



Program Reference Guide
netX Diagnostic and Remote Access
Target Device
V2.0.x.x

Hilscher Gesellschaft für Systemautomation mbH

www.hilscher.com

DOC090701PR03EN | Revision 3 | English | 2010-05 | Released | Public

Table of Contents

1	Introduction.....	4
1.1	About this Document.....	4
1.2	List of Revisions.....	4
1.3	Terms, Abbreviations and Definitions.....	5
1.4	References.....	5
1.5	System Requirements.....	6
1.6	Features.....	6
1.7	Limitations.....	6
1.8	Intended Audience.....	6
1.9	Legal Notes.....	7
1.9.1	Copyright.....	7
1.9.2	Important Notes.....	7
1.9.3	Exclusion of Liability.....	8
1.9.4	Export.....	8
2	netX Marshaller Overview.....	9
2.1	Marshaller Main.....	11
2.2	Connector.....	11
2.3	Transport.....	12
2.3.1	rcX Packet Transport.....	12
2.3.2	cifX API Transport.....	12
3	Marshaller Main Module Internals.....	13
3.1	Public Structure Definitions.....	13
3.1.1	Connector Configuration (HIL_MARSHALLER_CONNECTOR_PARAMS_T).....	13
3.1.2	Transport Layer Configuration (TRANSPORT_LAYER_CONFIG_T).....	13
3.1.3	Marshaller Configuration (HIL_MARSHALLER_PARAMS_T).....	14
3.2	Internal Structure Definitions.....	15
3.2.1	Connector Registration (HIL_MARSHALLER_CONNECTOR_T).....	15
3.2.2	Transport Registration (TRANSPORT_LAYER_DATA_T).....	15
3.2.3	Buffer (HIL_MARSHALLER_BUFFER_T).....	16
3.3	Functions.....	17
3.3.1	HilMarshallerStart.....	18
3.3.2	HilMarshallerStop.....	19
3.3.3	HilMarshallerTimer.....	19
3.3.4	HilMarshallerMain.....	19
3.3.5	HilMarshallerRegisterConnector.....	20
3.3.6	HilMarshallerUnregisterConnector.....	20
3.3.7	HilMarshallerConnRxData.....	21
3.3.8	HilMarshallerConnTxData.....	22
3.3.9	HilMarshallerConnTxComplete.....	23
3.3.10	HilMarshallerRegisterTransport.....	24
3.3.11	HilMarshallerUnregisterTransport.....	24
3.3.12	HilMarshallerGetBuffer.....	25
3.3.13	HilMarshallerFreeBuffer.....	25
3.4	Sequence Diagrams.....	26
3.4.1	Marshaller Data Processing.....	26
3.4.2	Unsolicited Packets / Requests.....	27
4	TCP Connector Internals.....	28
4.1	Overview.....	28
4.2	Structure Definitions.....	28
4.2.1	Run-time Information (TCP_CONNECTOR_MGT_INFO_T).....	29
4.2.2	Extended Task Information (TCP_CONNECTOR_TASK_INFO_T).....	29
4.2.3	Environment Information (TCP_CONNECTOR_ENV_INFO_T).....	30
4.2.4	Network Information (TCP_CONNECTOR_NET_INFO_T).....	30
4.2.5	Protocol Information (TCP_CONNECTOR_PROTOCOL_INFO_T).....	31
4.2.6	Latest Error Entry (RCX_ERROR_BUFFER_ENTRY_T).....	31
4.3	TCP Connector State Machine.....	32
5	Porting the Marshaller.....	35
5.1	Directory Structure.....	35
5.2	Implementing OS Dependencies.....	36

5.3	Implementing a Connector	37
5.3.1	Connector Initialization	37
5.3.2	Registration Data.....	38
5.3.3	Data Reception.....	38
5.3.4	Data Transmission.....	39
5.4	Implementing a Custom Transport.....	40
5.4.1	Initialization Function	40
5.4.2	Registration	41
5.4.3	Data Handling.....	41
6	rcX Marshaller Sample	42
6.1	Overview	42
6.2	Adding to an existing Firmware.....	43
6.2.1	Marshaller Task.....	44
6.2.2	Configuration Data.....	45
7	Appendix	48
7.1	Marshaller Configuration Entries in the rcX Configuration File	48
7.1.1	UART Connector	48
7.1.2	UART Connector (Emulation via USB).....	49
7.1.3	TCP Connector.....	50
7.1.4	Marshaller Task.....	51
7.2	Marshaller Error Codes	54
8	Appendix	55
8.1	List of Tables	55
8.2	List of Figures.....	55
8.3	Contacts	56

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 a way so the customer can integrate their own interface by attaching themselves to the communication protocol (*Hilscher Transport*).

This program reference guide describes the internal structure of the so called '*netX Marshaller*' package, which offers the *netX Diagnostic and Remote Access* services. It focuses the technical implementation and integration into an rcX or a custom target system.

1.2 List of Revisions

Rev	Date	Name	Chapter	Revision
1	2009-06-26	RM MT		Created
2	2009-09-18	MSt		Added subchapters describing TCP connector
3	2010-05-12	HH		Layout changes

Table 1: List of Revisions

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

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.

1.4 References

This document based on the following specifications and manuals:

- [1] netX Remote Access Services - Fundamentals, Rev. 1, Hilscher GmbH, April 2009
- [2] netX Dual-Port Memory Interface for netX based Products, Rev. 7, Hilscher GmbH, December 2008
- [3] cifX Device Driver Manual, Rev. 12, Hilscher GmbH, November 2008 cifX Device Driver Manual, Rev. 12, Hilscher GmbH, November 2008
- [4] rcX – Configuration, Rev. 6, Hilscher GmbH, May 2007
- [5] rcX – Driver Functions Reference, Rev. 6, Hilscher GmbH, July 2007
- [6] rcX – Kernel API Functions Reference, Rev. 4, Hilscher GmbH, January 2008

Table 3: References

1.5 System Requirements

Depending on the target platform, the system requirements for using the '*netX Marshaller*' may be different. Hilscher provides a sample for the rcX operating system and a basic development package which can be ported to a custom hardware.

rcX Package:

- netX 50 / 100 / 500
- rcX operating system V2.0.4.6 or higher
- Optional: Shared Memory API V0.927 or later

Development Package:

- 32Bit, little endian host system
- Physical access to a cifX DPM using the cifX Driver functions or possibility to transfer rcX packets to an attached netX target

1.6 Features

- Operating System independent
- ANSI C-Source

1.7 Limitations

The '*netX Marshaller*' will not synchronize access to the DPM, if multiple connections (local and remote) are open. It is part of the user's implementation to make sure that the mailbox data is correctly distributed to the applications (Packet Routing), by hooking into the cifX API's packet functions.

1.8 Intended Audience

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

1.9 Legal Notes

1.9.1 Copyright

© 2008-2009 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.

1.9.2 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.

1.9.3 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.

1.9.4 Export

The delivered product (including the technical data) is subject to export or import laws as well as the associated regulations of different countries, 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 netX Marshaller Overview

The '*netX Marshaller*' consists of three components.

Component	Description
Main module (HilMarshaller)	Manages all physical connections, reassembles incoming data packets and routes them to the appropriate transport layer.
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
Transport	The transport layer executes the commands from an incoming data packet and returns the response to the Marshaller, once the function has been executed. Note: Developers can create custom transports for individual purposes. The custom transports won't be available for Hilscher standard diagnostics, but can be used as a customers channel to its hardware.

Table 4: Marshaller Components

The Marshaller covers the following access functions:

- Diagnostics
- I/O data monitoring
- System management (channel init, reset device, etc.)
- Firmware and configuration download.
- **Note:** For remote access to a RAM based system (cifX), the firmware and configuration won't be stored on the remote PC and will be lost after a reboot of this PC.
- Target access via rcX Packets or cifX API calls
- Possible user interaction to allow synchronization of local and remote hardware access

Encoding and decoding of the remote access data packets is transparently done on the host and the target system, inside the Marshaller 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*'.

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

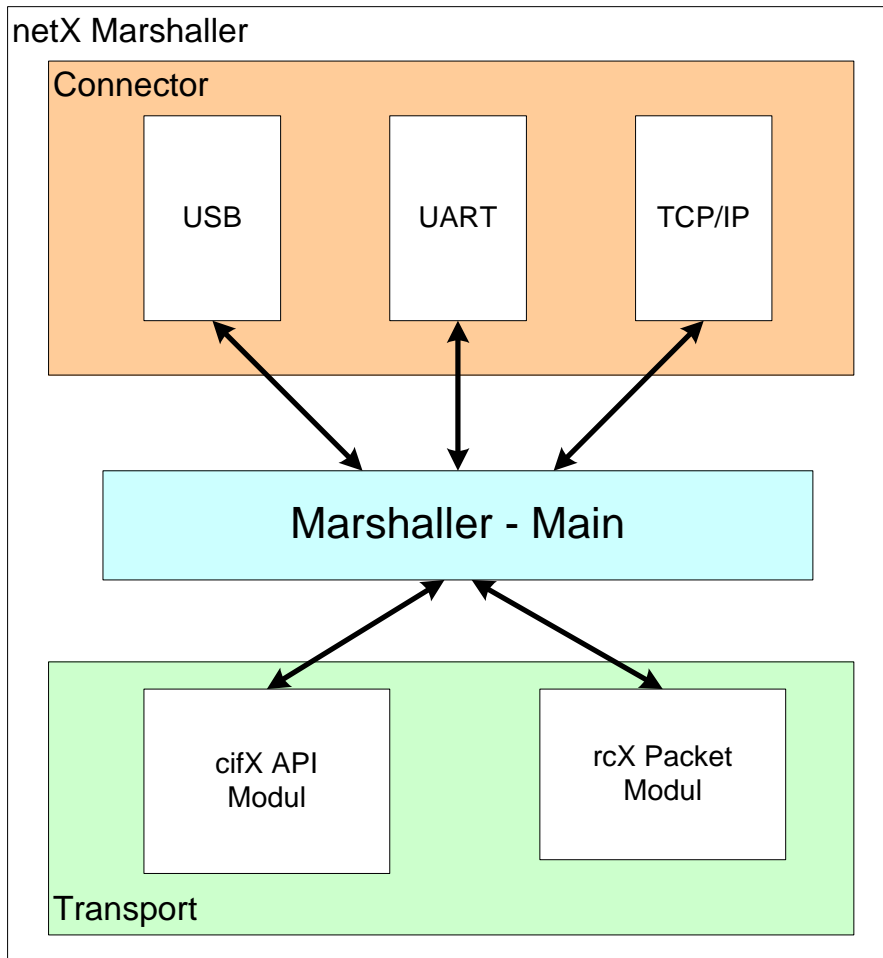


Figure 1: netX Marshaller Structure

2.1 Marshaller Main

The Marshaller main module contains the data management and routing components. It performs the following tasks:

- Start-up Initialization and configuration
- Timeout monitoring / keep-alive
- Per connection state machine handling
- Data packet reassembly and distribution to transport modules

All resources are allocated using dynamic memory management functions, but will only be used during startup. Once the Marshaller has been started no further memory allocation will be done.

2.2 Connector

The task of a connector is to provide a physical data transfers to the Marshaller main component. It just needs to provide a raw binary stream input and output handling, as the data evaluation is done inside the Marshaller main component (Data Scanner)

2.3 Transport

Transport modules handle the incoming data telegrams, evaluating their data contents. Each data telegram must be answered and processed.

Per default two transports are available. It is sufficient if a target only supports one of the both transport mode:

- rcX Packet Access: Used to route rcX packets into a system. This module is only available for rcX itself and must be reprogrammed when using another host system.
- cifX API Access: Used to communicate with the firmware components via a DPM (either 'Virtual DPM / Shared Memory' (SHM/VDPM) or real DPM)

2.3.1 rcX Packet Transport

The rcX packet transport layer is included as source running under rcX. It is a router component and offers the following functions.

- Sending requests to the rcX based system and reassign the answer packet to the request (reconstructing Hilscher *Transport Header*)
- Indication packet routing, so unsolicited rcX packets are sent to the application that requested them

2.3.2 cifX API Transport

The cifX API transport layer receives data packets from the Marshaller component. The packets are containing a description of the cifX function which should be called. The transport layer executes the function by using a function pointer table, initialized during the Marshaller start-up.

An application can hook every cifX function call, by overwriting the pointer table. This mechanism allows an application to take over control of the called cifX functions.

The cifX API transport module is operating system independent.

It can be used to:

- Access a real DPM using a cifX driver
- Access a local Virtual DPM (Shared memory API, SHM) running under rcX

Note: The cifX transport layer does not support sharing of the DPM by itself. If a local application is also accessing the DPM/VDPM, the user needs to hook into the functions to distribute packets to the correct caller.

3 Marshaller Main Module Internals

3.1 Public Structure Definitions

The following structures are needed for an application to startup and configure a Marshaller instance.

3.1.1 Connector Configuration (HIL_MARSHALLER_CONNECTOR_PARAMS_T)

Element	Type	Description
pfnConnectorInit	PFN_CONN_INIT	Initialization function of the connector. Will be called for every connector during Marshaller startup
ulDataBufferSize	UINT32	Size of the data buffers for this connector
ulDataBufferCnt	UINT32	Number of data buffers to allocate for this connector
usFlags	UINT16	
ulTimeout	UINT32	Timeout for monitoring receive data timeouts
pvConfigData	void*	Connector specific configuration data (e.g. pointer to UART_CONN_CONFIG_T structure for the rcX UART connector)

Table 5: Connector Configuration Structure

3.1.2 Transport Layer Configuration (TRANSPORT_LAYER_CONFIG_T)

Element	Type	Description
pfnInit	PFN_TRANSPORT_INIT	Initialization function of the transport layer. Will be called for every transport layer during Marshaller startup
pvConfig	void*	Layer specific configuration data (e.g. pointer to CIFX_TRANSPORT_CONFIG structure for the cifX API transport)

Table 6: Transport Layer Configuration Structure

3.1.3 Marshaller Configuration (HIL_MARSHALLER_PARAMS_T)

Element	Type	Description
szServerName	char[32]	Human readable name of server running this Marshaller instance
ulMaxConnectors	UINT32	Maximum number of connectors the Marshaller can handle
ulConnectorCnt	UINT32	Number of connectors to initialize at startup
ptConnectors	HIL_MARSHALLER_CONNECTOR_PARAMS_T[]	Array of connectors to initialize. Array size must be <i>ulConnectorCnt</i> .
ulTransportCnt	UINT32	Number of transport layers to initialize at startup
atTransports	TRANSPORT_LAYER_CONFIG_T[]	Array of transports to initialize. Array size must be <i>ulTransportCnt</i> .

Table 7: Marshaller Configuration Structure

3.2 Internal Structure Definitions

The following structures are needed for programmers of connectors and transport layers.

3.2.1 Connector Registration (HIL_MARSHALLER_CONNECTOR_T)

Element	Type	Description
pfnTransmit	PFN_CONN_TRANSMIT	Called by the Marshaller, if someone wants to send data on this connector
pfnDeinit	PFN_CONN_DEINIT	Shutdown function
pfnPoll	PFN_CONN_POLL	Optional function to call cyclically (set to NULL if not needed)
ulDataBufferSize	UINT32	Data buffer size in Bytes
ulDataBufferCnt	UINT32	Number of data buffers
ulTimeout	UINT32	Timeout for reception in ms
pvUser	void*	Parameter to pass on callbacks

Table 8: Connector Registration Structure

3.2.2 Transport Registration (TRANSPORT_LAYER_DATA_T)

Element	Type	Description
usDataType	UINT16	Data type (see Hilscher Transport Header) to register this transport for
pfnHandler	PFN_TRANSPORT_HANDLER	Handler for received data frames
pfnDeinit	PFN_TRANSPORT_DEINIT	Shutdown function
pfnPoll	PFN_TRANSPORT_POLL	Optional function to call cyclically (set to NULL if not needed)
pvUser	void*	Parameter to pass on callbacks

Table 9: Transport Registration Structure

3.2.3 Buffer (HIL_MARSHALLER_BUFFER_T)

Element	Type	Description
tList	STAILQ_ENTRY	Doubly linked list element for enqueueing buffer
tMgmt		
ulConnectorIdx	UINT32	Connector number the buffer is assigned to
pvMarshaller	void*	Marshaller instance handle
eType	MARSHALLER_BUFFER_TYPE_E	Type of the buffer
ulDataBufferLen	UINT32	Total length of buffer (<i>abData</i>)
ulUsedDataBufferLen	UINT32	Number of bytes used in <i>abData</i>
ulActualSendOffset	UINT32	Actual send offset. Can be used if the connector needs to split up transfers into chunks
tTransport	HIL_TRANSPORT_HEADER	Transport header
abData	UINT8[1]	Raw data

Table 10: Marshaller Data Frame Buffer Structure

3.3 Functions

Application:

Function	Description
HilMarshallerStart	Starts up the Marshaller component and initializes all transports and connectors
HilMarshallerStop	Stops all transports, connectors and shuts down the Marshaller
HilMarshallerTimer	Cyclic timer function, which needs to be called by the application so the Marshaller can monitor timeouts
HilMarshallerMain	Marshaller main functions processing the received frame. This function must be called when the Marshaller indicates a new data frame has arrived (<i>pfnRequest()</i> is called)

Connector:

Function	Description
HilMarshallerRegisterConnector	Called by a connector to register itself at the main module
HilMarshallerUnregisterConnector	Unregister a connector (usually called on shutdown of a connector / connection)
HilMarshallerConnRxData	Called with incoming data by connector
HilMarshallerConnTxData	Function called to send a data frame to the bus, usually done by the transport layers
HilMarshallerConnTxComplete	When sending of a frame has been finished this function must be called by a connector

Transport:

Function	Description
HilMarshallerRegisterTransport	Called by a transport layer to register itself at the main module
HilMarshallerUnregisterTransport	Unregister a transport layer (usually called on shutdown)
HilMarshallerGetBuffer	Get a new buffer from the main module (e.g. for unsolicited packets)
HilMarshallerFreeBuffer	Free previously allocated buffer

3.3.1 HilMarshallerStart

Start up the Marshaller.

Function call:

```
TLR_RESULT HilMarshallerStart ( const HIL_MARSHALLER_PARAMS_T* ptParams,  
void** ppvMarshHandle,  
PFN_MARSHALLER_REQUEST pfnRequest,  
void* pvUser)
```

Arguments:

Argument	Data type	Description
ptParams	HIL_MARSHALLER_PARAMS_T*	Marshaller startup parameters
ppvMarshHandle	void**	Returned handle of the Marshaller instance (needs to be provided on every other function call)
pfnRequest	PFN_MARSHALLER_REQUEST	Function to be called by Marshaller to indicate arrival of new data
pvUser	void*	User parameter being passed on <i>pfnRequest</i> call

Return Values:

Return Values	
TLR_S_OK	Success
HIL_MARSHALLER_E_OUTOFMEMORY	Not enough memory is available to start all services

3.3.2 HilMarshallerStop

Stop a running Marshaller instance.

Function call:

```
void HilMarshallerStop( void* pvMarshHandle)
```

Arguments:

Argument	Data type	Description
pvMarshHandle	void*	Marshaller handle returned by <i>HilMarshallerStart()</i>

3.3.3 HilMarshallerTimer

Cyclic timer event function provides timeout monitoring. It needs to be called every 10ms.

Function call:

```
void HilMarshallerTimer( void* pvMarshHandle)
```

Arguments:

Argument	Data type	Description
pvMarshHandle	void*	Marshaller handle returned by <i>HilMarshallerStart()</i>

3.3.4 HilMarshallerMain

This is the main worker function. Every time the Marshaller indicates a new data arrival (by calling '*pfnRequest*') the user needs to make sure this function is executed.

Function call:

```
void HilMarshallerMain( void* pvMarshHandle)
```

Arguments:

Argument	Data type	Description
pvMarshHandle	void*	Marshaller handle returned by <i>HilMarshallerStart()</i>

3.3.5 HilMarshallerRegisterConnector

Register a connector on a Marshaller instance.

Function call:

```
TLR_RESULT HilMarshallerRegisterConnector(    void* pvMarshaller,
TLR_UINT32* pulConnectorIdx,
HIL_MARSHALLER_CONNECTOR_T* ptConn)
```

Arguments:

Argument	Data type	Description
pvMarshHandle	void*	Marshaller handle returned by <i>HilMarshallerStart()</i>
pulConnectorIdx	UINT32*	Assigned connector number, returned by Marshaller main component
ptConn	HIL_MARSHALLER_CONNECTOR_T*	Connector registration information

Return Values:

Return Values	
TLR_S_OK	Success
HIL_MARSHALLER_E_OUTOFMEMORY	Not enough memory to allocate the buffers
HIL_MARSHALLER_E_OUTOFRESOURCES	Maximum number of connectors reached

3.3.6 HilMarshallerUnregisterConnector

Unregister a connector from a Marshaller instance.

Function call:

```
void HilMarshallerUnregisterConnector(    void* pvMarshaller,
TLR_UINT32 ulConnectorIdx)
```

Arguments:

Argument	Data type	Description
pvMarshHandle	void*	Marshaller handle returned by <i>HilMarshallerStart()</i>
ulConnectorIdx	UINT32	Connector number to unregister

3.3.7 HilMarshallerConnRxData

Pass bytes from a data stream to the Marshaller main component.

Function call:

```
TLR_RESULT HilMarshallerConnRxData( void* pvMarshaller,  
TLR_UINT32 ulConnector,  
TLR_UINT8* pbData,  
TLR_UINT32 ulDataCnt);
```

Arguments:

Argument	Data type	Description
pvMarshHandle	void*	Marshaller handle returned by <i>HilMarshallerStart()</i>
ulConnectorIdx	UINT32	Connector number that is passing the data
pbData	UINT8*	Pointer to received data
ulDataCnt	UINT32	Number of bytes in buffer

Return Values:

Return Values	
TLR_S_OK	Success
HIL_MARSHALLER_E_INVALIDPARAMETER	Invalid connector index

3.3.8 HilMarshallerConnTxData

Send a transmission buffer via the given connector.

Function call:

```
TLR_RESULT HilMarshallerConnTxData( void* pvMarshaller,  
TLR_UINT32 ulConnectorIdx,  
HIL_MARSHALLER_BUFFER_T* ptBuffer);
```

Arguments:

Argument	Data type	Description
pvMarshHandle	void*	Marshaller handle returned by <i>HilMarshallerStart()</i>
ulConnectorIdx	UINT32	Connector number to send data on
ptBuffer	HIL_MARSHALLER_BUFFER_T*	Pointer to a transmission buffer

Return Values:

Return Values	
TLR_S_OK	Success
	Connector specific errors

3.3.9 HilMarshallerConnTxComplete

This function is called, if sending of a transmission buffer has been completed and it will free the buffer.

Function call:

```
TLR_RESULT HilMarshallerConnTxComplete( void* pvMarshaller,  
TLR_UINT32 ulConnector,  
HIL_MARSHALLER_BUFFER_T* ptBuffer);
```

Arguments:

Argument	Data type	Description
pvMarshHandle	void*	Marshaller handle returned by <i>HilMarshallerStart()</i>
ulConnectorIdx	UINT32	Connector number to send was complete
ptBuffer	HIL_MARSHALLER_BUFFER_T*	Pointer to the transmitted buffer

Return Values:

Return Values	
TLR_S_OK	Success

3.3.10 HilMarshallerRegisterTransport

This function is called by a transport layer during initialization to register itself at the Marshaller instance.

Function call:

```
TLR_RESULT HilMarshallerRegisterTransport(    void* pvMarshaller,  
const TRANSPORT_LAYER_DATA_T* ptLayerData)
```

Arguments:

Argument	Data type	Description
pvMarshHandle	void*	Marshaller handle returned by <i>HilMarshallerStart()</i>
ptLayerData	TRANSPORT_LAYER_DATA_T*	Registration data of transport layer

Return Values:

Return Values	
TLR_S_OK	Success
HIL_MARSHALLER_E_INVALIDPARAMETER	Invalid data type passed in registration data (already occupied)
HIL_MARSHALLER_E_OUTOFRESOURCES	Too may transport layers already registered

3.3.11 HilMarshallerUnregisterTransport

Unregister a transport layer.

Function call:

```
void HilMarshallerUnregisterTransport(    void* pvMarshaller,  
TLR_UINT16 usDataType)
```

Arguments:

Argument	Data type	Description
pvMarshHandle	void*	Marshaller handle returned by <i>HilMarshallerStart()</i>
usDataType	UINT16	Data type passed during registration

3.3.12 HilMarshallerGetBuffer

Get a pre-allocated buffer from the Marshaller. This is needed for transports that require a transfer buffer for sending unsolicited packets and internally inside the Marshaller itself.

Function call:

```
HIL_MARSHALLER_BUFFER_T* HilMarshallerGetBuffer(    void* pvMarshaller,  
MARSHALLER_BUFFER_TYPE_E eType,  
TLR_UINT32 ulConnectorIdx)
```

Arguments:

Argument	Data type	Description
pvMarshHandle	void*	Marshaller handle returned by <i>HilMarshallerStart()</i>
eType	MARSHALLER_BUFFER_TYPE_E	Requested Buffer type
ulConnectorIdx	UINT32	Connector number to request buffer for

Return Values:

Return Values	
NULL	No more buffers available
!=NULL	Valid buffer address

3.3.13 HilMarshallerFreeBuffer

Free an allocated buffer.

Function call:

```
void HilMarshallerFreeBuffer(HIL_MARSHALLER_BUFFER_T* ptBuffer)
```

Arguments:

Argument	Data type	Description
ptBuffer	HIL_MARSHALLER_BUFFER_T*	Buffer to return

3.4 Sequence Diagrams

3.4.1 Marshaller Data Processing

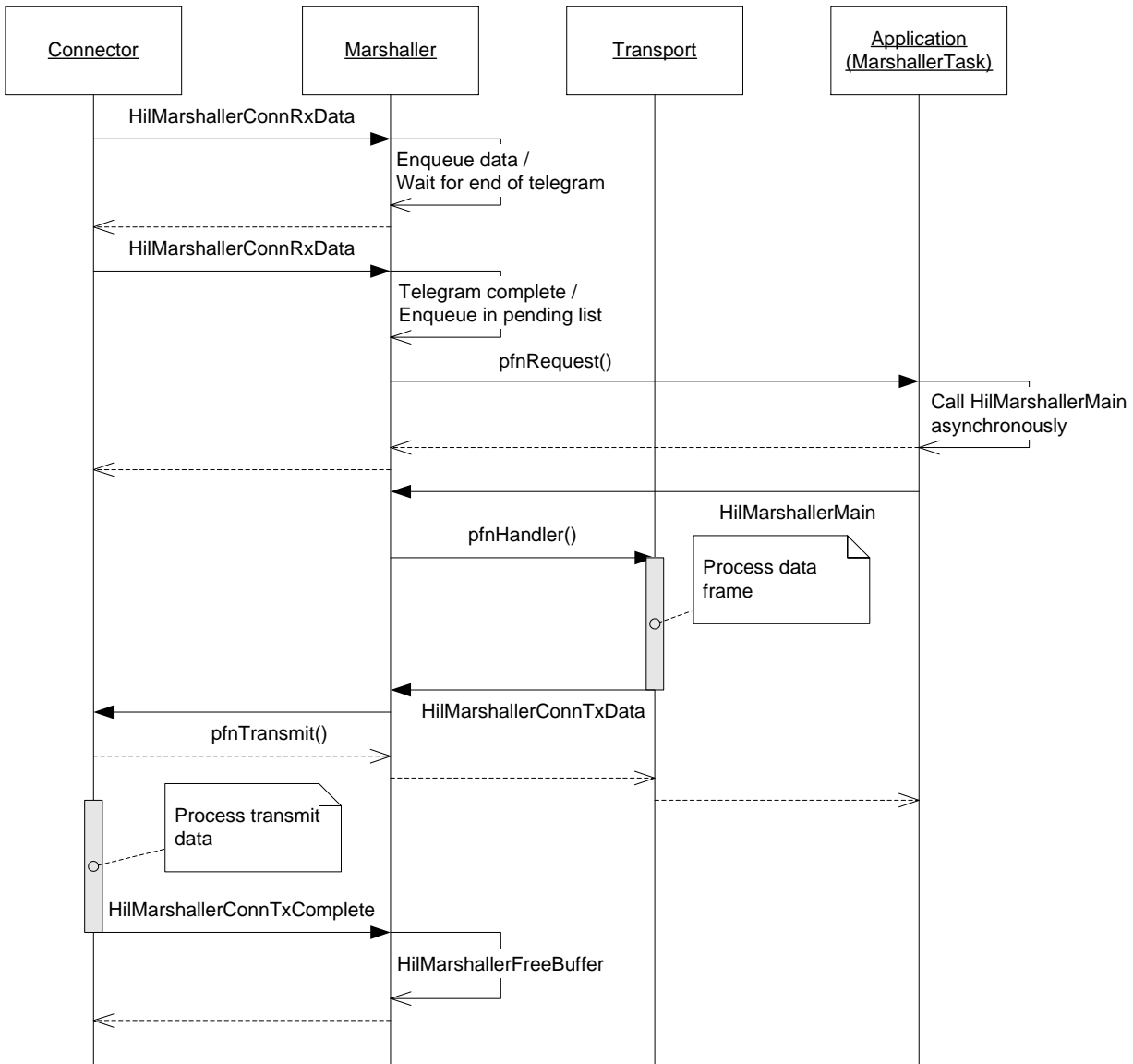


Figure 2: Sequence Diagram - Marshaller Data Processing

3.4.2 Unsolicited Packets / Requests

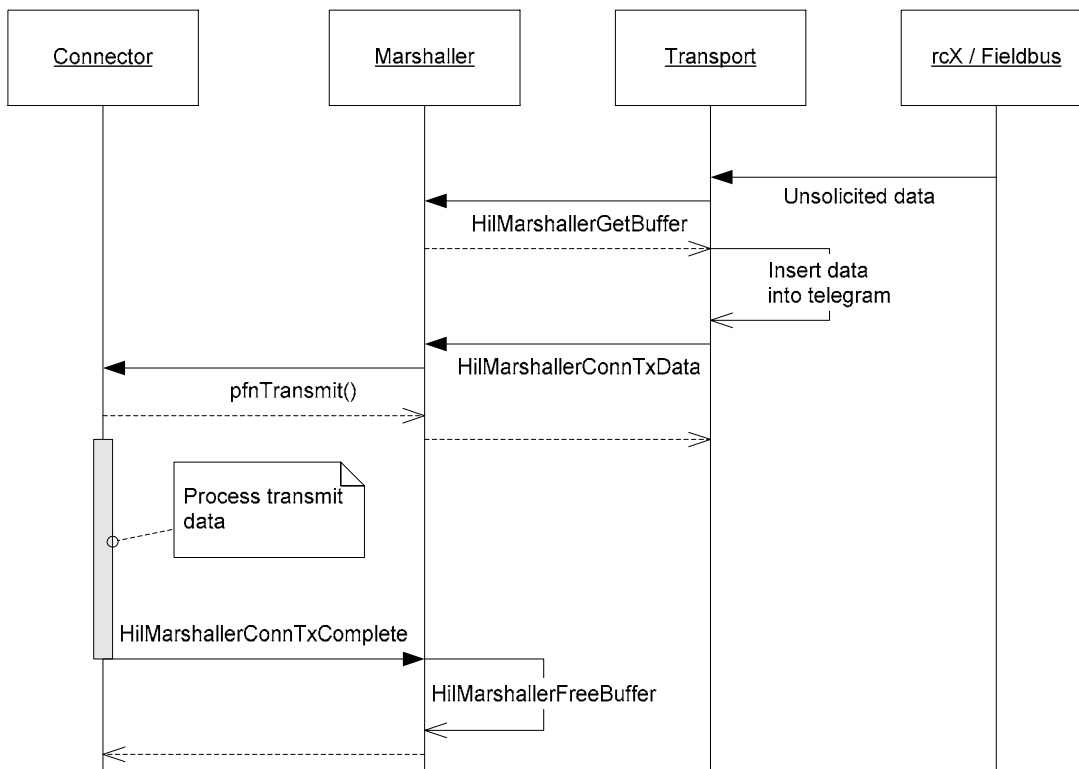


Figure 3: Sequence Diagram - Unsolicited Data Processing

4 TCP Connector Internals

4.1 Overview

The TCP connector is implemented in the form of an application task that features a task message queue for receiving rcX packets and an rcX packet buffer pool for managing the packet it sends to other tasks. This kind of implementation has been chosen to facilitate packet-based communication with the Hilscher TCP stack's application interface task. Interaction with the Marshaller main module is based on the use of callback functions. The TCP connector task is launched by the Marshaller main module, depending on whether a TCP connector configuration is included in the Marshaller's startup parameter set. See section "Configuration Data" on page 45.

4.2 Structure Definitions

The only data structure with public write access used with the TCP connector is the configuration data set which is usually instantiated once in the firmware configuration file (config.c). This structure is described in chapter "Configuration Data" on page 45.

In addition, the TCP connector instantiates several data structures for internal use which are also registered at the rcX task diagnostics interface and consequently can be made visible in the firmware diagnostics dialog of SYCON.net in addition to the standard task information.

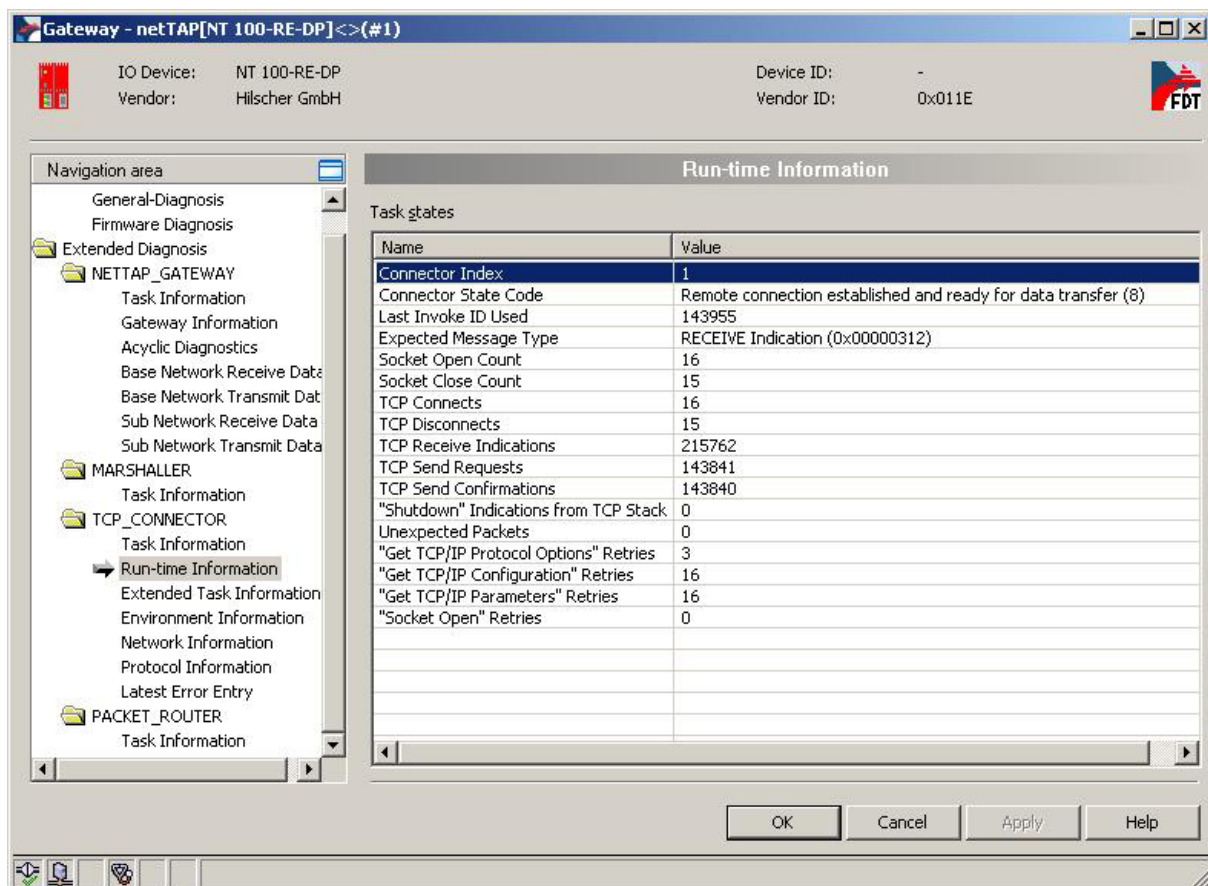


Figure 4: TCP Connector Diagnostics in SYCON.net

4.2.1 Run-time Information (TCP_CONNECTOR_MGT_INFO_T)

The following structure contains all information that is closely linked to the connector's internal state handling and performance monitoring.

Element	Type	Description
ulConnectorIndex	UINT32	connector index passed at initialization time
ulConnectorState	UINT32	connector task state
ulInvokeID	UINT32	invoke ID of the last packet sent (0 = no packet sent, yet)
ulExpectedMessage	UINT32	command of expected confirmation for outstanding request
ulOpenCount	UINT32	socket open count
ulCloseCount	UINT32	socket close count
ulConnectCount	UINT32	TCP remote connect count
ulDisconnectCount	UINT32	TCP remote disconnect count
ulReceiveIndCount	UINT32	TCP receive indication count
ulSendReqCount	UINT32	TCP send request count
ulSendCnfCount	UINT32	TCP send confirmation count
ulShutdownIndCount	UINT32	shutdown indication from the TCP stack
ulUnexpectedCount	UINT32	unexpected message count
ulGetOptionsRetryCount	UINT32	retry count for TCP_CONNECTOR_STATE_GET_OPTIONS
ulGetConfigRetryCount	UINT32	retry count for TCP_CONNECTOR_STATE_GET_CONFIG
ulGetParamRetryCount	UINT32	retry count for TCP_CONNECTOR_STATE_GET_REMOTE_MAC
ulOpenRetryCount	UINT32	retry count for TCP_CONNECTOR_STATE_OPEN

Table 11: TCP Connector Run-time Information Structure

4.2.2 Extended Task Information (TCP_CONNECTOR_TASK_INFO_T)

The following structure contains all information that is closely linked to the connector's implementation as a message handler task.

Element	Type	Description
szTaskName	STRING	task name as (max. 16 characters including NUL termination)
hTask	HANDLE	task handle assigned by the operating system
eTaskToken	UINT16	task token as defined in the connector configuration
eTaskPriority	UINT16	task priority as defined in the connector configuration
uiTaskInstance	UINT32	task instance number as defined in the connector configuration
szQueueName	STRING	task message queue name
hQueue	HANDLE	task message queue handle assigned by the operating system
ulQueueSize	UINT32	max. no. of queue entries as defined in the connector configuration
szPoolName	STRING	task packet buffer pool name
hPool	HANDLE	task packet pool handle assigned by the operating system
ulPoolSize	UINT32	no. of buffer entries as defined in the connector configuration

Table 12: TCP Connector Extended Task Information Structure

4.2.3 Environment Information (TCP_CONNECTOR_ENV_INFO_T)

The following structure contains all information that is closely linked to the connector's environment, i.e. the tasks it directly interfaces with.

Element	Type	Description
szTcpTaskName	STRING	TCP/IP stack interface task name as defined in the connector configuration (max. 16 characters including NUL termination)
hTcpTask	HANDLE	TCP/IP stack interface task handle assigned by the operating system
eTcpTaskToken	UINT16	TCP/IP stack interface task token assigned by the operating system
eTcpTaskPriority	UINT16	TCP/IP stack interface task priority assigned by the operating system
uiTcpTaskInstance	UINT32	TCP/IP stack interface task instance number as defined in the connector configuration
szTcpQueueName	STRING	TCP/IP stack interface task message queue name as defined in the connector configuration
hTcpQueue	HANDLE	TCP/IP stack interface task message queue handle assigned by the operating system
szMarshallerTaskName	STRING	Marshaller task name as defined in the connector configuration (max. 16 characters including NUL termination)
hMarshallerTask	HANDLE	Marshaller task handle assigned by the operating system
eMarshallerTaskToken	UINT16	Marshaller task token assigned by the operating system
eMarshallerTaskPriority	UINT16	Marshaller task priority assigned by the operating system
uiMarshallerTaskInstance	UINT32	Marshaller task instance number as defined in the connector configuration
szMarshallerQueueName	STRING	Marshaller task message queue name as defined in the connector configuration
hMarshallerQueue	HANDLE	Marshaller task message queue handle assigned by the operating system

Table 13: TCP Connector Environment Information Structure

4.2.4 Network Information (TCP_CONNECTOR_NET_INFO_T)

The following structure contains all information that is closely linked to the connector's network connection and addressing.

Element	Type	Description
ulSubnetMask	UINT32	subnet mask (0 = not yet known / configured)
ulDefaultGateway	UINT32	default gateway address (0 = not yet known / configured)
ulLocalIpAddress	UINT32	local IP address (0 = not yet known / configured)
ulRemoteIpAddress	UINT32	remote IP address (for established connection, otherwise 0)
ulLocalPort	UINT32	local port number, typically HIL_TRANSPORT_IP_PORT (50111)
ulRemotePort	UINT32	remote port number (for established connection, otherwise 0)
abLocalMACAddress	UINT8[6]	local MAC address (0 = not yet known / configured)
abRemoteMACAddress	UINT8[6]	remote MAC address (for established connection, otherwise 0)

Table 14: TCP Connector Network Information Structure

4.2.5 Protocol Information (TCP_CONNECTOR_PROTOCOL_INFO_T)

The following structure contains all information that is closely linked to the connector's protocol and socket handling.

Element	Type	Description
ulSocket	UINT32	socket handle assigned by the TCP/IP stack on open
ulSupportedProtocols	UINT32	list of supported protocols above the IP level in the form of a bitmap
ulSelectedProtocol	UINT32	identifier of selected protocol as returned on socket open (i.e. TCP)
ulMaxHops	UINT32	"Time to Live": maximum number of gateway hops for outgoing Marshaller TCP frames (either configured or the TCP stack's default)
ulPacketTimeout	UINT32	send timeout for local inter-task packets (0 = no timeout)
ulSendTimeout	UINT32	send timeout for remote transmission over the network (either configured or the TCP stack's default)
ulIdleTimeout	UINT32	timeout for automatic shutdown of an established connection if no Marshaller frames are exchanged (either configured or the TCP stack's default)
ulCloseTimeout	UINT32	close timeout for graceful shutdown of an established Marshaller TCP connection (as configured, 0 = immediate abort)

Table 15: TCP Connector Protocol Information Structure

4.2.6 Latest Error Entry (RCX_ERROR_BUFFER_ENTRY_T)

The following structure contains all information stored when the TCP connector detects an error.

Element	Type	Description
usEntryNumber	UINT16	entry number (incremented with each entry made)
usReserved	UINT16	reserved for future use, set to zero
ulTimeStampH	UINT32	system time when the entry was made (high order DWORD)
ulTimeStampL	UINT32	system time when the entry was made (low order DWORD)
ulModule	UINT32	numeric identifier of the source module where the error was detected
ulLine	UINT32	source code line number where the error was detected
ulErrorCode	UINT32	error code (see section "Marshaller Error Codes" on page 54)
ulErrorValue	UINT32	offending parameter value, state value, OS error code, etc.
ulDetail1	UINT32	violated lower limit, index value, etc. (depends on context)
ulDetail2	UINT32	violated upper limit, index value, etc. (depends on context)

Table 16: TCP Connector Latest Error Entry Structure

4.3 TCP Connector State Machine

In order to perform proper initialization and connection management, the TCP connector implements a state machine shown in figure below.

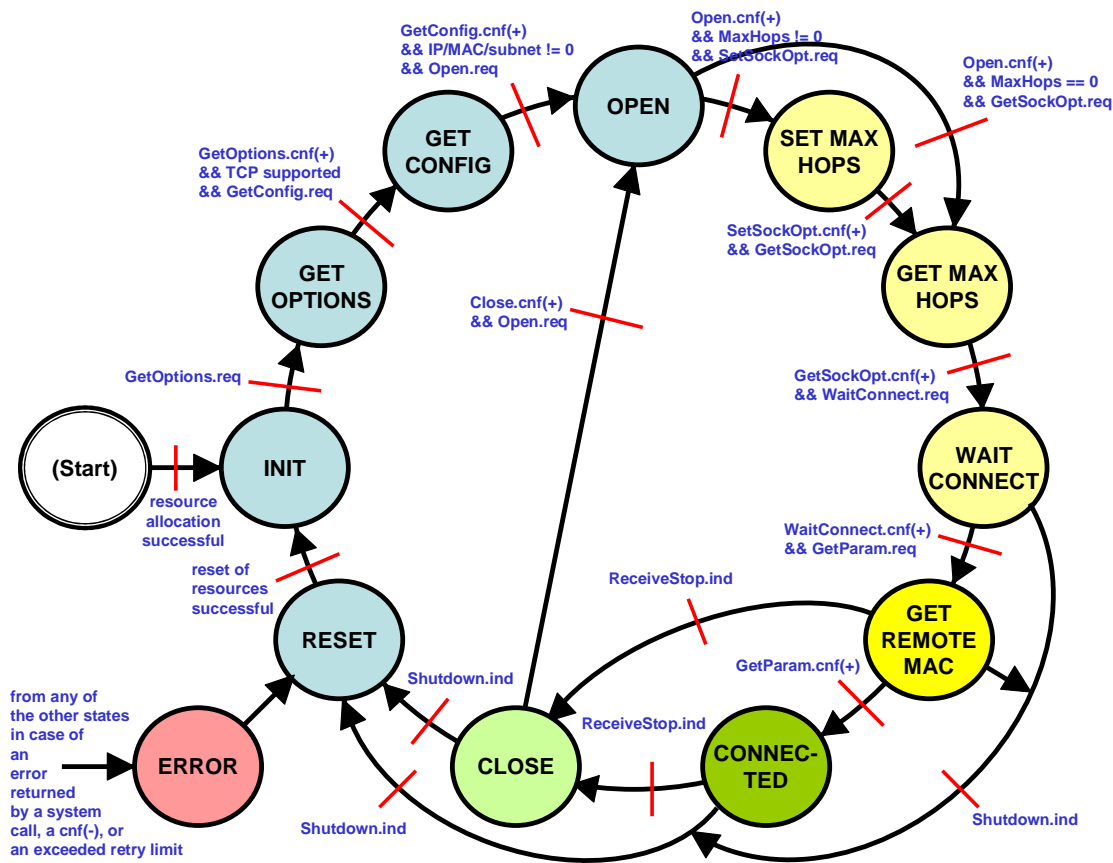


Figure 5: TCP Connector State Machine

The following table gives an explanation of each of the connector's states. Each description begins with the corresponding text shown in SYCON.net when equipped with diagnostics XML files that contain the TCP connector diagnostics structure descriptions. (Further information given in brackets.)

Note: The state #defines in the source code carry the prefix TCP_CONNECTOR_STATE_ which has been omitted in the table for better readability.

State	Value	Description
INIT	0	Initializing (Resets the retry counters for the setup / information requests. Sends the GetOptions request and, if successful, switches to GET_OPTIONS.)
GET_OPTIONS	1	Info on supported protocols requested (Evaluates the supported protocols bitmap in the GetOptions confirmation from the TCP stack. Repeats GetOptions until confirmation positive. If GetOptions retry limit exceeded or TCP not supported, switches to ERROR state to force cleanup and restart. If TCP supported, sends the GetConfig request and, if successful, switches to GET_CONFIG.)
GET_CONFIG	2	Info on TCP stack configuration requested (Evaluates the local address parameters in the GetConfig confirmation from the TCP stack. Repeats GetConfig until address data valid. If GetConfig retry limit exceeded, switches to ERROR state to force cleanup and restart. If address data valid, sends the Open request and, if successful, switches to OPEN.)
OPEN	3	Socket Open requested (Takes socket handle and local IP address from the Open confirmation from the TCP stack. Repeats Open until data valid. If Open retry limit exceeded, switches to ERROR state to force cleanup and restart. If data valid and max. number of hops configured, sends a SetSockOption request to configure TTL on the open socket and, if successful, switches to SET_MAX_HOPS. If data valid and max. number of hops not configured, sends a GetSockOption request to retrieve TTL from the TCP stack for the open socket and, if successful, switches to GET_MAX_HOPS.)
SET_MAX_HOPS	4	Setup of the maximum number of hops in the TCP stack requested (Repeats SetSockOption request until successful. If SetSockOption retry limit exceeded, switches to ERROR state to force cleanup and restart. If TTL setup confirmed, sends a GetSockOption request to retrieve TTL from the TCP stack for the open socket and, if successful, switches to GET_MAX_HOPS.)
GET_MAX_HOPS	5	Info on the maximum number of hops in the TCP stack requested (Repeats GetSockOption request until successful. If GetSockOption retry limit exceeded, switches to ERROR state to force cleanup and restart. If TTL setting retrieved successfully, sends a WaitConnect request to listen on the open socket and, if successful, switches to WAIT_CONNECT.)
WAIT_CONNECT	6	Waiting for remote connect request (Takes remote IP address from the WaitConnect confirmation from the TCP stack. Sends a GetParams request to retrieve complete set of remote address information (including MAC address) and, if successful, switches to GET_REMOTE_MAC. Handles unexpected Shutdown indication from the TCP stack gracefully by switching to RESET state.)
GET_REMOTE_MAC	7	Info on the remote MAC address requested (Takes remote IP and MAC address from the GetParams confirmation from the TCP stack and, if successful, switches to CONNECTED. Handles unexpected Shutdown indication from the TCP stack gracefully by switching to RESET state. Handles unexpected ReceiveStop indication from the TCP stack gracefully by sending Close request and, if successful, by switching to CLOSE state. Re-queues early Receive indications in order not to lose them while still waiting for the GetParams confirmation.)
CONNECTED	8	Remote connection established and ready for data transfer (Handles Receive indications by passing the received data to the Marshaller main module's callback. Transmits data passed by the Marshaller main module and confirms send completion. Handles unexpected Shutdown indication from the TCP stack gracefully by switching to RESET state. Handles unexpected ReceiveStop indication from the TCP stack gracefully by sending Close request and, if successful, by switching to CLOSE state.)

State	Value	Description
CLOSE	9	Socket Close requested Waits for Close confirmation from the TCP stack and, on receipt, resets socket handle and remote address information. Then sends the Open request and, if successful, switches to OPEN. Handles unexpected Shutdown indication from the TCP stack gracefully by switching to RESET state. Handles unexpected ReceiveStop indication from the TCP stack gracefully by requesting Close/Abort request as needed.
RESET	10	Resetting (Waits for trailing packets and rejects them or releases them to the buffer pool. Sends Close request, if socket still open and waits for the Close confirmation. Then switches to INIT.)
ERROR	11	Error (Actually an intermediate state that advances unconditionally to RESET.)

Table 17: TCP Connector State Descriptions

5 Porting the Marshaller

This chapter describes how to port the existing Marshaller modules to a custom operating system.

rcX Sample implementations of the following components are available:

- rcX packet router for the Marshaller
- serial connection via one or more UARTs integrated in the netX chip
- USB connection via the USB core integrated in the netX chip
- TCP/IP connection via Ethernet on port 50111 (under development)

5.1 Directory Structure

Directory		Contents
Marshaller		Operating system independent Marshaller core modules Remote cifX API module Marshaller main component
	rcX	Sample implementation of UART Connector (USB and "real" UART) rcX Packet Router O/S Abstraction
	sys	Free linked list implementation (BSD)
	machine	Needed header files for the BSD linked list implementation
Documentation		Marshaller Documentation
targets		Subdirectories for O/S specific examples
	rcX	
		libs
		Libraries (not included) rcX V2.0.4.6 needs to be placed under libs/rcX rcX SHM V0.930 needs to be placed under libs/rcX_SHM
		Marshaller
		Build environment for Marshaller library (libmarshaller.a)
		rcXFirmwareInfo
		Version information for Firmware in <i>TestApp</i>
		TestApp
		Test firmware including Marshaller
	Win32	
		cifXTCPServer
		Visual Studio 2003 project for a TCP/IP Server accessing the cifX Device Driver through Marshaller functions. Listens on TCP port 50111

Table 18 : Marshaller Directory Structure

Note: Adding the OS dependent part into a separate subdirectory is recommended.

5.2 Implementing OS Dependencies

There are few functions which are needed by the Marshaller main module to adapt it to another operating system. To avoid the need for new global functions, the adaptation is done via macros, so it's the user choice if new global functions need to be implemented or if redefinition of the macros is sufficient.

The following table shows the needed macros:

Macro	Calling convention	Description
OS_LOCK	lock = OS_LOCK()	Used for locked access to the linked list. - - Can be a global mutex or an IRQ lock
OS_UNLOCK	OS_UNLOCK(lock)	Used for unlocked access to the linked list.
OS_MALLOC	mem = OS_MALLOC(size);	Returns a dynamically allocated chunk of memory
OS_FREE	OS_FREE(mem)	Frees a previously allocated memory chunk
OS_STRNICMP	int = OS_STRNICMP(sz1, sz2, maxsize)	Compares two strings (case-insensitive) - Returns 0 if strings are equal
OS_GETTICKCOUNT	ticks = OS_GETTICKCOUNT()	Returns the system tick count. - This function is used to generate a unique identifier (e.g. Keep-Alive)

Table 19: Operating System Abstraction

Windows 2000/XP sample ("OS_Include.h"):

```
#include "Windows.h"
/* SLIST_ENTRY is also defined in Windows.h, so we need to undefine it here */
#undef SLIST_ENTRY

#include <string.h>
#include <stdlib.h>
#include <malloc.h>

#define OS_MALLOC      malloc
#define OS_FREE        free
#define OS_STRNICMP    strnicmp
#define OS_GETTICKCOUNT GetTickCount

/* The critical section needs to be allocated in the main module */
extern CRITICAL_SECTION g_tcsQueueLock;

int OS_LOCK(void)
{
    EnterCriticalSection(&g_tcsQueueLock);
    return 1;
}

void OS_UNLOCK(int iLock)
{
    LeaveCriticalSection(&g_tcsQueueLock);
}
```

5.3 Implementing a Connector

5.3.1 Connector Initialization

Each connector module must have an own initialization function, which is defined as follows:

```
TLR_RESULT ConnectorInit( const HIL_MARSHALLER_CONNECTOR_PARAMS_T* ptParams,
void* pvMarshaller)
```

Arguments:

Argument	Data type	Description
ptParams	HIL_MARSHALLER_CONNECTOR_PARAMS_T	Initialization data
pvMarshHandle	void*	Marshaller handle returned by <i>HilMarshallerStart()</i>

Return Values:

Return Values	
TLR_S_OK	Successfully initialized connector
Connector specific error otherwise	

This function must setup all internal structures of the connector and register itself as a valid connector at the given Marshaller instance. It can be called during startup of the Marshaller by inserting it into the automatically loaded connectors list or at runtime by the main application.

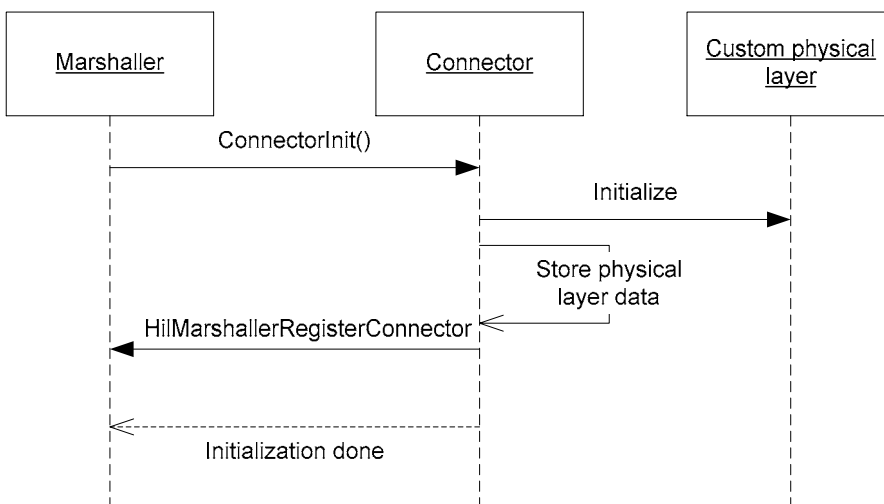


Figure 6: Sequence Diagram - Connector Registration

Connector specific initialization data can be given via '*ptParams-> pvConfigData*' structure element.

5.3.2 Registration Data

Connectors need to register themselves at a Marshaller instance and need to pass the following data during registration (see structure `HIL_MARSHALLER_CONNECTOR_T`):

Element	Description
<code>pfnTransmit</code>	Function called by Marshaller to send data to physical layer
<code>pfnDeinit</code>	Called during shutdown of Marshaller
<code>pfnPoll</code>	Optional function called with every timer tick (<i>HilMarshallerTimer()</i>) to allow cyclic action. Can be NULL if not used
<code>pvUser</code>	Pointer that will automatically be passed on every call of the functions above
<code>ulDataBufferSize</code>	Size of data buffers to allocate (passed during initialization in <i>ptParams</i>)
<code>ulDataBufferCnt</code>	Number of data buffers to allocate (passed during initialization in <i>ptParams</i>)

Table 20: Connector Registration Data

During registration, a number called *Connector Index*, will be assigned to the connector. This number must be used in all further connector specific Marshaller calls (e.g. *HilMarshallerConnRxData()*).

5.3.3 Data Reception

A connector is expected to provide incoming data (raw binary data), event based. Usually a thread is used for this task. With every new arriving data, the connector must call the Marshaller function *HilMarshallerConnRxData()*.

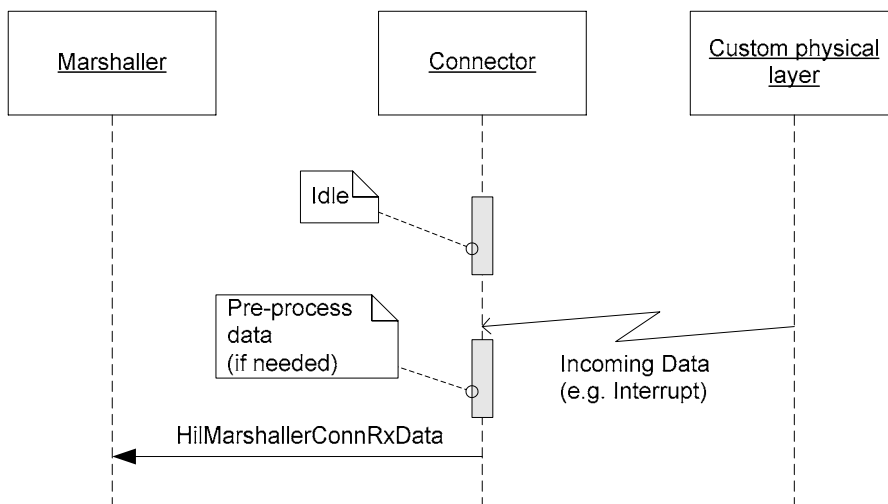


Figure 7: Sequence diagram - Connector Data Reception

5.3.4 Data Transmission

A complete transfer buffer will be passed to a connector via the *pfnTransmit* function. The connector can decide if it queues the data and processes it asynchronously or passed the data synchronously to the physical layer. If synchronous processing takes a longer amount of time, it is recommended to process the sending asynchronously.

Once the frame has been transmitted (or aborted) the connector needs to call *HilMarshallerConnTxComplete*.

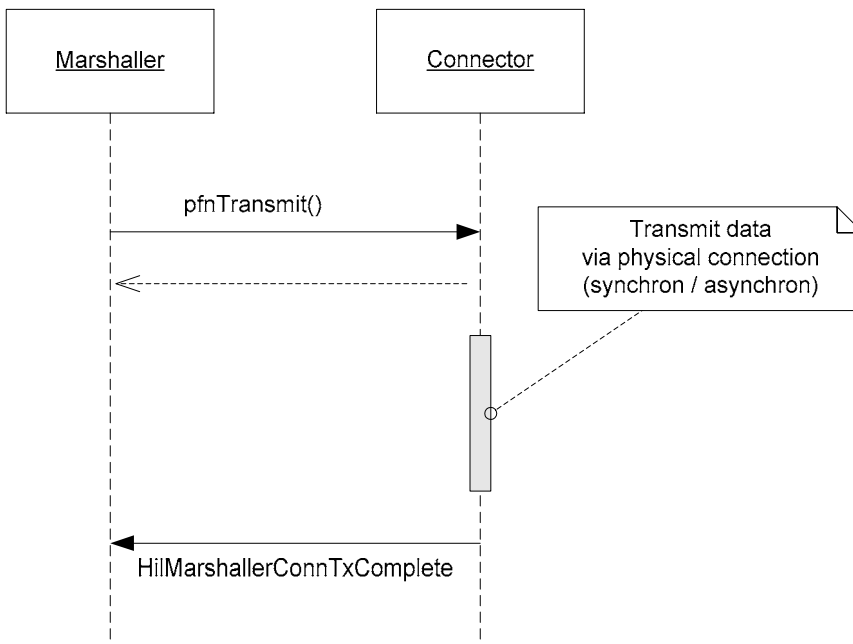


Figure 8: Sequence diagram - Connector Data Transmission

5.4 Implementing a Custom Transport

Custom transports need a unique data layer number. This number will be used to forward incoming data packets to the correct transport layer.

5.4.1 Initialization Function

Each transport module must have an own initialization function, which is defined as follows:

```
TLR_RESULT TransportInit( void* pvMarshaller,
void* pvConfig)
```

Arguments:

Argument	Data type	Description
pvMarshaller	void*	Marshaller handle returned by <i>HilMarshallerStart()</i>
pvConfig	void*	Transport specific initialization structure

Return Values:

Return Values	
TLR_S_OK	Successfully initialized connector
Transport specific error otherwise	

This function must setup all internal structures of the transport layer and register itself as a valid layer at the given Marshaller instance. A transport layer must always be known by the Marshaller at startup and cannot be initialized or registered during runtime.

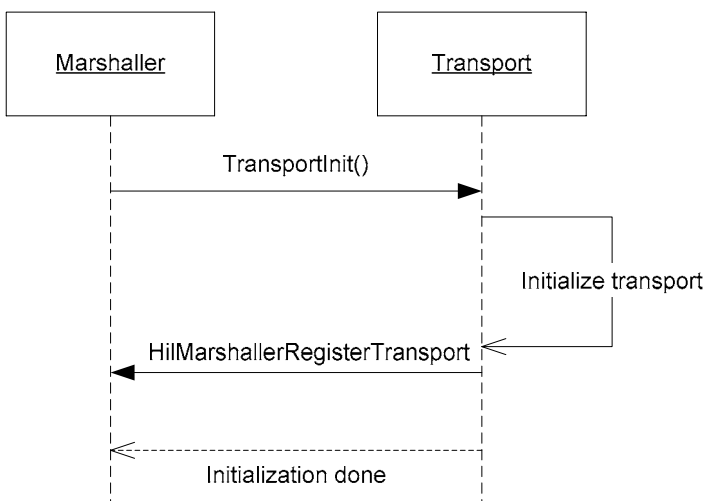


Figure 9: Sequence Diagram - Transport Registration

5.4.2 Registration

Transport layers need to register themselves at a Marshaller instance and need to pass the following data during registration (see structure `TRANSPORT_LAYER_DATA_T`):

Element	Description
<code>pfnHandler</code>	Handler that will handle incoming data packets with the given data type (see <i>usDataType</i>)
<code>pfnDeinit</code>	Called during shutdown of Marshaller
<code>pfnPoll</code>	Optional function, called with every timer tick (<i>HilMarshallerTimer()</i>) to allow cyclic action. Can be NULL if not used
<code>pvUser</code>	Pointer that will automatically be passed on every call of the functions above
<code>usDataType</code>	Data type this transport layer is able to handle

Table 21: Transport Registration Data

5.4.3 Data Handling

The transport layer is responsible to correctly handle the *Transport Header* of a frame and to process the data portion of the frame. Once the request is completely handled it needs to send an answer packet with the same *Transport Header* content and the returned data. The buffer pointer, passed during *pfnHandler* call, will be valid until the transport frees the buffer by calling *HilMarshallerFreeBuffer()* or it passes the answer packet (using this buffer) by calling *HilMarshallerConnTxData()*.

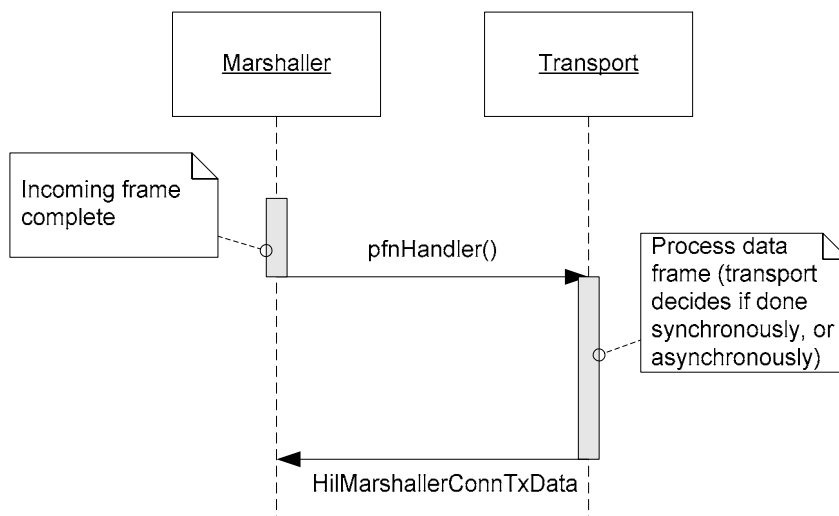


Figure 10: Sequence Diagram - Transport Data Handling

6 rcX Marshaller Sample

6.1 Overview

The rcX Marshaller sample consists of the following parts:

- rcX O/S adaptation of the Marshaller
- UART connector component (also used for USB using rcX V2.0.4.6 or later)
- rcX packet router (transport component)
- Application task to handle initialize and operate the Marshaller

The following figure shows the integration of the netX Marshaller into a typical rcX based firmware scenario:

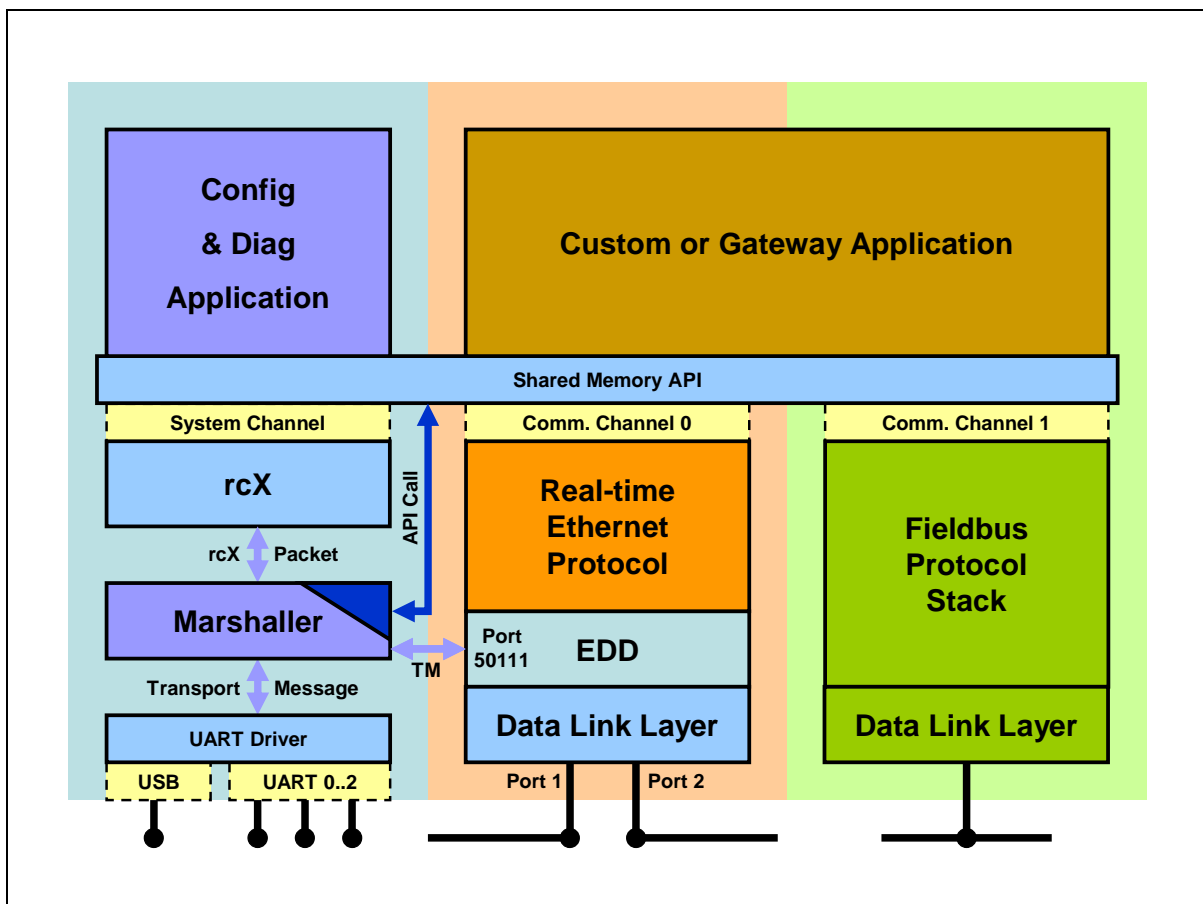


Figure 11: rcX Marshaller Integration in a Typical Firmware Scenario

Available Target Connections:

Interface	Driver	Description
RS-232	UART driver	Uses a standard UART driver. Default settings: Baudrate: 115.2 kBaud Data size: 8 Bit Parity: no Parity Stop Bits: 1 Flow control: no Handshake
USB	UART driver (since rcX V2.0.4.6)	Offers a USB CDC device on the host (serial emulation)
TCP/IP (Ethernet)	TCP/IP Stack	TCP/IP server listening on Port 50111 This interface is currently under development

Table 22: rcX Target Connections

6.2 Adding to an existing Firmware

The netX Marshaller for an rcX based firmware is provided as a static library (libmarshaller.a). To implement the Marshaller into an existing firmware, the following steps need to be processed:

- Adding the library to the linker parameters list
- Adding the configuration entries to the rcX configuration file
- Implementing a task to start and operate the Marshaller functionality

6.2.1 Marshaller Task

The Marshaller task is not a part of the Marshaller library, because creating a task depends on the operating system.

The task for the rcX operating system is included as a source code module. The task receives the Marshaller main configuration structure and can directly be included in an rcX project.

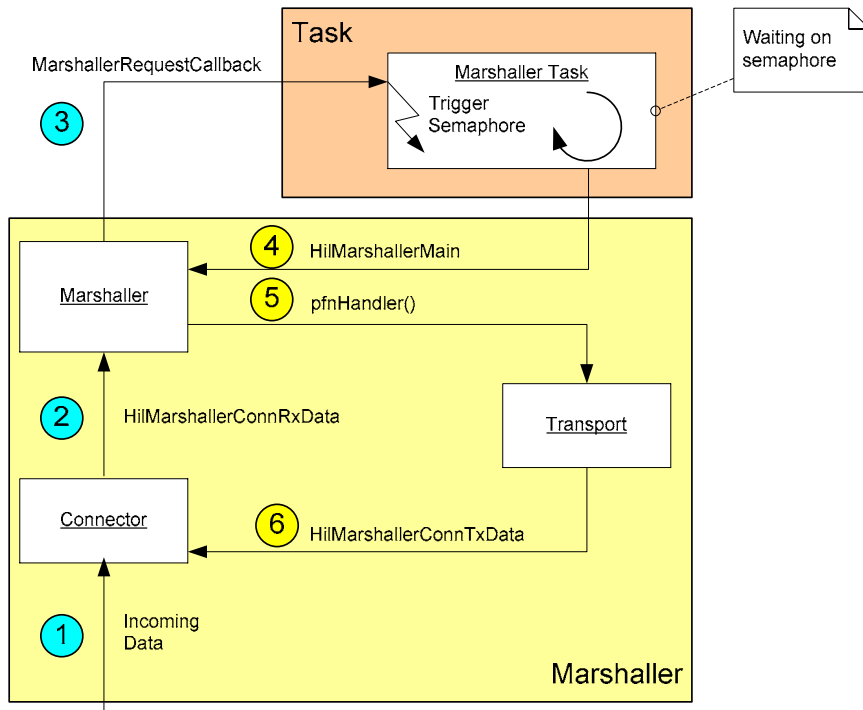


Figure 12: rcX Marshaller Task - Structure

Step	Caused by	Action
1		Connector prepares data
2	Connector indicating new data	Parse data stream and enqueue completely arrived telegrams
3	Telegram finished detection	Call function ' <i>MarshallerRequestCallback</i> ' to indicate the Marshaller task, it needs to call ' <i>HilMarshallerMain</i> ' to do the processing. The Callback function signals a semaphore to activate the task and process the request asynchronously.
4	Semaphore was signaled	Call ' <i>HilMarshallerMain</i> ' to do the processing of incoming telegrams
5		Pass telegrams to the corresponding <i>Transport Layer</i> , by calling the registered handler function
6	Processing completed by transport layer	Call ' <i>HilMarshallerConnTxData</i> ' to send the answer to the processed request.

Note: Step 1-3 are processed in the connector context, while steps 3-6 are handled in the Marshaller task context, so the Marshaller/Connector can continue with the handling of incoming data.

6.2.2 Configuration Data

Connectors and the rcX packet transport layer are operating system specific components. Their configuration structures and internal structure are described in the following chapters.

6.2.2.1 rcX Packet Transport

The rcX Packet transport uses a separate rcX packet queue for each possible connector and channel on the device. When a new telegram arrives it is stored in a pending request list, after being sent to the destination mailbox. When the answer arrives, the stored data are used to generate the answer. When an unsolicited packet arrives the transport requests a new transmission buffer and sends it directly to the connector.

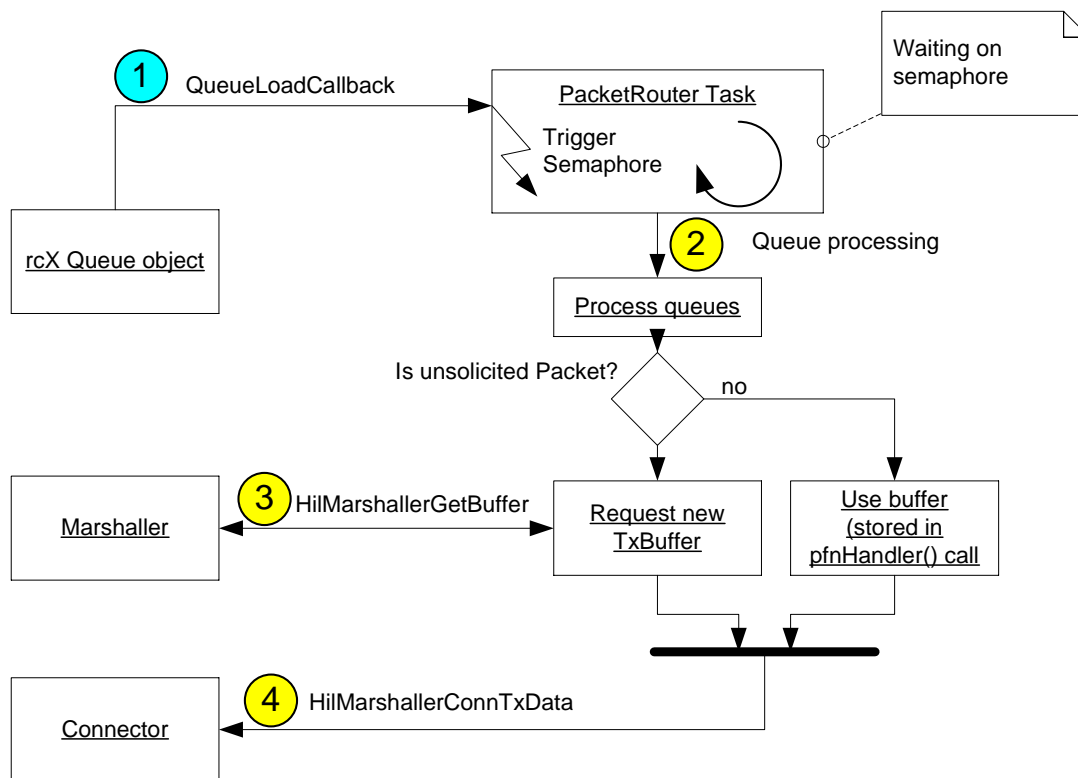


Figure 13: rcX Packet Transport - Structure

Note: Currently a separate task is needed under rcX. It is planned to remove this task and let the Marshaller task also handle the packet router.

Configuration structure (PACKET_TRANSPORT_CONFIG_T):

Element	Type	Description
eTaskPriority	RX_TASK_PRIORITY	Priority of the PacketRouter task
eTaskToken	RX_TASK_TOKEN	Token of the PacketRouter task
szTaskname	STRING*	Name of the PacketRouter task
ulTaskInstance	UINT32	Instance of the PacketRouter task
ulStackSize	UINT32	Stack size (in DWORDS) of the PacketRouter task
ulQueueEntries	UINT32	Number of queue entries to allocate per channel and connector
ulMaxChannels	UINT32	Number of channels the transport must be able to handle

Table 23: rcX Packet Transport - Configuration Structure

6.2.2.2 UART Connector

The rcX UART connector needs a pre-configured UART (usually done via rcX configuration file).

The following settings are mandatory, to communicate with standard SYCON.NET / ODM:

- Baudrate : 115.200
- Databits : 8
- Parity : None
- Stopbits : 1
- Handshake : none

It is also recommended to use interrupt mode for UARTs and to enable UART FIFO's by setting the threshold to 1 (for rcX Version 2.0.4.5 and earlier) or to 8 (starting at V2.0.4.6). Setting the threshold to higher values will decrease interrupt load and is known to work with rcX V.2.0.4.6.

Configuration structure (UART_CONN_CONFIG_T):

Element	Type	Description
szUartName	STRING*	Name of the UART to use
ulUartInst	UINT32	Instance of the UART to use
fInterrupt	BOOLEAN	FALSE: Enable invocation of the UART connector's polling callback from within the Marshaller's cyclic timer callback. TRUE: Enable invocation of the UART connector's interrupt callback from within the UART driver's interrupt handler. Note: This needs correct interrupt configuration in rcX and in the following to elements
szIrqName	STRING*	Name of the UART Interrupt. Must be set when using interrupt mode.
ullrqlnst	UINT32	Instance of the UART Interrupt. Must be set when using interrupt mode.

Table 24: rcX-based UART Connector - Configuration Structure

6.2.2.3 TCP Connector

The rcX TCP connector requires the Hilscher TCP/IP stack to be present in the system. As the interface of this protocol stack is managed by a stack interface task, the TCP connector parameters provide information about this task and its interface for external communication, i.e. its message queue. Similar information is given for the Marshaller task and for the TCP connector itself. The parameter set is completed by TCP specific settings.

Configuration structure (TCP_CONNECTOR_CONFIG_T):

Element	Type	Description
ulParameterVersion	UINT32	parameter structure version (major/minor, binary encoding, e.g. 0x0003000C = V3.12)
szMarshallerTaskName	STRING	Marshaller task name as defined in config.c (max. 16 characters including NUL termination)
uiMarshallerTaskInstance	UINT32	Marshaller task instance number as defined in config.c
szTcpTaskName	STRING	TCP/IP stack interface task name as defined in config.c (max. 16 characters including NUL termination)
uiTcpTaskInstance	UINT32	TCP/IP stack interface task instance number as defined in config.c
szTcpTaskQueueName	STRING	TCP stack interface task message queue name (max. 16 characters including NUL termination)
eConnectorTaskPriority	UINT16	connector task priority (valid rcX task priority used by the Marshaller when instantiating the TCP connector, typically lower than other protocol stack task priorities)
eConnectorTaskToken	UINT16	connector task token (valid rcX task token used by the Marshaller when instantiating the TCP connector)
uiConnectorTaskInstance	UINT32	connector task instance number (determines the channel on which the connector's diagnostics information is shown in SYCON.net)
ulConnectorQueueSize	UINT32	max. number of packets that the connector's message queue can buffer (typically messages from the TCP stack interface task)
ulConnectorPoolSize	UINT32	number of packet buffers in the connector's message buffer pool (typically used for messages to the TCP stack interface task)
ulLocalPort	UINT32	TCP port number for listening (typically the Hilscher Transport port number, i.e. 50111)
ulMaxHops	UINT32	"Time to Live": maximum number of gateway hops for outgoing Marshaller TCP frames (1...255, 0 = use the TCP stack's default)
ulSendTimeout	UINT32	send timeout for remote transmission over the network (0 = use the TCP stack's default)
ulIdleTimeout	UINT32	idle timeout for automatic shutdown of an established Marshaller TCP connection if no Marshaller frames are exchanged (0 = no timeout)
ulCloseTimeout	UINT32	close timeout for graceful shutdown of an established Marshaller TCP connection (0 = immediate abort)

Table 25: rcX-based TCP Connector - Configuration Structure

7 Appendix

7.1 Marshaller Configuration Entries in the rcX Configuration File

This chapter shows a sample configuration for a Marshaller running under rcX, a UART connector using a standard UART integrated in the netX, a UART connector using UART emulation on the integrated netX USB interface, and a TCP connector.

7.1.1 UART Connector

The following code shows a UART connector configuration example with interrupt and UART configuration for the netX internal UART 0.

```
#include "rcX/UartConnector.h"
#include "MarshallerConfig.h"

/* Interrupt Configuration */
STATIC CONST FAR RX_INTERRUPT_SET_T atrXInt[] =
{
    { {"INT_UART", RX_PERIPHERAL_TYPE_INTERRUPT, 0}, // UART 0
      SRT_NETX_VIC_IRQ_STAT_uart0,                // UART IRQ status bit
      27,                                           // interrupt priority
      RX_INTERRUPT_MODE_TASK,                      // create an IRQ handler task
      RX_INTERRUPT_EOI_AUTO,                      // EOI handled automatically
      RX_INTERRUPT_TRIGGER_RISING_EDGE,           // edge triggered
      RX_INTERRUPT_PRIORITY_STANDARD,             // normal priority
      RX_INTERRUPT_REENTRANCY_DISABLED,           // interrupt is not reentrant
      TSK_PRIO_7,                                  // interrupt task priority
      TSK_TOK_7,                                  // interrupt task token
      1024,                                        // interrupt task stack size
    }
};

/* UART configuration */
STATIC CONST FAR RX_UART_SET_T atrXUrt[] =
{
    { {"REALUART", RX_PERIPHERAL_TYPE_UART, 0},
      0,                                           // use integrated UART 0
      RX_UART_BAUDRATE_115200, // baud rate 115,2k
      RX_UART_PARITY_NONE,                       // no parity
      RX_UART_STOPBIT_1,                         // 1 stop bit
      RX_UART_DATABIT_8,                        // 8 data bits
      8,                                           // "rx ready" trigger level
      4,                                           // "tx empty" trigger level
      RX_UART_RTS_NONE,                         // no RTS in use
      RX_UART_RTS_DEFAULT,                      // no RTS in use
      0,                                           // no RTS forerun
      0,                                           // no RTS trail
      RX_UART_CTS_NONE,                        // no CTS in use
      RX_UART_CTS_DEFAULT                      // default CTS handling
    }
};

/* Connector configuration */
static const UART_CONN_CONFIG_T s_tUartConfig =
{ // real UART integrated in the netX
  .szUartName = "REALUART", // name of the UART peripheral device
  .ulUartInst = 0,          // UART device instance number
  .fInterrupt = TRUE,       // use interrupt mode
}
```



```
.szIrqName = "INT_UART", // interrupt name
.ulIrqInst = 0           // interrupt instance number
};

static const HIL_MARSHALLER_CONNECTOR_PARAMS_T s_atConnectorParams[] =
{
    { // virtual UART based on the netX USB interface
        .pfnConnectorInit = UartConnectorInit, // initialization function
        .ulDataBufferSize = 6000,           // size of the RX/TX buffers
        .ulDataBufferCnt = 1,               // number of RX/TX buffers
        .usFlags = 0x0000,                  // connection management flags
        .ulTimeout = 1000,                  // timeout in ms
        .pvConfigData = (void*) &s_tUartConfig // configuration parameter set
    },
};
```

7.1.2 UART Connector (Emulation via USB)

The following code shows a UART connector configuration example with interrupt and UART configuration for an emulated serial port via USB. This feature is available starting with rcX V2.0.4.6.

Note: Configuration of GPIO12 as '*Enable USB*' is necessary on hardware revisions that allow the netX to signal USB ready to the host.

```
#include "rcX/UartConnector.h"
#include "MarshallerConfig.h"

/* Interrupt Configuration */
STATIC CONST FAR RX_INTERRUPT_SET_T atrXInt[] =
{
    { "INT_USB", RX_PERIPHERAL_TYPE_INTERRUPT, 0}, // USB
    SRT_NETX_VIC_IRQ_STAT_usb, // use USB interrupt bit
    28, // interrupt priority
    RX_INTERRUPT_MODE_TASK, // Create a task for IRQ
    RX_INTERRUPT_EOI_AUTO, // EOI handled automatically
    RX_INTERRUPT_TRIGGER_RISING_EDGE, // edge triggered
    RX_INTERRUPT_PRIORITY_STANDARD, // normal priority
    RX_INTERRUPT_REENTRANCY_DISABLED, // interrupt is not reentrant
    TSK_PRIO_6, // interrupt task priority
    TSK_TOK_6, // interrupt task token
    1024 // interrupt task stack size
    }
}

/* Only needed if GPIO12 must be set to enable USB */
STATIC CONST FAR RX_GPIO_SET_T atrXGpio[] =
{
    { {"USBENABLE", RX_PERIPHERAL_TYPE_GPIO, 0},
      12, // GPIO number
      RX_GPIO_TYPE_OUTPUT, // GPIO type
      RX_GPIO_POLARITY_NORMAL, // GPIO polarity
      RX_GPIO_OUTPUTMODE_STANDARD_0, // GPIO mode
      RX_GPIO_COUNTER_NONE, // counter reference (not used here)
      FALSE, // enable counter IRQ (not used here)
      0, // threshold / capture value (not used here)
    },
};

/* UART configuration */
static const FAR RX_UART_SET_T atrXUrt[] =
{
```

```

{
    {"VIRTUART", RX_PERIPHERAL_TYPE_UART, 0},
    16, // UART 16 = USB
    RX_UART_BAUDRATE_115200, // baud rate 115,2k
    RX_UART_PARITY_NONE, // no parity
    RX_UART_STOPBIT_1, // 1 stop bit
    RX_UART_DATABIT_8, // 8 data bits
    0, // "rx ready" trigger level
    0, // "tx empty" trigger level
    RX_UART_RTS_NONE, // no RTS in use
    RX_UART_RTS_DEFAULT, // no RTS in use
    0, // no RTS forerun
    0, // no RTS trail
    RX_UART_CTS_NONE, // no CTS in use
    RX_UART_CTS_DEFAULT, // default CTS handling
    0x00000000, // Management flags
    0x1939, // USB vendor ID (VID)
    0x000B, // USB product ID (PID)
    0x0001, // USB product release number
    "Hilscher GmbH", // USB vendor name
    "NPLC-M100-DP", // USB product name
    "00000000" // USB serial number
}
};

/* Connector configuration */
static const UART_CONN_CONFIG_T s_tUsbUartConfig =
{
    .szUartName = "VIRTUART", // name of the UART peripheral device
    .ulUartInst = 0, // UART device instance number
    .fInterrupt = TRUE, // use interrupt mode
    .szIrqName = "INT_USB", // interrupt name
    .ulIrqInst = 0 // interrupt instance number
};

static const HIL_MARSHALLER_CONNECTOR_PARAMS_T s_atConnectorParams[] = {
{
    // virtual UART based on the netX USB interface
    .pfnConnectorInit = UartConnectorInit, // initialization function
    .ulDataBufferSize = 6000, // size of the RX/TX buffers
    .ulDataBufferCnt = 1, // number of RX/TX buffers
    .usFlags = 0x0000, // connection management flags
    .ulTimeout = 1000, // timeout in ms
    .pvConfigData = (void*) &s_tUsbUartConfig // configuration parameter set
    },
};

```

7.1.3 TCP Connector

The following code shows a TCP connector configuration example.

Note: The TCP_CONNECTOR_xxx definitions used in the example are defined in "TcpConnector.h".

```

#include "rcX/TcpConnector.h"
#include "MarshallerConfig.h"

/* Connector configuration */
static const TCP_CONN_CONFIG_T s_tTcpConnConfig =
{
    // parameter structure version
    .ulParameterVersion = TCP_CONNECTOR_CONFIG_V1,
    // Marshaller task name
    .szMarshallerTaskName = "MARSHALLER",
    // Marshaller task instance

```

```

.uiMarshallerTaskInstance = 0,
// TCP stack interface task name
.szTcpTaskName            = "TCP_UDP",
// TCP stack interface task instance number
.uiTcpTaskInstance        = 0,
// TCP stack interface task message queue name
.szTcpTaskQueueName       = "EN_TCPUDP_QUE",
// connector task priority
.eConnectorTaskPriority    = TSK_PRIO_17,
// connector task token
.eConnectorTaskToken       = TSK_TOK_17,
// connector task instance (also channel for diagnostics)
.uiConnectorTaskInstance  = 0,
// max. number of concurrent messages from the TCP task that can be buffered
.ulConnectorQueueSize     = TCP_CONNECTOR_DEFAULT_QUEUE_SIZE,
// max. number of concurrent messages to the TCP task that can be buffered
.ulConnectorPoolSize      = TCP_CONNECTOR_DEFAULT_POOL_SIZE,
// TCP port number for listening
.ulLocalPort              = TCP_CONNECTOR_DEFAULT_PORT,
// max. no. of gateway hops for a frame (1...255, 0: use the TCP stack's default)
.ulMaxHops                = 0,
// send timeout for remote transmission (0 = use the TCP stack's default)
.ulSendTimeout            = TCP_CONNECTOR_DEFAULT_SEND_TIMEOUT,
// idle timeout for automatic connection shutdown (0 = no timeout)
.ulIdleTimeout            = TCP_CONNECTOR_DEFAULT_IDLE_TIMEOUT,
// close timeout for graceful connection shutdown (0 = immediate abort)
.ulCloseTimeout           = TCP_CONNECTOR_DEFAULT_CLOSE_TIMEOUT
};

static const HIL_MARSHALLER_CONNECTOR_PARAMS_T s_atConnectorParams[] =
{
    { // TCP stack interface
        .pfnConnectorInit = TcpConnectorInit,           // initialization function
        .ulDataBufferSize = 6000,                      // size of the RX/TX buffers
        .ulDataBufferCnt  = 1,                          // number of RX/TX buffers
        .usFlags           = 0x0000,                    // connection management flags
        .ulTimeout         = 1000,                      // timeout in ms
        .pvConfigData      = (void*) &s_tTcpConnConfig // configuration parameter set
    },
};

```

7.1.4 Marshaller Task

```

#include "MarshallerConfig.h" // global Marshaller configuration definitions
#include "rcX/PacketTransport.h" // Marshaller rcX packet transport definitions
#include "cifXTransport.h" // Marshaller cifX API transport definitions

/* Packet router configuration */
static const PACKET_TRANSPORT_CONFIG_T s_tPacketTransportCfg =
{
    // make sure that these settings do not conflict with those in atrXStaticTasks
    .eTaskPriority = TSK_PRIO_15, // task priority
    .eTaskToken    = TSK_TOK_15, // task token
    .szTaskname     = "PacketRouter", // task name
    .ulTaskInstance = 0, // task instance (also channel for diagnostics)
    .ulStackSize    = 1024, // task stack size
    .ulQueueEntries = 8, // max. entries in the task's message queue
    .ulMaxChannels  = 2 // number of channels to handle (including
}; // system channel

/* cifX API marshalling configuration */
static const CIFX_TRANSPORT_CONFIG s_tcifXTransportCfg =
{
    // this is the list of cifX API entry points
    .tDRVFunctions.pfnxDriverOpen      = xDriverOpen,
    .tDRVFunctions.pfnxDriverClose     = xDriverClose,
    .tDRVFunctions.pfnxDriverGetInformation = xDriverGetInformation,
};

```

```

.tDRVFunctions.pfnxDriverGetErrorDescription = xDriverGetErrorDescription,
.tDRVFunctions.pfnxDriverEnumBoards         = xDriverEnumBoards,
.tDRVFunctions.pfnxDriverEnumChannels       = xDriverEnumChannels,
.tDRVFunctions.pfnxDriverMemoryPointer      = xDriverMemoryPointer,
.tDRVFunctions.pfnxSysdeviceOpen            = xSysdeviceOpen,
.tDRVFunctions.pfnxSysdeviceClose          = xSysdeviceClose,
.tDRVFunctions.pfnxSysdeviceReset          = xSysdeviceReset,
.tDRVFunctions.pfnxSysdeviceGetMBXState     = xSysdeviceGetMBXState,
.tDRVFunctions.pfnxSysdevicePutPacket       = xSysdevicePutPacket,
.tDRVFunctions.pfnxSysdeviceGetPacket       = xSysdeviceGetPacket,
.tDRVFunctions.pfnxSysdeviceDownload        = xSysdeviceDownload,
.tDRVFunctions.pfnxSysdeviceFindFirstFile   = xSysdeviceFindFirstFile,
.tDRVFunctions.pfnxSysdeviceFindNextFile    = xSysdeviceFindNextFile,
.tDRVFunctions.pfnxSysdeviceInfo            = xSysdeviceInfo,
.tDRVFunctions.pfnxChannelOpen              = xChannelOpen,
.tDRVFunctions.pfnxChannelClose             = xChannelClose,
.tDRVFunctions.pfnxChannelDownload          = xChannelDownload,
.tDRVFunctions.pfnxChannelFindFirstFile     = xChannelFindFirstFile,
.tDRVFunctions.pfnxChannelFindNextFile      = xChannelFindNextFile,
.tDRVFunctions.pfnxChannelGetMBXState       = xChannelGetMBXState,
.tDRVFunctions.pfnxChannelPutPacket         = xChannelPutPacket,
.tDRVFunctions.pfnxChannelGetPacket         = xChannelGetPacket,
.tDRVFunctions.pfnxChannelGetSendPacket     = xChannelGetSendPacket,
.tDRVFunctions.pfnxChannelConfigLock        = xChannelConfigLock,
.tDRVFunctions.pfnxChannelReset             = xChannelReset,
.tDRVFunctions.pfnxChannelInfo              = xChannelInfo,
.tDRVFunctions.pfnxChannelWatchdog          = xChannelWatchdog,
.tDRVFunctions.pfnxChannelHostState         = xChannelHostState,
.tDRVFunctions.pfnxChannelBusState          = xChannelBusState,
.tDRVFunctions.pfnxChannelIOInfo            = xChannelIOInfo,
.tDRVFunctions.pfnxChannelIORead            = xChannelIORead,
.tDRVFunctions.pfnxChannelIOWrite           = xChannelIOWrite,
.tDRVFunctions.pfnxChannelIOReadSendData    = xChannelIOReadSendData,
.tDRVFunctions.pfnxChannelControlBlock      = xChannelControlBlock,
.tDRVFunctions.pfnxChannelCommonStatusBlock = xChannelCommonStatusBlock,
.tDRVFunctions.pfnxChannelExtendedStatusBlock = xChannelExtendedStatusBlock,
.tDRVFunctions.pfnxChannelUserBlock         = xChannelUserBlock,
.tDRVFunctions.pfnxChannelPLCMemoryPtr      = xChannelPLCMemoryPtr,
.tDRVFunctions.pfnxChannelPLCIsReadReady    = xChannelPLCIsReadReady,
.tDRVFunctions.pfnxChannelPLCIsWriteReady   = xChannelPLCIsWriteReady,
.tDRVFunctions.pfnxChannelPLCActivateWrite  = xChannelPLCActivateWrite,
.tDRVFunctions.pfnxChannelPLCActivateRead   = xChannelPLCActivateRead,
};

/* Transport layer configuration */
static const TRANSPORT_LAYER_CONFIG_T s_atTransportParams[] =
{
    // number of entries must match entry s_tMarshallerParams.ulTransportCnt
    {
        // rcX packet routing
        .pfnInit = PacketTransportInit,           // transport initialization function
        .pvConfig = (void*) &s_tPacketTransportCfg // transport configuration data
    },
    {
        // cifX API
        .pfnInit = cifXTransportInit,           // transport initialization function
        .pvConfig = (void*) &s_tcifXTransportCfg // transport configuration data
    }
};

/* Marshaller startup parameters */
static const HIL_MARSHALLER_PARAMS_T s_ttMarshallerParams =
{
    // name of server running this Marshaller instance
    .szServerName = " NPLC-M100-DP",
    // maximum number of connectors the Marshaller can handle
    .ulMaxConnectors = MAX_CNT(s_atConnectorParams),
    // number of connectors to initialize at startup (see next entry)
    .ulConnectorCnt = MAX_CNT(s_atConnectorParams),
    // connector initialization parameters (number of entries must match entry above)
    .ptConnectors = s_atConnectorParams,
    // number of transport layers to initialize at startup

```

```
.ulTransportCnt    = MAX_CNT(s_atTransportParams),
// transport initialization parameters (number of entries must match entry above)
.atTransports      = s_atTransportParams
};

/* Static task configuration */
STATIC CONST FAR RX_STATIC_TASK_T FAR atrXStaticTasks[] = {
{
    "MARSHALLER",           // identification string
    TSK_PRIO_11,            // priority
    TSK_TOK_11,             // token ID
    2,                      // task instance (also channel for diagnostics)
    NULL,                   // stack buffer (NULL = set automatically)
    TSK_STACK_SIZE_MARSHALLER, // size of stack buffer
    0,
    RX_TASK_AUTO_START,     // start task automatically
    MarshallerTestTask,     // task entry function
    NULL,                   // function called before task is deleted
    (UINT32)&s_tMarshallerParams, // task startup parameter set
    {0,0,0,0,0,0,0,0}      // reserved area
},
};
```

7.2 Marshaller Error Codes

The following table summarizes the error codes used by the Marshaller main module.

Value	Definition / Description
0xC0000003	HIL_MARSHALLER_E_OUTOFMEMORY Out of memory during heap memory allocation.
0xC0000009	HIL_MARSHALLER_E_INVALIDPARAMETER Invalid parameter or NULL pointer passing.
0xC0000100	HIL_MARSHALLER_E_OUTOFRESOURCES Unable to create, initialize, or access a resource (queue, buffer, etc.).
0xC0000201	HIL_MARSHALLER_E_ALREADYREGISTERED Duplicate transport registration.

Table 26: Error Codes Used by the Marshaller Main Module

The following table summarizes the error codes used by the TCP connector module.

Value	Definition / Description
0xC0860001	TLR_E_TCP_CONNECTOR_MEM_ADDRESS Invalid memory address. (NULL pointer passing.)
0xC0860002	TLR_E_TCP_CONNECTOR_MEMORY Out of memory during heap memory allocation.
0xC0860003	TLR_E_TCP_CONNECTOR_RESOURCE Unable to create, initialize, or access a resource (queue, buffer, etc.).
0xC0860004	TLR_E_TCP_CONNECTOR_PARAMETER_VALUE Invalid parameter value.
0xC0860005	TLR_E_TCP_CONNECTOR_STATE_MACHINE Invalid state. (Results in a reset of the connector's state machine.)
0xC0860006	TLR_E_TCP_CONNECTOR_PROTOCOL_SUPPORT TCP not supported by network protocol stack.
0xC0860007	TLR_E_TCP_CONNECTOR_ADDRESS_INFO Invalid addressing information from network protocol stack.
0xC0860008	TLR_E_TCP_CONNECTOR_SERVICE_CNF Error in service confirmation from network protocol stack.
0xC0860009	TLR_E_TCP_CONNECTOR_SEND Error in service confirmation from network protocol stack.
0xC086000A	TLR_E_TCP_CONNECTOR_MAX_RETRIES Maximum number of service retries exceeded.

Table 27: Error Codes Used by the TCP Connector

8 Appendix

8.1 List of Tables

Table 1: List of Revisions	4
Table 2: Terms, Abbreviations and Definitions	5
Table 3: References	5
Table 4: Marshaller Components	9
Table 5: Connector Configuration Structure	13
Table 6: Transport Layer Configuration Structure	13
Table 7: Marshaller Configuration Structure	14
Table 8: Connector Registration Structure	15
Table 9: Transport Registration Structure	15
Table 10: Marshaller Data Frame Buffer Structure	16
Table 11: TCP Connector Run-time Information Structure	29
Table 12: TCP Connector Extended Task Information Structure	29
Table 13: TCP Connector Environment Information Structure	30
Table 14: TCP Connector Network Information Structure	30
Table 15: TCP Connector Protocol Information Structure	31
Table 16: TCP Connector Latest Error Entry Structure	31
Table 17: TCP Connector State Descriptions	34
Table 18 : Marshaller Directory Structure	35
Table 19 : Operating System Abstraction	36
Table 20: Connector Registration Data	38
Table 21: Transport Registration Data	41
Table 22: rcX Target Connections	43
Table 23: rcX Packet Transport - Configuration Structure	46
Table 24: rcX-based UART Connector - Configuration Structure	46
Table 25: rcX-based TCP Connector - Configuration Structure	47
Table 26: Error Codes Used by the Marshaller Main Module	54
Table 27: Error Codes Used by the TCP Connector	54

8.2 List of Figures

Figure 1: netX Marshaller Structure	10
Figure 2: Sequence Diagram - Marshaller Data Processing	26
Figure 3: Sequence Diagram - Unsolicited Data Processing	27
Figure 4: TCP Connector Diagnostics in SYCON.net	28
Figure 5: TCP Connector State Machine	32
Figure 6: Sequence Diagram - Connector Registration	37
Figure 7: Sequence diagram - Connector Data Reception	38
Figure 8: Sequence diagram - Connector Data Transmission	39
Figure 9: Sequence Diagram - Transport Registration	40
Figure 10: Sequence Diagram - Transport Data Handling	41
Figure 11: rcX Marshaller Integration in a Typical Firmware Scenario	42
Figure 12: rcX Marshaller Task - Structure	44
Figure 13: rcX Packet Transport - Structure	45

8.3 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 Ges.f.Systemaut. mbH
Shanghai Representative Office
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 - 110 025
Phone: +91 11 40515640
E-Mail: info@hilscher.in

Italy

Hilscher Italia srl
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.
Suwon-Si, 443-810
Phone: +82-31-204-6190
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