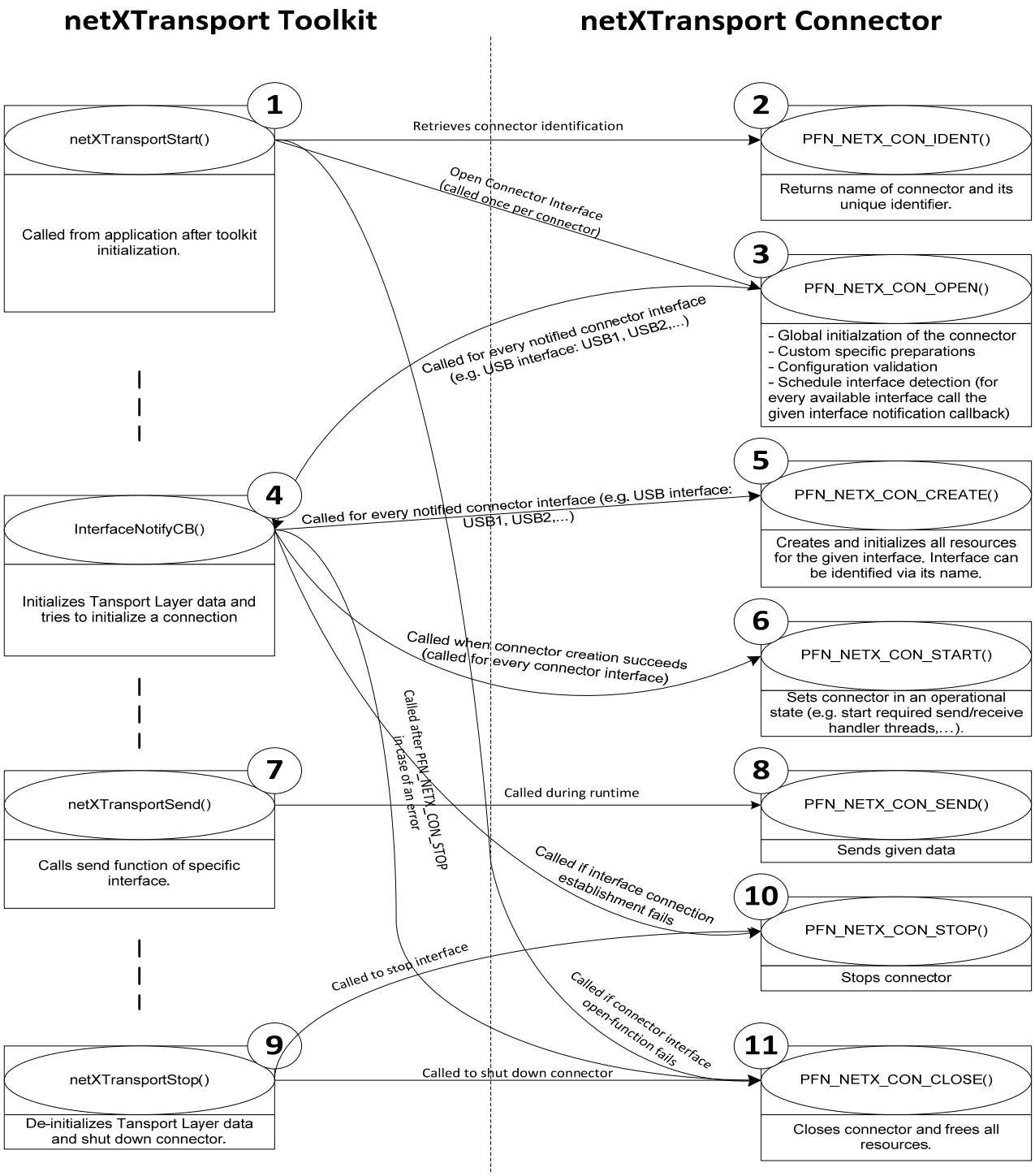

Figure 10: netXTransport Connector - Calling Sequence

© Hilscher, 2010-2011

3.2.1 netXConGetIdentifier

The function returns the name and a unique identifier of the connector. The UUID is used to prevent multiple connector registration for the same interface. There is no restriction how to generate a UUID. To be consistent the name of the connector should reappear in the particular interface names notified in *netXConOpen()* (see section *netXConOpen* on page 22).

Example for TCP connector:

Connector name (returned by *netXConGetIdenifier()*) : “TCP”

Interface names (notified by *netXConOpen()*) : “TCP0”, “TCP1”, “TCP2”, …

Function call:

```
int32_t netXConGetIdentifier(   char* szIdentifier, NETX_TRANSPORT_UUID_T* ptUUID);
```

Arguments:

Argument	Data type	Description
szIdentifier	char*	Pointer to buffer receiving connector name
ptUUID	NETX_TRANSPORT_UUID_T*	Pointer to buffer receiving connector UUID

Return Values:

Return Values	
NXCON_NO_ERROR	Connector opened successfully
NXCON_DRV_INVALID_POINTER	Invalid pointer reference

Example of netXConGetIdentifier for TCP connector:

```
int32_t netXConGetIdentifier(   char* szIdentifier, NETX_TRANSPORT_UUID_T* ptUUID)
{
  if(!szIdentifier)
    return NXCON_DRV_INVALID_POINTER;

  if(!ptUUID)
    return NXCON_DRV_INVALID_POINTER;

  /* Copy identifier of the connector (e.g. for TCP interface ‘TCP’) */
  strcpy( szIdentifier, “TCP”);

  /* Copy UUID of the connector */
  Memcpy( ptUUID, &g_MyConnectorUUID, sizeof(g_MyConnectorUUID));

  return NXCON_NO_ERROR;
}
```


© Hilscher, 2010-2011

3.2.2 netXConOpen

The function initializes all global connector specific resources. If the connector provides more than one interface all required resources should be allocated at this point. If a connector provides a configurable interface, the configuration should be evaluated and setup within the *netXConOpen* function.

Every available connector interface needs to be registered by notifying it to the upper layer using the notify callback *pfnDevNotifyCallback*. The delivered user data *pvUser* has to be passed through to the upper layer. For more information of the notify callback see section *PFN_NETXCON_DEVICE_NOTIFY_CALLBACK* on page 28.

Function call:

```
int32_t netXConOpen (     PFN_NETXCON_DEVICE_NOTIFY_CALLBACK pfnDevNotifyCallback,
                          void* pvUser)
```

Arguments:

Argument	Data type	Description
pfnDevNotifyCallback	PFN_NETXCON_DEVICE_NOTIFY_CALLBACK	Function pointer for the device notification callback. (See section 0)
pvUser	void*	User pointer

Return Values:

Return Values	
NXCON_NO_ERROR	Connector opened successfully
NXCON_DRV_WAS_OPENED_BEFORE	The Connector is already opened
NXCON_DRV_INIT_ERROR	The opening of the Connector failed due to initialization errors
NXCON_DRV_INVALID_PARAMETER	Invalid pointer reference (pfnDevNotifyCallback)

Example of netXConOpen for TCP connector:

```
MY_TCP_INFO_T tTCPMyInfo;

int32_t netXConOpen (     PFN_NETXCON_DEVICE_NOTIFY_CALLBACK pfnDevNotifyCallback,
                          void* pvUser)
{
  MY_IFC_INFO_T* ptMyInterfaceInfo;
  /* TODO: global initialization */
  /* GetMyTCPConnectorInfo()(custom defined) gets the configuration and creates */
  /* for every valid configuration an interface (e.g. TCP0=192.168.178.3,       */
  /* TCP1=10.11.5.7,…)                                                          */
  while(GetMyTCPConnectorInfo(&ptMyInterfaceInfo))
  {
    /* Store the interface information for later usage (AddTCPInterface() custom)*/
    AddTCPInterface( &tTCPMyInfo, ptInterfaceInfo);
    /* notify all configured interfaces (TCP0, TCP1,…) */
    pfnDevNotifyCallback( tInterfaceInfo->szName, /* interface name (e.g. TCP0) */
                          eATTACHED,              /* event to be notified       */
                          (void*)ptInterfaceInfo, /* my private info            */
                          pvUser);                /* upper layer private info   */
  }
  Return NXCON_NO_ERROR;
}
```


© Hilscher, 2010-2011

3.2.3 netXConClose

Close the connector and disconnect all associated interfaces.

Function call:

```
int32_t netXConClose (    void)
```

Arguments:

Argument	Data type	Description
none		

Return Values:

Return Values	
NXCON_NO_ERROR	Connector closed successfully
NXCON_DRV_NOT_INITIALIZED	The Connector is not initialized

Example:

```
int32_t netXConClose (    void)
{
  MY_IFC_INFO_T* ptMyInterfaceInfo;

  /* free all resources */
  while(ptMyInterfaceInfo = GetInterface( &tTCPMyInfo))
  {
    /* check if the interface is currently running */
    if (ptMyInterfaceInfo ->fRunning == 1)
    {
      /* notify all configured interfaces (e.g. TCP0, TCP1,…) */
      pfnDevNotifyCallback( ptMyInterfaceInfo->szName,
                            eDETTACHED,
                            ptInterfaceInfo,
                            pvUser);
    }
    FreeMyInterface( &tTCPMyInfo, ptMyInterfaceInfo)
  }
  Return NXCON_NO_ERROR;
}
```


© Hilscher, 2010-2011

3.2.4 netXConCreateInterface

The function netXConCreateInterface creates an interface by its name. While *netXConOpen()* allocates and initializes resources in a more global context *netXConCreateInterface()* allocates and initializes all resources for a specific interface. If the initialization succeeds the function returns a pointer to its private interface information. This information will be delivered when the connector start/stop/send functions are called.

The name given by *szDeviceName* will be one of the interface names notified in *netXConOpen(),* so the interface and its appropriate configuration can be identified via its name.

Function call:

```
void* netXConCreateInterface( const char* szDeviceName)
```

Arguments:

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

Return Values:

Return Values
If the creation of the interface succeeds, the return value is the pointer reference to the related interface object. If the creation of the interface fails, the return value is NULL.

Example:

```
void* netXConCreateInterface( const char* szDeviceName)
{
  MY_IFC_INFO_T* ptMyInterfaceInfo;
  void*          pvRet;

  /* try to find the requested interface and get its information   */
  /* gatherd in netXTConOpen() (custom defined: FindMyInterface()) */
  if (ptMyInterfaceInfo = FindMyInterface( szDeviceName))
  {
    /* check if we have found a valid configuration for the current interface */
    /* (custom defnied: ValidateTCPInfo())                                    */
    if (ValidateTCPInfo(ptMyInterfaceInfo->TCPConfig))
    {
      /* configuration is valid return pointer to prive connector info */
      pvRet = ptMyInterfaceInfo;
    } else
    {
      pvRet = NULL;
    }
  }
  return pvRet;
}
```


© Hilscher, 2010-2011

3.2.5 netXConStartInterface

Setup the connector interface into a working state. If a call to this function succeeds, the upper layer expects sending and receiving of data via this interface is possible. The interface the call belongs to, can be identified via *pvInterface,* the interface private information (see section *netXConCreateInterface* on page 24).

The given receive callback *pfnRecvData* is required to notify the incoming data to the upper layer. *pvUser* is a administrative structure of the upper layer and needs to be passed through.

For more information of the receive callback see section *PFN_NETXCON_DEVICE_RECEIVE_CALLBACK* on page 29.

Function call:

```
int32_t netXConStartInterface( void* pvInterface,
                           PFN_NETXCON_DEVICE_RECEIVE_CALLBACK pfnRecvDat, void* pvUser)
```

Arguments:

Argument	Data type	Description
pvInterface	void*	Pointer reference to the interface object
pfnRecvDat	PFN_NETXCON_DEVICE_RECEIVE_ CALLBACK	Function pointer of the receive callback (see section 3.2.9)
pvUser	void*	User pointer passed to receive callback function

Return Values:

Return Values	
NXCON_NO_ERROR	Interface started successfully
NXCON_DRV_NOT_INITIALIZED	The connector is not initialized
NXCON_DRV_INVALID_POINTER	Invalid pointer reference (pvInterface)
NXCON_DRV_NOT_START	Failed to start interface

Example:

```
int32_t netXConStartInterface( void* pvInterface,
                           PFN_NETXCON_DEVICE_RECEIVE_CALLBACK pfnRecvDat, void* pvUser)
{
MY_IFC_INFO_T* ptMyInterfaceInfo = (MY_IFC_INFO_T*)pvInterface;

  /* TODO: create all required resources (e.g. start send/receive handler) */
}
```


© Hilscher, 2010-2011

3.2.6 netXConStopInterface

Disconnect the interface connection. Stop all services, started under *netXConStartInterface()*. The interface the call belongs to, can be identified via *pvInterface* the interface private information (see section *netXConCreateInterface* on page 24).

Function call:

```
int32_t netXConStopInterface(void* pvInterface)
```

Arguments:

Argument	Data type	Description
pvInterface	Void*	Pointer reference to the interface object

Return Values:

Return Values	
NXCON_NO_ERROR	Interface started successfully
NXCON_DRV_NOT_INITIALIZED	The connector is not initialized
NXCON_DRV_INVALID_POINTER	Invalid pointer reference (pvInterface)

© Hilscher, 2010-2011

3.2.7 netXConSendInterface

Send binary data to the interface. The interface the call belongs to, can be identiefied via *pvInterface*, the interface private information (see netXConCreateInterface on page 24).

Function call:

```
int32_t netXConSendInterface(   void* pvInterface,
                            unsigned char* pabData,
                            unsigned long ulDataLen)
```

Arguments:

Argument	Data type	Description
pvInterface	bVoid*	Pointer reference to the interface object
pabData	unsigned char*	Pointer reference to the data to send
ulDataLen	unsigned long	Length of the data to send

Return Values:

Return Values	
NXCON_NO_ERROR	Interface started successfully
NXCON_DRV_NOT_INITIALIZED	The connector is not initialized
NXCON_DRV_SEND_ERROR	Failed to send data
NXCON_DRV_NOT_START	Selected interface was not started
NXCON_DRV_INVALID_POINTER	Invalid pointer reference (pvInterface)

© Hilscher, 2010-2011

3.2.8 PFN_NETXCON_DEVICE_NOTIFY_CALLBACK

This callback function is used to signal the upper layers the attachment or detachment of an interface. The callback is passed by the netXConOpen() function (see section 3.2.2). *pvUser* is a administrative structure of the netXTransport upper layer and needs to be passed through. (*pvUser* is delivered in *netXConOpen()*).

Interface Attachment or detachment can be notified asynchronous during the whole life cycle.

Function call:

```
typedef void       (*PFN_NETXCON_DEVICE_NOTIFY_CALLBACK)
                   (    const char*szDeviceName,
                        NETX_INTERFACE_NOTIFICATION_E eNotify,
                        void* pvLayer,
                        void* pvUser)
```

Arguments:

Argument	Data type	Description
szDeviceName	const char*	Name of interface (e.g. TCP0)
eNotify	NETX_INTERFACE_NOTIFICATION_E	**eATTACHED** New Interface attached (e.g. plugged usb) **eDETACHED** Interface detached (e.g. unplugged usb)
pvLayer	void*	Reference of the layer
pvUser	void*	User pointer

Return Values:

Return Values
none

© Hilscher, 2010-2011

3.2.9 PFN_NETXCON_DEVICE_RECEIVE_CALLBACK

This callback function is used to pass a byte stream of incoming data to the upper layers. The callback is passed by the *netXConStartInterface()* function (see section 3.2.5). Incoming data can be notifed if *netXConStartInterface()* is called until *netXConStopInterface()* is called. *pvUser* is a administrative structure of the upper layer and needs to be passed through. (*pvUser* is delivered in *netXConStartInterface()*).

Function call:

```
typedef void    (*PFN_NETXCON_DEVICE_RECEIVE_CALLBACK)
                (    unsigned char* pabRxBuffer,
                     unsigned long ulReadLen,
                     void* pvUser)
```

Arguments:

Argument	Data type	Description
pabRxBuffer	unsigned char*	Reference of received data
ulReadLen	unsigned long	Length of received data
pvUser	void*	User pointer

Return Values:

Return Values
none

© Hilscher, 2010-2011

3.2.10 netXConGetInformation

Query connector specific information.

Function call:

```
long netXConGetInformation (    NETX_CONNECTOR_INFO_E eCmd,
                                unsigned long ulSize,
                                void* pvConnectorInfo)
```

Arguments:

Argument	Data type	Description
eCmd	NETX_CONNECTOR_INFO_E	**eSTATE** Query state of the connector **eUUID** Query UUID of the connector **eTYPE** Query type of the connector **eIDENTIFIER** Query identifier of the connector **eSHORT_INTERFACE_NAME** Translates long interface name to related short interface name **eLONG_INTERFACE_NAME** Translates short interface name to related long interface name **eDESCRIPTION** Query connector description
ulSize	unsigned long	Size of buffer referenced by pvConnectorInfo
pvConnectorInfo	**eSTATE** unsigned long* **eUUID** struct UUID* **eTYPE** unsigned long* **eIDENTIFIER** char* **eSHORT_INTERFACE_NAME** INTERFACE_NAME_TRANSLATION_T* **eLONG_INTERFACE_NAME** INTERFACE_NAME_TRANSLATION_T* **eDESCRIPTION** char*	Pointer reference to connector information

© Hilscher, 2010-2011

Parameter options:

eSTATE

Bit	Description
0	Connector enabled
1 … 31	Reserved, do not use (set to zero)

eTYPE

Bit	Description
0	DPM
1	USB
2	Serial
3	Ethernet
4 … 31	Reserved for future connection types

eSHORT_INTERFACE_NAME / eLONG_INTERFACE_NAME

struct INTERFACE_NAME_TRANSLATION_T		
Element	**Data type**	**Description**
szSource	const char*	Pointer reference to string containing short interface name (eLONG_INTERFACE_NAME) or long interface name (eSHORT_INTERFACE_NAME)
szDestination	char*	Pointer reference to string buffer cointaining translated (short or long) interface
ulDestinationSize	unsigned long	Size of string buffer referenced by szDestination

Return Values:

Return Values	
NXCON_NO_ERROR	Information retrieved successfully
NXCON_DRV_INVALID_POINTER	Invalid pointer reference (pvConnectorInfo)
NXCON_DRV_BUFFER_TOO_SHORT	The supplied buffer referenced by pvConnectorInfo is too short
NXCON_DRV_INVALID_PARAMETER	The supplied command identifier is invalid

© Hilscher, 2010-2011

3.2.11 netXConGetConfig

Get the configuration of the connector and the connectors configuration dialog as a null-terminated string (e.g. "DEVNAME=GLOBAL,ENABLED=1; DEVNAME=COM1,BAUDRATE=115200")

Function call:

```
long netXConGetConfig (    NETX_CONNECTOR_CONFIG_CMD_E eCmd,
                           void* pvConfig)
```

Arguments:

Argument	Data type	Description
eCmd	NETX_CONNECTOR_CONFIG_CMD_E	**eCMD_CONFIG_GETLEN** Query length of connector configuration string **eCMD_CONFIG_GETSTRING** Query connector configuration string **eCMD_DIALOG_GETLEN** Query length of current dialog configuration string **eCMD_DIALOG_GETSTRING** Query current dialog configuration string
pvConfig	**eCMD_CONFIG_GETLEN** unsigned long* **eCMD_CONFIG_GETSTRING** char* **eCMD_DIALOG_GETLEN** unsigned long* **eCMD_DIALOG_GETSTRING** char*	Pointer reference to configuration data

Return Values:

Return Values	
NXCON_NO_ERROR	Configuration data retrieved successfully
NXCON_DIALOG_NOT_INITIALIZED	The dialog of the connector is not initialized
NXCON_DRV_INVALID_POINTER	Invalid pointer reference (pvConfig)
NXCON_DRV_INVALID_PARAMETER	The supplied command identifier is invalid

© Hilscher, 2010-2011

3.2.12 netXConSetConfig

Set the configuration of the connector and the connectors configuration dialog via a null-terminated string.

Function call:

```
long netXConSetConfig (    NETX_CONNECTOR_CONFIG_CMD_E eCmd,
                           const char* szConfig)
```

Arguments:

Argument	Data type	Description
eCmd	NETX_CONNECTOR_CONFIG_CMD_E	**eCMD_CONFIG_SETSTRING** Set connector configuration **eCMD_DIALOG_SETSTRING** Set current dialog configuration
pvConfig	const char*	Pointer reference to null-terminated connector configuration string

Return Values:

Return Values	
NXCON_NO_ERROR	Configuration data retrieved successfully
NXCON_DIALOG_NOT_INITIALIZED	The dialog of the connector is not initialized
NXCON_DRV_INVALID_POINTER	Invalid pointer reference (szConfig)
NXCON_DRV_INVALID_PARAMETER	The supplied command identifier is invalid

© Hilscher, 2010-2011

3.2.13 netXConGetInformationInterface

Query connector specific information.

Function call:

```
long netXConGetInformationInterface( PCONNECTOR_INTERFACE pvInterface,
                                     NETX_INTERFACE_INFO_E eCmd,
                                     unsigned long ulSize,
                                     void* pvInterfaceInfo)
```

Arguments:

Argument	Data type	Description
pvInterface	PCONNECTOR_INTERFACE	Pointer reference to the interface object
eCmd	NETX_INTERFACE_INFO_E	**eINTERFACE_STATE** Query interface state **eSEND_TIMEOUT** Query send timeout **eRESET_TIMEOUT** query reset timeout **eKEEPALIVE_TIMEOUT** Query keep-alive timeout
ulSize	unsigned long	Size of interface information data
pvInterfaceInfo	**eINTERFACE_STATE** unsigned long* **eSEND_TIMEOUT** unsigned long* **eRESET_TIMEOUT** unsigned long* **eKEEPALIVE_TIMEOUT** unsigned long*	Pointer reference to interface information data

Parameter options:

Interface state	Value	Description
eINTERFACE_STATE_NOT_SET	0	Interface invalid, no information are valid
eINTERFACE_STATE_NOT_INIT	1	Interface object created, but not initialized
eINTERFACE_STATE_INITIALIZED	2	Interface is initialized
eINTERFACE_STATE_AVAILABLE	3	Interface is available, but not running
eINTERFACE_STATE_NOT_AVAIL	4	Interface not mor available
eINTERFACE_STATE_RUNNING	5	Interface is ready to handle communication
eINTERFACE_STATE_STOPPED	6	Interface was stopped

© Hilscher, 2010-2011

Return Values:

Return Values	
NXCON_NO_ERROR	Information retrieved successfully
NXCON_DRV_INVALID_POINTER	Invalid pointer reference (pvInterfaceInfo)
NXCON_DRV_BUFFER_TOO_SHORT	The supplied buffer referenced by pvInterfaceInfo is too short
NXCON_DRV_INVALID_PARAMETER	The supplied command identifier is invalid
NXCON_DRV_NOT_INITIALIZED	The connector is not initialized

3.2.14 netXConCreateDialog

Create dialog for graphical connector configuration.

Function call:

```
HWND netXConCreateDialog (HWND hParentWnd,
                          const char* szConfig)
```

Arguments:

Argument	Data type	Description
hParentWnd	HWND	Handle to the window that owns the dialog
szConfig	const char*	Configuration string for inital dialog settings

Return Values:

Return Values
If the creation of the dialog succeeds, the return value is the window handle to the dialog. If the creation of the dialog fails, the return value is NULL.

© Hilscher, 2010-2011

3.2.15 netXConEndDialog

Close dialog for graphical connector configuration.

Function call:

```
void netXConEndDialog ( void)
```

Arguments:

Argument	Data type	Description
none		

Return Values:

Return Values
none

© Hilscher, 2010-2011

3.3 Handling of Interface Notifications

The connector module notifies the transport layer about the arrival or removal of an interface via a callback function. This callback is processed synchronously or asynchronously, depending on runtime state of the transport layer. The figure below illustrates the handling of notification messages from the connector.

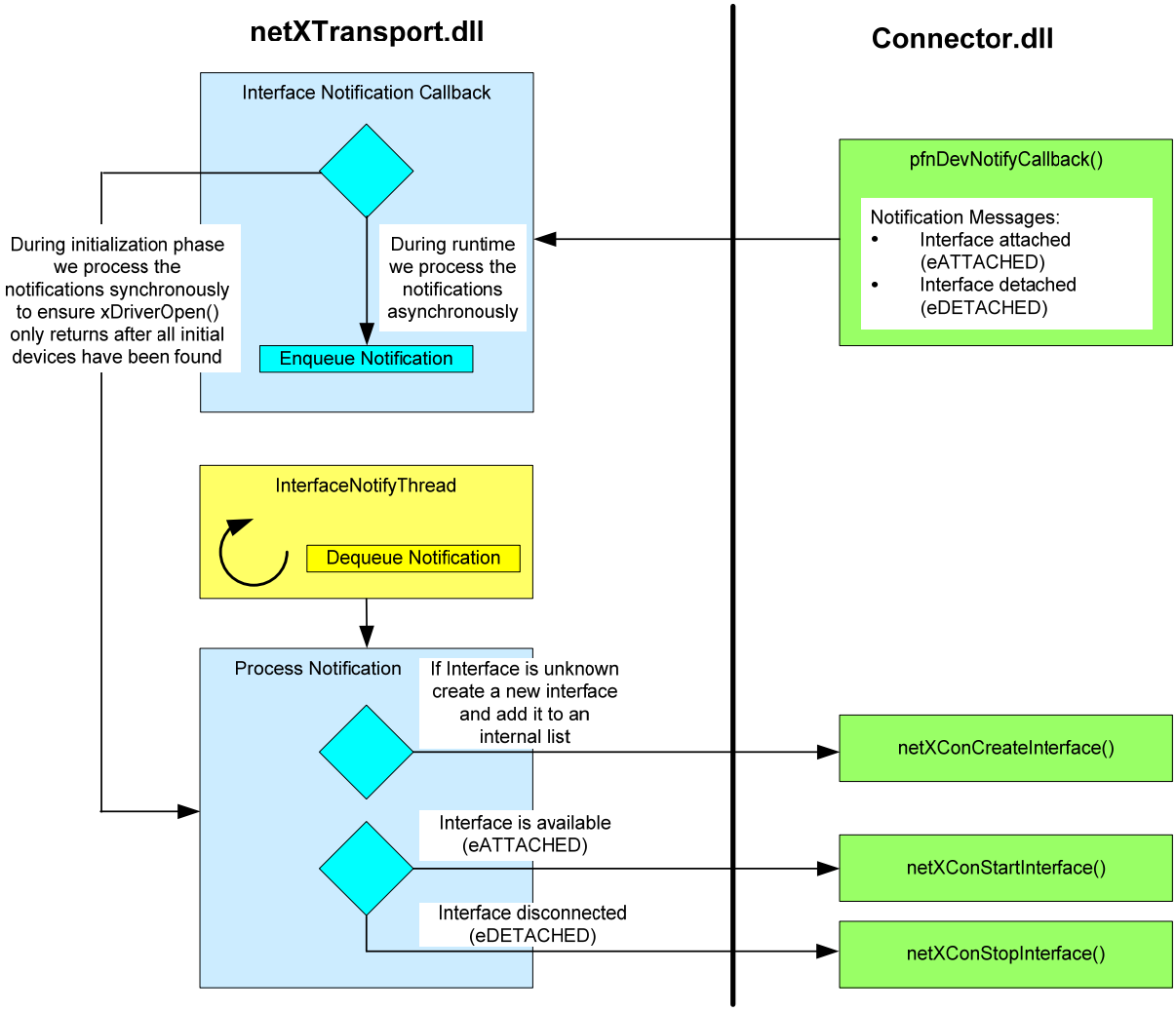

Figure 11: Handling of Interface Notifications

© Hilscher, 2010-2011

3.4 Connector Configuration

The Connector API provides two methods to configure a connector:

- Configuration via a configuration string
- Configuration via a graphical user interface

The host implementation of the 'netX Diagnostic and Remote Access' includes two sample connectors. A USB/RS232 connector to connect a device via a physical or virtual serial port and a TCP connector to connect a device via the TCP/IP protocol stack. The configuration is done with an configuration class (CConnectorConfig) which is shipped in source code with the host-side marshaller implementation. This class can be used to provide a configuration interface for a custom connector. A universally unique identifier (UUID) must be assigned to each connector. The identifiers of the example connectors a listed in the table below.

Connector	Universally Unique Identifier (UUID)
USB/RS232 Connector (RS232Connector.dll)	`4DF3EA2F-C989-46BD-81BF-A02EC6F8217D`
TCP Connector (TCPConnector.dll)	`1719D5A0-DD3B-48E2-BBF6-DDDAAED7E8B7`

The configuration class stores the settings of a connector in the windows registry.

HKEY_LOCAL_MACHINE/Software/Hilscher/netXTransport/<UUID of the connector>

For each configured interface a subkey is created under the main key of a connector. A typical body of a registry tree for a connector configuration a is shown in the figure below.

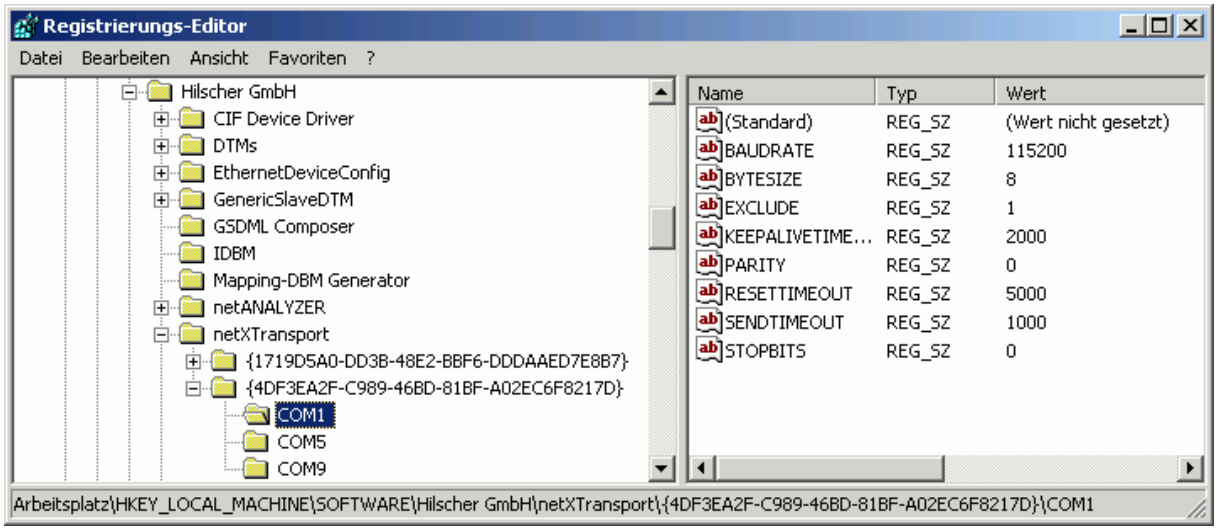

Figure 12: Registry Tree of a Configuration for the USB/RS232 Connector

© Hilscher, 2010-2011

3.4.1 Configuration via String Descriptor

To configure the connector via a configuration string the API functions netXConGetConfig() and netXConSetConfig() must be used. The reading and storing of the configuration is done by using a special formatted string descriptor. The advantage of a configuration string descriptor is the easy extension of new information and parameters and the possibility of a human readable configuration description.

Note: A call to the netXConSetConfig() routine with a properly formatted string descriptor will replace the old configuration of the connector.

The layout of a descriptor follows some simple rules:

- A descriptor can contain multiple datasets, being separated by a semicolon (;)
- Each dataset can consist of multiple value pairs separated by a comma (,)
- A value pair, consisting of a key and a value, is separated by an equal sign (=)

Type	Example	Separator
Dataset	<dataset>**;**<dataset>**; ... ;**<dataset>	**;** (semicolon)
Value pair	<value pair 1>**,**< value pair 2>**, ... ,**< value pair n>	**,** (comma)
Key / Value	key**=**value	**=** (equal sign)

Every configuration descriptor string consists of a single dataset keeping the global layer configuration and an arbitrary count of datasets keeping interface specific configurations.

A dataset with the global layer configuration must include the value pair as stated below:

DEVNAME=GLOBAL

Every dataset with the an interface configuration must include the name of the interface:

DEVNAME=<Name of interface configuration>

© Hilscher, 2010-2011

3.4.1.1 Common Configuration Keys

There is a set of configuration keys which needs to be supported by every connector. The description of the configuration keys which are mandatory for every connector is given below:

- Keep Alive Timeout: After a configured time of inactivity, the condition of the connection is determined by transmitting heart beat packets via the communication line (see reference [1]).
- Reset Timeout: Processing a device reset could cause a termination of the interface connection. As the reestablishment of the connection could be very time consuming, the reset timeout setting must be adjusted to the type of physical connection.
- Send Timeout: The data transmission may fail due to a broken connection or a full transmission buffer. The transmission is cancelled if the send process cannot be completed in this time period.
- Exclude interface: Configured interfaces which are excluded are ignored during the process of connection establishment.
- Enable/Disable connector: If the connector is disabled, it is not loaded at start up of the netX Marshaller.

Description	Range	Default	String Descriptor Key	Scope
Keep Alive Timeout	100 -60000 (ms)	2000 ms	KEEPALIVETIMEOUT	Interface
Reset Timeout	100 -60000 (ms)	TCP: 20000 ms RS232: 5000ms	RESETTIMEOUT	Interface
Send Timeout	100 -60000 (ms)	1000 ms	SENDTIMEOUT	Interface
Exclude configured interface	0 = Not excluded 1 = Excluded	Not excluded	EXCLUDE	Interface
Enable/Disable connector	0 = Disabled 1 = Enabled	Enabled	ENABLED	Layer
Dataset identifier to mark global layer or interface specific configuration	Global layer config: GLOBAL Interface config: Name (e.g. COM1)	-	DEVNAME	Interface/ Layer

© Hilscher, 2010-2011

3.4.1.2 Configuration Keys of the TCP Connector

To establish a connection to a device via TCP, the IP address and TCP port have to be specified via the TCP connector specific configuration keys. In place of a single IP address, it is also possible to define a whole address range. The scan timeout limits the time slot for a successful connection establishment.

Note: Do not use large address ranges in combination with a low scan timeout. Microsoft introduced in Windows XP SP2 a limit of concurrent half-open outbound TCP connections (connection attempts) to slow the spread of virus and malware from system to system. This limit makes it impossible to have more than 10 concurrent half-open outbound connections. Every further connection attempt is put in a queue and forced to wait. Due to this limitation a large address range used in combination with a low scan timeout could prevent the connection establishment to a device.

The TCP interface is configured by the configuration keys listed below.

Description	Range	Default	String Descriptor Key	Scope
Interface Name	IPRANGE<0..n> (e.g. IPRANGE2)	-	DEVNAME	Interface
IP Address (Begin of range)	e.g. 192.168.1.2	192.168.1.1	IPADDR	Interface
IP Address (End of range)	e.g. 192.168.1.5 0.0.0.0 = No Range	No Range	IPADDREND	Interface
TCP Port	0 - 65535	50111	PORT	Interface
Scan Timeout	10 -10000 (ms)	100 ms	CONNECTTIMEOUT	Layer

Example configuration descriptor for the TCP Connector:

```
DEVNAME=GLOBAL,CONNECTTIMEOUT=100,ENABLED=1;
DEVNAME=IPRANGE0,EXCLUDE=0,IPADDR=192.168.10.107,IPADDREND=0.0.0.0,
KEEPALIVETIMEOUT=2000,PORT=50111,RESETTIMEOUT=2000,SENDTIMEOUT=1000;
```


© Hilscher, 2010-2011

3.4.1.3 Configuration Keys of the USB/RS232 Connector

To establish a connection through a serial port interface, conventions of baud rate and character framing (byte size, parity and stop bits) must be agreed to by device and host side. On the host side the serial interface is configured by the configuration keys listed below.

Description	Range	Default	String Descriptor Key	Scope
Interface Name	COM<1..n> (e.g. COM1)	-	DEVNAME	Interface
Baud Rate	9600 = 9,6 kbps 19200 = 19,2 kbps 38400 = 38,4 kbps 57600 = 57,6 kbps 115200 = 115,2 kbps	115,2 kbps	BAUDRATE	Interface
Byte Size	7 = 7 Bytes 8 = 8 Bytes	8 Bytes	BYTESIZE	Interface
Parity	0 = No Parity 1 = Odd Parity 2 = Even Parity 3 = Mark Parity 4 = Space Parity	No Parity	PARITY	Interface
Stop Bits	0 = 1 Stop bit 1 = 1,5 Stop bits 2 = 2 Stop bits	1 Stop bit	STOPBITS	Interface

Example configuration descriptor for the USB/RS232 connector:

```
DEVNAME=GLOBAL,ENABLED=1;
DEVNAME=COM1,BAUDRATE=115200,BYTESIZE=8,EXCLUDE=0,KEEPALIVETIMEOUT=2000,
PARITY=0,RESETTIMEOUT=2000,SENDTIMEOUT=1000,STOPBITS=0;
```


© Hilscher, 2010-2011

3.4.2 Configuration via Graphical User Interface

To ease the configuration of a device connection, a connector can provide a graphical user interface. This interface must be implemented inside the connector library as a child dialog. The Connector API provides the functions netXConCreateDialog() and netXConDeleteDialog() to access the graphical user interface of a connector. To display the dialog in the main application the netXConCreateDialog() function places its dialog in the parent window, handed over by the caller.

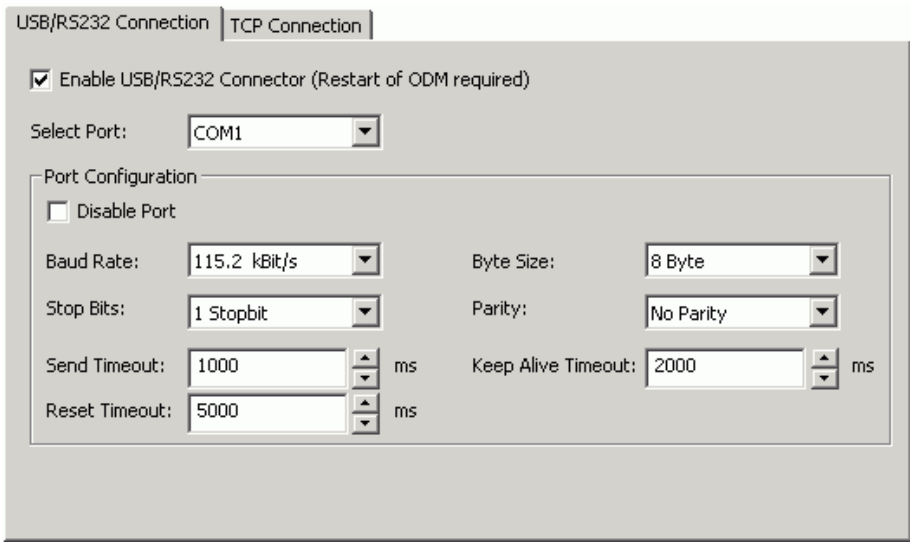

Figure 13: Configuration Dialog for USB/RS232 Connections

© Hilscher, 2010-2011

4 Appendix

4.1 List of Tables

4.2 List of Figures

© Hilscher, 2010-2011

4.3 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

© Hilscher, 2010-2011