TCNOpen	Open Source Developement

TRDP

Train Real Time Data Protocol

TRDP User's Manual

Document reference no: TCN-TRDP2-D-BOM-011-21

Author :	A-H. Weiss/B. Löhr
Organisation :	Bombardier Transportation
Document date:	11 September 2013
Revision:	21
Status:	Issued

Dissemination Level		
PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

DOCUMENT SUMMARY SHEET

This is the TRDP User's Manual.

Participants		
Name and Surname	Organisation	Role
Armin-Hagen Weiss	BOM	Lead
Bernd Löhr	Newtec	Participant, Reviewer
Javier Goikoetxea	CAF	Reviewer
Yoshio Sashida	TOS	Reviewer

History			
V1	05 Mar 12	Armin-H. Weiss	Initial version
V2	14 Mar 12	Armin-H. Weiss	Update of chapter: 7, 10, 12, 13, 15
V3	19 Mar 12	Armin-H. Weiss	Changes acc. to CAF review
V4	29 Mar 12	Armin-H. Weiss	Extended and corrected chapters: 10.2.5, 10.8, 13, 15.2 All times given in µs
V5	02 Apr 12	Armin-H. Weiss	Chapter 15.2 XML configuration changed after internal review
V6	16 Apr 12	Armin-H. Weiss	Review, Description for train configuration access and address translation added in chapter 14
V7	03 May 12	Armin-H. Weiss	UUID -> 16*UINT8, chapter 13.3.18 vos_sockAccept interface changed, changes acc. Toshiba review introduction of specific types for listener, publisher, subscriber and application session handles changes acc. CAF review
V8	11 May 12	Armin-H. Weiss	Review chapter 14.2
V9	21 May 12	Armin-H. Weiss	Page numbers updated
V10	05 Jun 2012	Armin-H. Weiss	chapter 10.8 updated, chapter 9.10 updated (tlm_reply split up) changes due to Toshiba and CAF review

V11	11 Sep 2012	Armin-H. Weiss	Changes in chapter 10.2.5.2 due to ladder topology support, Ladder topology specific offset address and subs removed from tlp_xxx functions Types corrected in Table 6 Chapter 10 Configuration Data reworked vos_qsort and vos_bsearch added
V12	22 Oct 2012	Armin-H. Weiss	Error correction in xml structure for configuration Datatype STRING removed and description for UTF16 and CHAR8 strings added tlc_openSession() – interface changed TRDP_STATISTICS_T changed Interface of functions to retrieve statistics changed
V13	20 Dec 2012	Armin-H. Weiss	Interface of tau_readXmlDatasetConfig and tau_initMarshall extended. List of reserved data set id's added. Table 6 Structure TIMEDATE64 - time with microsecond resolution, acc. Posix definition Table 7 Structure TRDP_SEND_PARAM_T – send parameters Table 16 Structure TRDP_MD_INFO_T - received MD telegram information Table 61 Enumeration TRDP_ERR_T – Error code definitions for TRDP
V14	20 Jan 2013	Armin-H. Weiss	vos_xxxTime() functions added in chapter 13.4 chapter 10.1, 10.2, 10.7 updated chapter 14.1 introduced chapter 14.4 updated
V15	07 Feb 2013	Armin-H. Weiss	chapter 14.1 updated old chapter 10.8 Configuration interface removed chapter 9 updated chapter 13.2 updated
V16	15 Mar 2013	Armin-H. Weiss	Figure 3 updated Table 19 updated chapter 7.4.4, 9.10, 17 updated

V17	02 May 2013	Armin-H. Weiss	Figure 3 updated, New tables: Table 108, Table 109, Table 110, Table 111, Table 112, Table 115, Table 116, Table 117 Updated tables regarding memory configuration: Table 20, Table 65, Table 98 New chapters: 13.1.3, 13.1.4, 13.3.3, 13.3.4, 13.3.3, 13.3.6, 13.3.7, 18 Updated chapters: 13.3.15, 16.2, 17
V18	10 June 2013	Armin-H. Weiss	New chapters: 15.2.2, 17.3, 19.1 Updated chapters: 10.1.112.1(new err codes), 13.1.1(new error codes), 13.1.4, 13.2.10, 13.2.11, 13.2.12, 13.3.20 (new return value) , 13.4.20, 16.2 Updated tables: Table 16, Table 36 (MD retries removed) New tables: Table 97, Table 118 Issued
V19	18 June 2013	Armin-H. Weiss	Updated tables: Table 116 Updated chapters: 15.2.4, 15.2.10 (multicast join)
V20	10 Sept 2013	Armin-H. Weiss	XML configuration for data sets updated. Changes according Toshiba review.
V20	11 Sept 2013	Armin-H. Weiss	Data type for “Boolean” unified for all platforms to BOOL8.

Table of Contents

Table of Tables

Table of Figures

1. Introduction

1.1. Purpose

The *Train Realtime Data Protocol (TRDP)* product consists of different modules.

This document describes briefly the architecture and how to use the different modules.

The document covers the function of TRDP from following perspectives:

- Applications written in C
- Applications written in C+
- TRDP configuration
- TRDP statistics
- TRDP simulation

1.2. Intended Audience

This user manual is intended for software programmers, writing programs in C or C++ language, for different operating systems e.g. Linux, Integrity, VxWorks and Windows.

1.3. References / Related Documents

Reference	Number	Title
[Wire]	IEC61375-2-3	TRDP Protocol (Annex A)
[Req]	TCN-TRDP1-D-BOM-003	TRDP System Requirement Specification

Table 1: References

1.4. Abbreviations and Definitions

Abbreviation.	Definition
API	Application Programming Interface
CCU	Central Computing Unit
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
ED	End Device
ETBN	Ethernet Train Backbone Node
FRG	Functional Redundancy Group
HMI	Human Machine Interface
ID	Identifier
IGMP	Internet Group Management Protocol
IP	Internet Protocol
MCG	Mobile Communication Gateway
MD	Message Data
NRTOS	Non-RealTime Operating System
PD	Process Data

Abbreviation.	Definition
QoS	Quality of Service
RTOS	Real Time Operating System
SDT	Safe Data Transmission
SNMP	Simple Network Management Protocol
TCN	Train Communication Network
TOS	Type Of Service
UDP	User Datagram Protocol
UIC	Union Internationale de Chemins de Fer (railway standardisation body)
URI	Universal Resource Identifier
XML	eXtensible Markup Language

Table 2: Abbreviations and Definitions

2. TRDP Architecture

The TRDP component comprises PDCom, MDCom, TRDP Light, VOS (Virtual OS) and Utilities. PDCom handles Process Data and MDCom handles Message Data communication on TCN. TRDP coexists with other users of the network, e.g. streaming communication (like TCP/IP) and communication based on best effort (like UDP/IP).

TRDP consists out of two levels – the TRDP Light and the full TRDP. Both levels are supported by different optional utilities e.g. for marshalling/unmarshalling, reading a TRDP XML configuration, converting IP/URI addresses, safe data transmission support, train topology information access. So TRDP is providing scalability for low end devices using only TRDP Light to high end devices using the full TRDP interface.

Process Data is data that is cyclically distributed among many applications. Payload size is limited to 1436 bytes (without SDT).

Message Data is data that is sent event driven from one application to one or more other applications. Payload using UDP or TCP can be up to 65388 bytes.

TRDP handles all aspects of network communication, e.g. buffering, send/receive, optional marshalling, optional traffic shaping and data integrity.

Applications using TRDP can communicate with each other in a transparent way, within or outside an end device, consist or train.

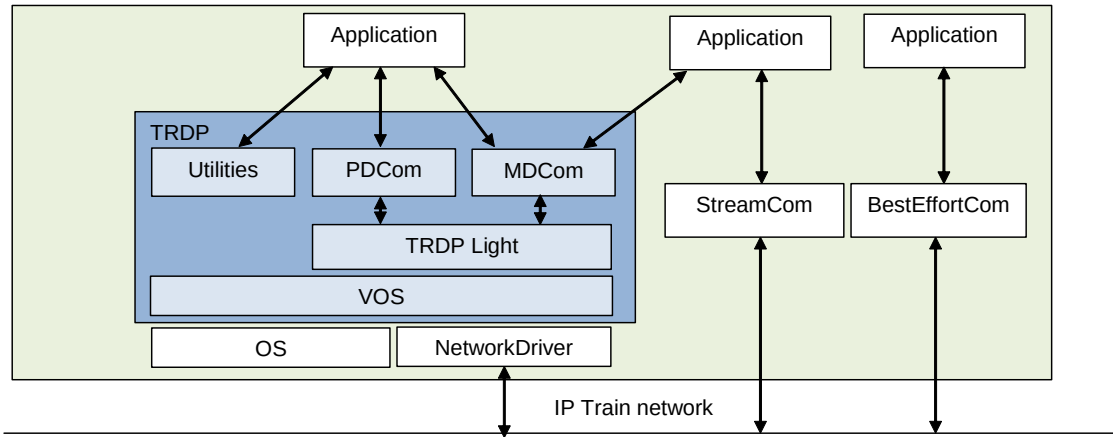

Figure 1 TRDP Architecture – High End Devices

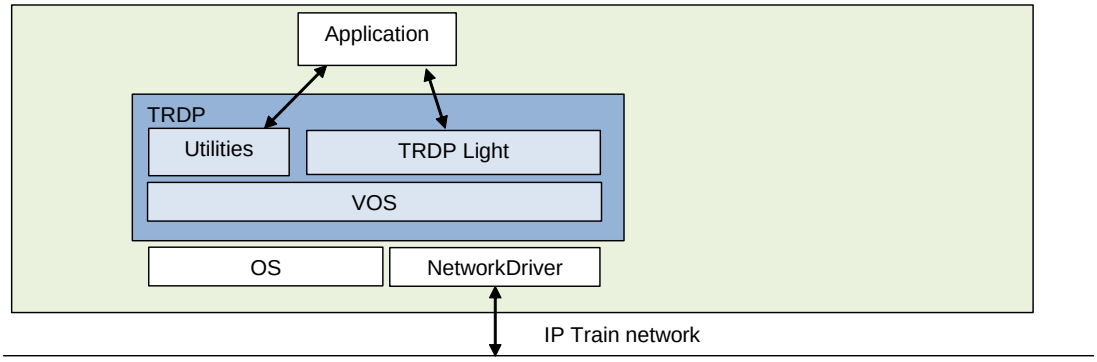

Figure 2 TRDP Architecture – Low End Devices

3. TRDP Interfaces

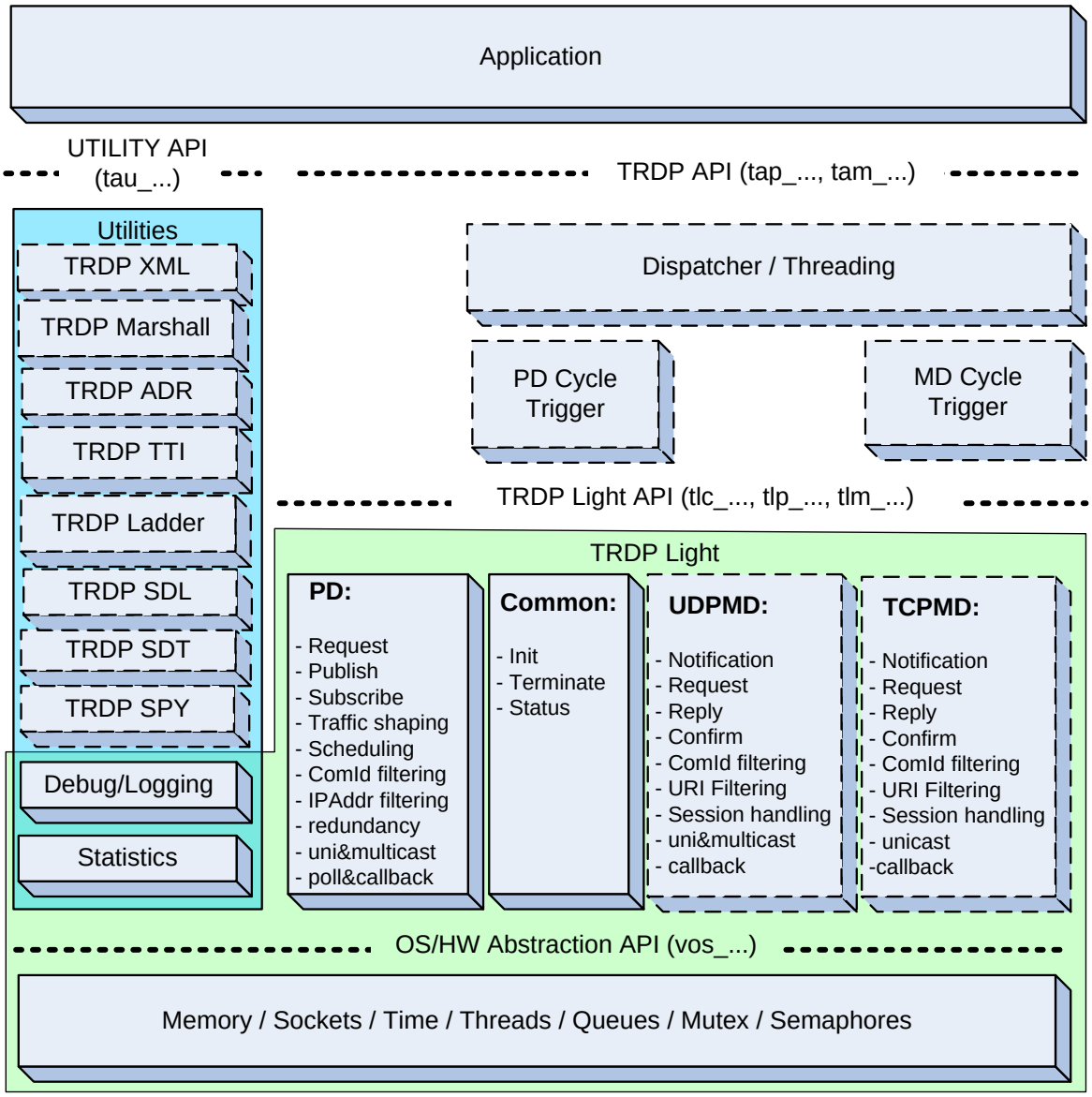

Figure 3 TRDP Interfaces

TRDP API, *application programming interface* is designed as a C interface and as a C++ interface. Both interfaces are functionally equal.

Information about how datasets shall be transmitted, marshalled etc, can be stored in a *Configuration Database*. This is populated by use of configuration file(s), created by an off-line tool, which is analyzed by an *XML parser* or by use of configuration API functions.

TRDP statistic and information data, e.g number of transmitted/ received packets, configuration data etc. can be retrieved via TRDP PD and MD directly from the ED. A MIB browser can be used to retrieve these information via the ETBN.

TRDP offers different API's:

1. The high level application interface (TRDP API) for access to advanced process data communication (tap_…) and message data communication (tam_…)
2. The low level application interface (TRDPLight API) for access to common functionality (tlc_...), process data communication (tlp_…) and the optional UDP/TCP message data communication (tlm_…).
3. The API towards different optional application utility components (tau_...) that can be used as extensions to the communication layer. The following TRDP utilities/services can be used:
   - TRDP SPY – wireshark plugin to interprete the TRDP telegrams
   - TRDP SDT – safe data transmission support according to [wire]
   - TRDP SDL – software download support
   - TRDP TTI – train topology information access
   - TRDP ADR – URI-IP address translation
   - TRDP Marshall – marshalling/unmarshalling for TRDP user data
   - TRDP XML – reading TRDP XML configuration files
   - TRDP Ladder – ladder architecture support
4. The OS and hardware abstraction layer (virtual OS) which provides a standard interface for the OS functions (vos_...) which are used by the TRDP functions internally as well as by the application. This interface ensures that TRDP can be adapted for different OS like Linux, Integrity, VxWorks, Windows without changing the generic TRDP functionality itself. All differences between the OS are completely hided in the virtual OS.

Note:

The two interface levels for message data and process data communication (TRDP API and TRDPLight API) must not be mixed within the application!

4. Network Data Exchange

4.1. Overview

Exchange of data over an IP Network is done according to information stored in a configuration database. The information is pre-configured by an off-line tool and distributed to all end devices.

The primary key for the configuration data is the *communication identity*, ComID.

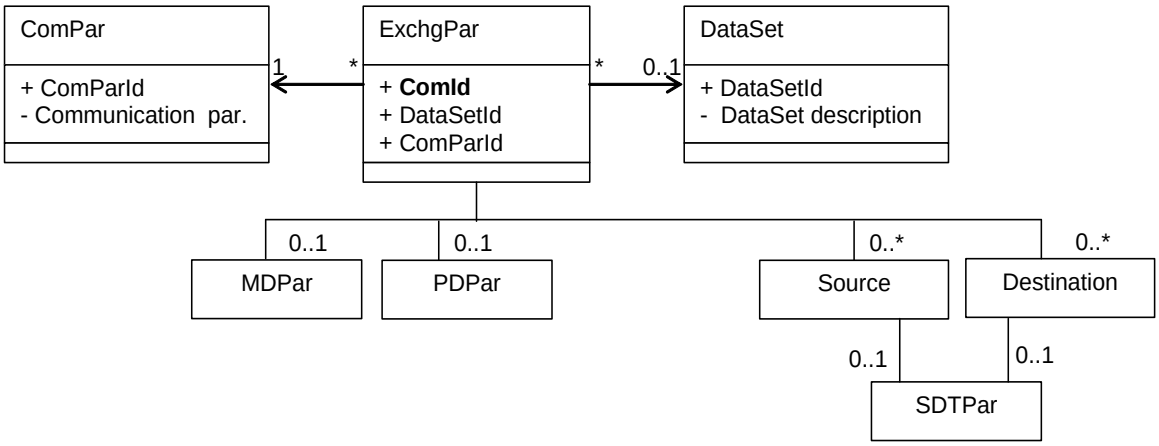

Figure 4 Exchange Parameter Definition

The ComID identifies the *exchange parameters* to be used. This in turn points to which *communication parameters* to use (if not overridden by parameters in the call of API functions), and how the transfer data is structured, the *DataSet.*

4.2. Basic Data Types

To make the TRDP source code compatible, independent on operating system and hardware representation, TRDP uses generic data types:

Type name	Type #	Description
BOOL8	1	=UINT8, 1 bit relevant (equal to zero ➔ false, not equal to zero ➔ true)
CHAR8	2	char, can be used also as UTF-8
UTF16	3	Unicode UTF-16 character
INT8	4	Signed integer, 8 bit
INT16	5	Signed integer, 16 bit
INT32	6	Signed integer, 32 bit
INT64	7	Signed integer, 64 bit
UINT8	8	Unsigned integer, 8 bit
UINT16	9	Unsigned integer, 16 bit
UINT32	10	Unsigned integer, 32 bit
UINT64	11	Unsigned integer, 64 bit
REAL32	12	Floating point real, 32 bit
REAL64	13	Floating point real, 64 bit
TIMEDATE32	14	32 bit UNIX time
TIMEDATE48	15	48 bit TCN time (32 bit seconds and 16 bit ticks)
TIMEDATE64	16	32 bit seconds and 32 bit microseconds

Table 3 Basic Data Types

Note: Strings of CHAR8 and UTF16 can be defined via data sets of variable or fixed size. The strings shall contain the terminating zero. See also chapter 10.5.

Note: For systems using the data types INT64, UINT64, REAL64, TIMEDATE48 or TIMEDATE64, see chapter 16.3.

4.3. DataSet

Data transferred over the TCN must be structured in such a way that it is possible to describe and handle by the marshalling software in a generic way. All data should be packaged into *DataSets*. Exception from this is described in chapter 9.2.10.

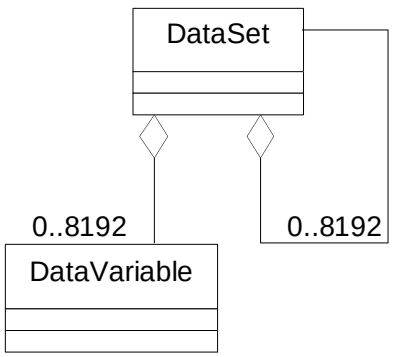

Figure 5 Dataset Definition

Type	Name	Description
TRDP_DATA_TYPE_T	type	Data type (INT8, INT16 ..., see Table 3) or dataset identifier (>1000)
UINT32	size	Number of items or TDRP_VAR_SIZE (0)

Table 4 Structure TRDP_DATASET_ELEMENT_T – Dataset element definition

Type	Name	Description
UINT32	id	dataset identifier (>1000), **Note:** values up to 1000 are reserved
UINT16	reserved1	Must be zero
UINT16	numElement	Number of elements in the following list of elements
TRDP_DATASET_ELEMENT_T	*pElement	Pointer to a dataset element, used as array

Table 5 Structure TRDP_DATASET_T – Dataset definition

A DataSet is a container of data items, like a *structure* in C language. A DataSet can contain up to 8192 DataVariables or other DataSets.

A DataVariable is data of one of the basic data types (UINT8, REAL32 etc.)

Each DataVariable or DataSet can be single or multiple instances. Multiple instances (arrays) can be of fix or variable size.

Note: Process data communication should use only fixed size arrays.

By this data model it is possible to effectively transmit high-speed small, fixed size datasets but also large, dynamically sized datasets.

For more details on DataSets see chapter 10.5.

4.4. Marshalling

All devices connected to the TCN use a standardized set of data types to exchange data over the network.

All end devices do not handle data types internally equally, e.g. Big/Little Endian, and alignment. To handle these differences all devices convert data before transmission so it complies with the standard on the TCN. This is called *marshalling*.

When an end device receives data from the network it converts data from the network data format to its own internal representation. This is called *unmarshalling*.

To use the bandwidth on the network effectively only relevant data is transmitted. This is implemented by *packing* data. E.g. if a flag is represented as a byte in the application programs it can still be packed into one single bit on the network, together with other bits/flags. Packing/unpacking is done by the use of the TRDP utility during marshalling/unmarshalling.

Note: A safe data channel can only start after marshalling and ends before unmarshalling, due to the necessary CRC calculations.

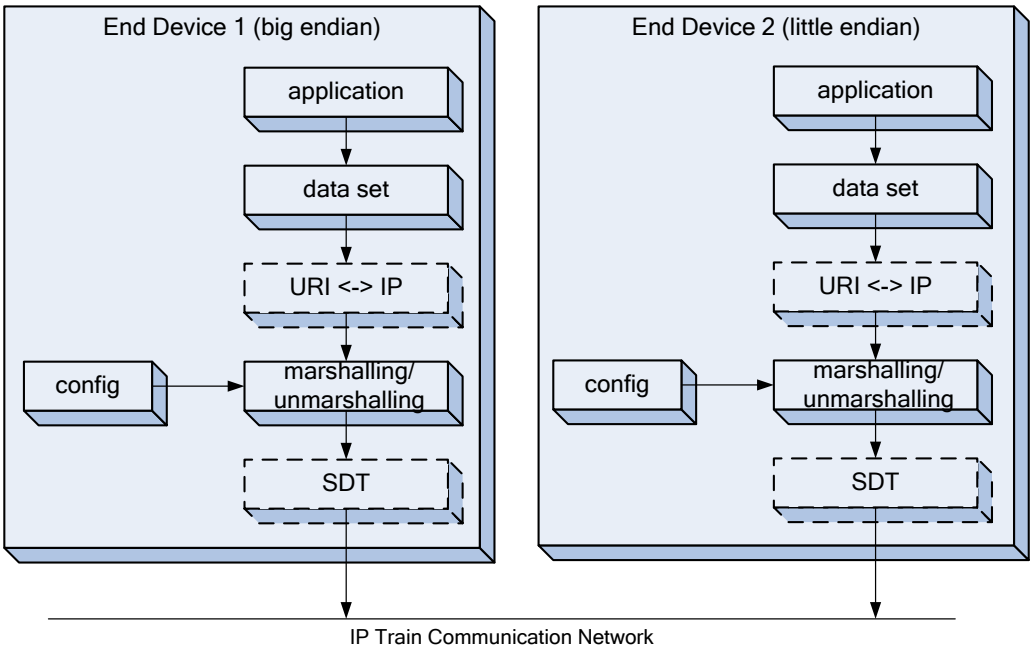

Figure 6 Network Data Exchange

Data that are transmitted on the network are contained in a *dataset*. Datasets are predefined and descriptions are stored in a configuration database created off-line by a tool.

By use of the dataset descriptions the application can use the TRDP utility to perform marshalling and unmarshalling in a generic way.

For more details on marshalling see chapter 11.

5. URI Strings

A URI string consists of a user and a host part (user@host). The host part is a substitute for an IP address and the user part is used for application addressing. Application addressing can only be used for message data. For a detailed description of URI strings and IP addresses see [Wire].

5.1. Device Addressing

TRDP use transmits only the user part of URI's since the host part can be retreieved out of the transmitted IP addresses. The destination IP address is used for sending message to one (unicast) device or to several (multicast) devices. The destination IP address is also used in the receiving end to join multicast addresses.

The destination URI as well as the destination IP address can be configured for the ComId to be sent or can be given in the call of the API functions.

The source IP address is generated automatically in the TRDP.

5.2. Process Data

Each message includes a ComId, see [Wire]

For process data the sending application publishes the ComId and the receiving application subscribes the ComId.

The receiving application can use source filtering to only receive messages from selected devices. The application uses source filter URI strings and IP addresses as filter. The source filter URI strings and IP addresses can be configured for each ComId or can be given in the call of the subscribe method.

5.3. Message Data

Each message includes a ComId, a destination URI string and a source URI string, see [Wire]

The sending application uses one of the MDCom API put methods to send a message. The receiving application adds listener(s) for the ComId(s) and/or for the user part(s) given in the destination URI.

The source URI is not used by TRDP it is only transported to the receiving application.

The user parts given in destination URI and source URI can be configured for each ComId or can be given in the call of the API functions.

6. TRDP Class Diagram

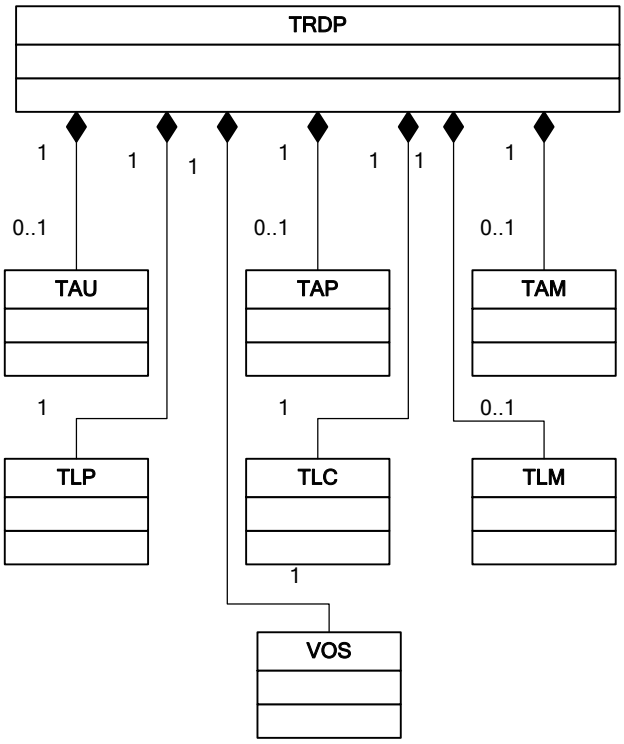

Figure 7 TRDP Class Diagram - Composition

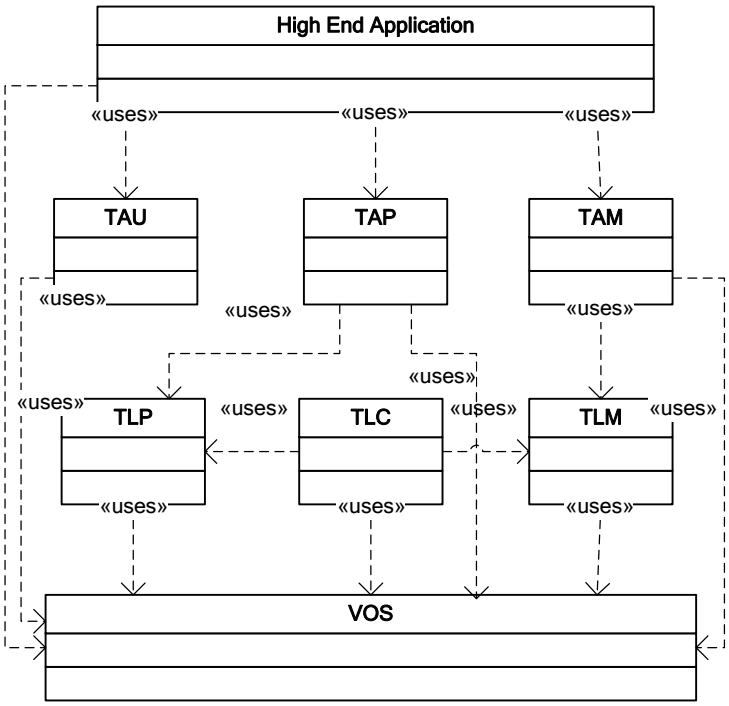

Figure 8 TRDP Class Diagram – High end application relations

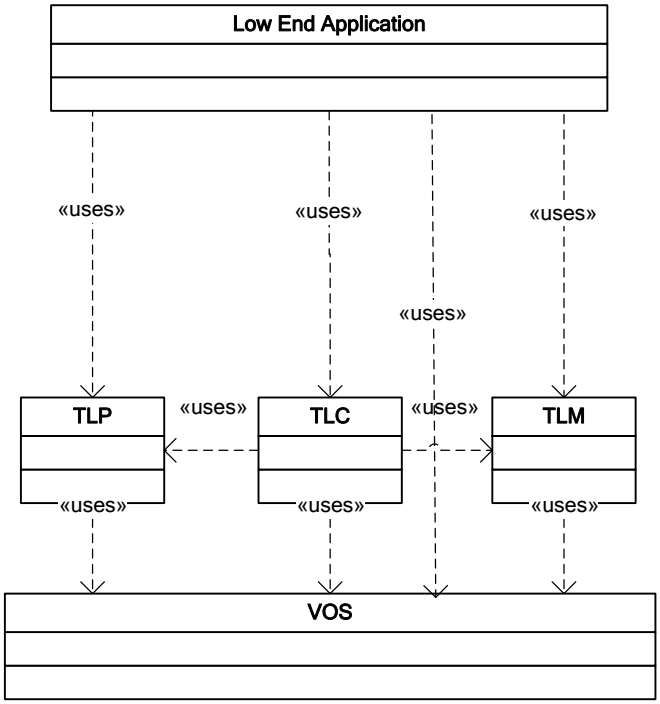

Figure 9 TRDP Class Diagram –Low end application relations

7. How to use TRDP

7.1. Introduction

TRDP provides both a C and a C++ interface. The TRDP C++ component is a container object that comprises the components for process and message data communication as well as the OS/HW abstraction layer and different utilities.

A number of configurations can be done for TRDP either with XML file(s) and/or with API functions, see chapter 10.

There are a number of statistics and information data that can be red from the application via API function or viewed via a MIB browser on a tool PC, see chapter 15.

Redundancy is supported with additional functionality for PD communication. The redundancy handling is described further in each sub chapter, see chapter 8.7 for PD and chapter 9.2.4 for MD.

7.2. Operating System Environment

TRDP can be used on different platforms like e.g Linux, Integrity, VxWorks and Windows. Installation depends on the environment. For more details see chapter 16.

Small systems can use the single process version of TRDP and use only the TRDP Light interface for MD and PD communication.

For more complex systems the TRDP Interface provides the possibility to access the different functions from different processes via shared memory interface.

Note: To use TRDP in a multi process environment there is the limitation to use for PD communication each ComID only in one process or to use different virtual IP addresses for the different processes.

7.3. Start-up of TRDP

7.3.1. Start-up on a single process environment using the TRDP Light interface

The lower layer API allows different modes of using the TRDP stack. The selected mode depends on the architecture and size of the target system and the architecture of the host application. Basically these options can be used:

1. use callbacks or getters
2. use select or polling

For further explanation and examples with PD communication please see chapter 8.9.

7.3.2. Start-up on a multi process environment using the TRDP interface – to be added

7.4. TRDP Start-up - Application Programmers Interface

7.4.1. Common data types

Name	Type	Description
tv_sec	UINT32	seconds
tv_usec	UINT32	microseconds

Table 6 Structure `TIMEDATE64` **- time with microsecond resolution, acc. Posix definition**

Name	Type	Description
qos	UINT8	Quality of service (default should be 5 for PD and 3 for MD)
ttl	UINT8	Time to live (default should be 64)

Table 7 Structure `TRDP_SEND_PARAM_T` **– send parameters**

Value	Name	Description
0	TRDP_FLAGS_DEFAULT	Invalid, use default value
1	TRDP_FLAGS_NONE	No flags set
2	TRDP_FLAGS_MARSHALL	Use internal marshalling/unmarshalling
4	TRDP_FLAGS_CALLBACK	Use the call back function
8	TRDP_FLAGS_TCP	Use TCP instead of UDP – only for MD

Table 8 Enumeration `TRDP_FLAGS_T` **– options for receiving and sending telegrams**

Value	Name	Description
0	TRDP_TO_DEFAULT	Invalid, use default value
1	TRDP_TO_SET_TO_ZERO	Set data to zero if invalid
2	TRDP_TO_KEEP_LAST_VALUE	Keep the last value if invalid

Table 9 Enumeration `TRDP_TO_BEHAVIOUR_T` **- indicates the timeout behaviour of the data**

Value	Name	Description
0	TRDP_RED_UNKNOWN	Redundancy state unknown
1	TRDP_RED_FOLLOWER	Redundancy follower - redundant PD will not be sent out
2	TRDP_RED_LEADER	Redundancy leader - redundant PD will be sent out

Table 10 Enumeration `TRDP_RED_STATE_T` **- indicates the redundancy group state**

Value	Name	Description
0	TRDP_MSG _INV	Invalid
0x5064	TRDP_MSG _PD	Process data
0x5072	TRDP_MSG_PR	Process data request
0x5070	TRDP_MSG_PP	Process data reply
0x5065	TRDP_MSG _PE	Process data error
0x4D6E	TRDP_MSG _MN	Message data notification
0x4D72	TRDP_MSG _MR	Message data request
0x4D70	TRDP_MSG _MP	Message data reply without confirmation request
0x4D71	TRDP_MSG _MQ	Message data reply with confirmation request
0x4D63	TRDP_MSG _MC	Message data confirmation
0x4D65	TRDP_MSG _ME	Message data error

Table 11 Enumeration `TRDP_MSG_T` **- indicates the type of the message**

Value	Name	Description
0	TRDP_REPLY _OK	OK

-1	TRDP_REPLY_APPL_TIMEOUT	Waiting for application response timed out
-2	TRDP_REPLY_SESSION_ABORT	Session abort
-3	TRDP_REPLY_NO_REPLIER	No Listener / Destination unknown
-4	TRDP_REPLY_NO_MEM_REPLIER	Buffer/memory not available at replier side
-5	TRDP_REPLY_NO_MEM_LOCAL	Buffer/memory not available at caller side
-6	TRDP_REPLY_NO_REPLY	No reply received
-7	TRDP_REPLY_NOT_ALL_REPLIES	Not all expected replies received
-8	TRDP_REPLY_NO_CONFIRM	No confirm received
-9	TRDP_REPLY _WRONG_TOPO_COUNT	Wrong topo count
-10	TRDP_REPLY_SENDING_FAILED	Sending failed
-99	TRDP_REPLY _UNSPECIFIED_ERROR	Unspecified error

Table 12 Enumeration `TRDP_REPLY_STATUS_T` **- indicates the result of the transmission**

Value	Name	Description
0	TRDP_LOG_ERROR	This is a critical error
1	TRDP_LOG_WARNING	This is a warning
2	TRDP_LOG_INFO	This is a information

Table 13 Enumeration `TRDP_LOG_T` **- indicates the log type of an error message**

Type	Name	Description
void *	TRDP_APP_T	Handle for a TRDP application session

Table 14 Type `TRDP_APP_T` **– application session handle**

7.4.2. TRDP PD Message Information

The TRDP PD message information structure is defined as data type `TRDP_PD_INFO_T`. It contains following information about the received PD message.

Type	Name	Description
`TRDP_IP_ADDR_T`	`srcIpAddr`	Source IP address **For MsgType = Error**: The value is the source IP address of the sent message for which the result is reported. **else:** The value is the source IP address of the sender device.
`TRDP_IP_ADDR_T`	`destIpAddr`	Destination IP address **For MsgType = Error**: The value is the destination IP address of the sent message for which the result is reported. **else:** The value is the destination IP address of the sender device.
`UINT32`	`seqCount`	The sequence counter for sending process datagrams is managed per ComId/MsgType at each requester/publisher. The sequence counter for received process datagrams is managed(stored) per SourceIPAddr/ComId/MsgType at each publisher/subscriber. Counter is incremented with each sending of the process datagram. Datagrams sent in parallel via different subnets

		are sent with the same sequence counter to detect duplication at receiver side. So the sequence counter can be used for the surveillance if the communication layer is still sending the PD. A surveillance if the application is still uptating the PD can be done via the safe data transmission protocol or needs to be implemented by the application (e.g. life sign).
UINT16	protVersion	The version of the protocol. Higher significant octet: Version Lower significant octet: Release The version will be incremented for incompatible changes, the release for compatible changes EXAMPLE – 0x0102 = protocol version 1.2
TRDP_MSG_T	msgType	Message type, see chapter 7.4.1
UINT32	comId	Communication identifier **For MsgType = Request or Reply:** Identifier of the user data set for the received message. **For MsgType = Error**: The value is ComId of the sent message for which the result is reported.
UINT32	topoCount	Topo counter value. **For MsgType = Request or Reply:** Topo counter value of a received message. **For MsgType = Error:** The value is set to zero.
UINT32	replyComId	**For MsgType = PD Request:** The requested ComId, if set to 0, ComId is used for the reply. **For MsgType = PD Reply or Error:** The value is set to zero
TRDP_IP_ADDR_T	replyIpAddr	**For MsgType = Request:** The requested reply address, if set to 0, source IP address is used for the reply. **For MsgType = Reply or Error:** The value is set to zero
void	*pUserRef	Reference value given with the local subscription.
TRDP_ERR_T	resultCode	Result code **For MsgType = Request or Reply:** The value is set to OK **For MsgType = Error**: See chapter 7.4.1

Table 15 Structure TRDP_PD_INFO_T **– received PD telefram information**

7.4.3. TRDP MD Message Information

The TRDP MD message information structure is defined as data type `TRDP_MD_INFO_T`. It contains following information about the received MD message.

Type	Name	Description
`TRDP_IP_ADDR_T`	`srcIpAddr`	Source IP address **For MsgType = Error**: The value is the source IP address of the sent message for which the result is reported. **Else:** The value is the source IP address of the sender device.
`TRDP_IP_ADDR_T`	`destIpAddr`	Destination IP address **For MsgType = Error**: The value is the destination IP address of the sent message for which the result is reported. **Else:** The value is the destination IP address of the sender device.
`UINT32`	`seqCount`	Counter incremented with each repetition of the request message. The counter value shall be returned with the reply message. Start value: 0
`UINT16`	`protVersion`	The protocol version of the protocol. Higher significant octet: Version Lower significant octet: Release The version will be incremented for incompatible changes, the release for compatible changes EXAMPLE – 0x0102 = protocol version 1.2
`TRDP_MSG_T`	`msgType`	Message type, see chapter 7.4.1
`UINT32`	`comId`	Communication identifier **For MsgType = Request or Reply:** Identifier of the user data set for the received message. **For MsgType = Error**: The value is ComId of the sent message for which the result is reported.
`UINT32`	`topoCount`	Topo counter value. **For MsgType = Request or Reply:** Topo counter value of a received message. **For MsgType = Error**: The value is set to zero.
`BOOL8`	`aboutToDie`	Session is about to die
`UINT32`	`numRepliesQuery`	number of ReplyQuery received, used to count number of expected Confirm sent
`UINT32`	`numConfirmSent`	number of Confirm sent
`UINT32`	`numConfirmTimeout`	number of Confirm Timeouts (incremented by listeners
`UINT16`	`userStatus`	**For MsgType = MD Reply or Confirm:** The header offers the possibility to transfer an application defined error code. 0 shall be used for

		normal operation.
TRDP_REPLY_STATUS_T	replyStatus	**For MsgType = MD Reply, Confirm or Error:** **Reply status in case of errors in TRDP layer**
TRDP_UUID_T	sessionId	16 byte Session identification. Used by MDCom to connect reply messages with request message. **For MsgType = Request:** The value of the received message.The application shall use the value received in the request message when sending the reply message**.** **For MsgType = MD Notification, Reply or Confirm:** The value of the received message. **For MsgType = Error**: The value is set to the value used by TRDP
UINT32	replyTimeout	The reply timeout in µs value used in a request / reply session.
TRDP_UR,I_USER_T	destURI	User part of destination URI sent in MD header
TRDP_URI_USER_T	srcURI	User part of source URI sent in MD header
UINT32	numExpReplies	Number of expected replies for the request
UINT32	numReplies	Number of received replies for the request
void	*pUserRef	**For MsgType = MD Notification or Request:** At the receiving side the value is the same as the application used when adding a listener. This value can be used by the application to associate a received message with the corresponding listener added before. **For MsgType = MD Reply or Error:** At receiving side the value is the same as the application used when sending the MD..This value can be used by the application to associate a response or a communication result with the corresponding MD send.
TRDP_ERR_T	resultCode	Result code **For MsgType = MD Notification, Request or Confirm:** The value is set to OK **For MsgType = MD Reply:** See chapter 7.4.1 **For MsgType = Error**: See chapter 7.4.1

Table 16 Structure TRDP_MD_INFO_T **- received MD telegram information**

7.4.4. Callback Functions

Callback functions can be used by the application to get responses and communication results and in the receiving end to get the received message.

The callback function should have the following syntax.

Note: Use the callback function with care. The function shall be non-blocking and shall return as fast as possible as it is called from the TRDP receiver and sender threads, i.e.no further message can be treated until the callback function returns.

Note: Even if configured, the data given to the callback functions are NOT unmarshalled to prevent additional copying. Thus using callback mechanism, unmarshalling needs to be done in the application.

Name	TRDP_PD_CALLBACK_T	

Synopsis C	`typedef void (*TRDP_PD_CALLBACK_T) (`
	`  void  *pRefCon,`
	`  TRDP_APP_SESSION_T  appHandle,`
	`  const TRDP_PD_INFO_T *pPdInfo,`
	`  const char  *pData,`
	`  UINT32  dataLength);`	

Synopsis C++		
Abstract	Application function used to get received message and/or communication result for a sending. The callback function can be used to: • Get received PD messages • Get communication result for a message	
Parameters	`pRefCon`	Context reference given at initialization time
	`appHandle`	Application handle returned by tlc_openSession. To be used for subsequent TRDP function calls in the callback routine.
	`pPdInfo`	Pointer to PD message Information. See sec 7.4.2
	`pData`	Pointer to data buffer with received data. **Note:** The data in the buffer will not be available after function return, i.e. data has to be used within the function or has to be copied.
	`dataLength`	Length of data in the data buffer.
Returns	–	

Name	TRDP_MD_CALLBACK_T	

Synopsis C	`typedef void (*TRDP_MD_CALLBACK_T) (`
	`  void  *pRefCon,`
	`  TRDP_APP_SESSION_T  appHandle,`
	`  const TRDP_MD_INFO_T *pMdInfo,`
	`  const char  *pData,`
	`  UINT32  dataLength);`	

Synopsis C++		
Abstract	Application function used to get received message and/or communication result for a sending. The callback function can be used to: • Get received messages for added listeners • Get received response messages and/or communication result for a sent request message • Get communication result for a sent data or response message	
Parameters	`pRefCon`	Context reference given at initialization time
	`appHandle`	Application handle returned by tlc_openSession. To be used for subsequent TRDP function calls in the callback routine.
	`pMdInfo`	Pointer to MD message Information. See sec 7.4.2
	`pData`	Pointer to data buffer with received data. **Note:** The data in the buffer will not be available after function return, i.e. data has to be used within the function or has to be copied.
	`dataLength`	Length of data in the data buffer.
Returns	–	

Name	TRDP_MARSHALL_T	

Synopsis C	`typedef void (*TRDP_MARSHALL_T) (`
	`  void  *pRefCon,`
	`  UINT32  comId`
	`  const UINT8  *pSrc,`
	`  UINT8  *pDst,`
	`  UINT32  *pDstSize);`	

Synopsis C++		
Abstract	Type for marshalling callback function used in the initialization of TRDP.	
Parameters	`pRefCon`	Context reference given at initialization time
	`comId`	ComId of the telegram
	`pSrc`	Pointer to message to be marshaled
	`pDst`	Pointer to data buffer with marshalled message
	`pDstSize`	Pointer to length of marshalled data
Returns	–	

Name	TRDP_UNMARSHALL_T	

Synopsis C	`typedef void (*TRDP_UNMARSHALL_T) (`
	`  void  *pRefCon,`
	`  UINT32  comId`
	`  const UINT8  *pSrc,`
	`  UINT8  *pDst,`
	`  UINT32  *pDstSize);`	

Synopsis C++	`typedef void (*TRDP_UNMARSHALL_T) (`
	`  void  *pRefCon,`
	`  UINT32  comId`
	`  const UINT8  *pSrc,`
	`  UINT8  *pDst,`
	`  UINT32  *pDstSize);`	

Abstract	Type for unmarshalling callback function used in the initialization of TRDP.	
Parameters	`pRefCon`	Context reference given at initialization time
	`comId`	ComId of the telegram
	`pSrc`	Pointer to message to be unmarshalled
	`pDst`	Pointer to data buffer with unmarshalled message
	`pDstSize`	Pointer to length of unmarshalled data
Returns	-	

7.4.5. tlc_freeBuf

Name	tlc_freeBuf	

Synopsis C	`TRDP_ERR_T tlc_freeBuf(`
	`  TRDP_APP_T  appHandle,`
	`  char  *pBuf);`	

Synopsis C++	`TRDP_ERR_T tlc::freeBuf(` `  char  *pBuf);`	
Abstract	De-allocate buffer memory. This method is used when the application is using a buffer, for received message or communication result, allocated by the TRDP.	
Parameters	`appHandle`	(C) Handle returned by tlc_openSession()
	`pBuf`	Pointer to data buffer.
Returns C/C++	0 if OK. !=0 if error, see chapter 12.1. (No C++ exception is thrown.)	

7.4.6. tlc_init

Name	***tlc_init***	

Synopsis C	`TRDP_ERR_T tlc_init(`
	`  TRDP_PRINT_DBG_T  pPrintDebugString,`
	`  const TRDP_MEM_CONFIG_T  *pMemConfig);`	

Synopsis C++	`TRDP_ERR_T tlc::init(`
	`  TRDP_PRINT_DBG_T  pPrintDebugString,`
	`  const TRDP_MEM_CONFIG_T  *pMemConfig);`	

Abstract	Initializes TRDP base and VOS.	
	`pPrintDebugString`	Pointer to function to print debug strings. The function type reuses VOS_PRINT_DBG_T (see chapter 13.1.2).
	`pMemConfig`	Pointer to memory configuration
Returns C	0 if OK, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_PARAM_ERR initialization error	
Returns C++	“TRDPException” according to chapter 12.1.	

7.4.7. tlc_openSession

Name	***tlc_openSession***	

Synopsis C	`TRDP_ERR_T tlc_openSession(`
	`  TRDP_APP_T  *pAppHandle,`
	`  TRDP_IP_ADDR_T  ownIpAddr,`
	`  TRDP_IP_ADDR_T  leaderIpAddr,`
	`  const TRDP_MARSHALL_CONFIG_T *pMarshall,`
	`  const TRDP_PD_CONFIG_T  *pPdDefault,`
	`  const TRDP_MD_CONFIG_T  *pMdDefault,`
	`  const TRDP_PROCESS_CONFIG  *pProcessConfig);`	

Synopsis C++	`TRDP_ERR_T tlc::openSession(`
	`  TRDP_IP_ADDR_T  ownIpAddr,`
	`  TRDP_IP_ADDR_T  leaderIpAddr,`
	`  const TRDP_MARSHALL_CONFIG_T *pMarshall,`
	`  const TRDP_PD_CONFIG_T  *pPdDefault,`
	`  const TRDP_MD_CONFIG_T  *pMdDefault,`
	`  const TRDP_PROCESS_CONFIG  *pProcessConfig);`	

Abstract	Opens and initializes a TRDP session.	
Parameters	`pAppHandle`	(C) A handle for further calls to the trdp stack necessary for multiprocessing systems
	`ownIpAddr`	Own IP address, can be different for each process in multiprocessing systems
	`leaderIpAddr`	IP address of redundancy leader (depending on redundancy concept)
	`pMarshall`	Pointer to marshalling configuration
	`pPdDefault`	Pointer to default PD send parameters
	`pMdDefault`	Pointer to default MD send parameters
	`pProcessConfig`	Pointer to process configuration only option parameter is used here to define session behavior, all other parameters are only used to feed statistics
Returns C	0 if OK, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_PARAM_ERR initialization error TRDP_SOCK_ERR socket error	
Returns C++	“TRDPException” according to chapter 12.1.	

Type	**Name**	**Description**
TRDP_MARSHALL_T	pfCbMarshall	Pointer to marshall callback function
TRDP_UNMARSHALL_T	pfCbUnmarshall	Pointer to unmarshall callback function
void	pRefCon	Pointer to user context for call back

Table 17 Structure `TRDP_MARSHALL_CONFIG_T` **- marshalling/unmarshalling configuration**

Type	Name	Description
TRDP_PD_CALLBACK_T	pfCbFunction	Pointer to PD callback function
void	pRefCon	Pointer to user context for call back
TRDP_SEND_PARAM_T	pSendParam	Pointer to default send parameters
TRDP_FLAGS_T	flags	Default flags for PD packets
UINT32	timeout	Default timeout in µs
TRDP_TO_BEHAVIOR_T	toBehavior	Default timeout behaviour
UINT32	port	Port to be used for PD communication.

Table 18 Structure `TRDP_PD_CONFIG_T` **- default PD configuration**

Type	Name	Description
TRDP_MD_CALLBACK_T	pfCbFunction	Pointer to MD callback function
void	pRefCon	Pointer to user context for call back
TRDP_SEND_PARAM_T	pSendParam	Pointer to default send parameters
TRDP_FLAGS_T	flags	Default flags for MD packets
UINT32	replyTimeout	Default reply timeout in µs
UINT32	confirmTimeout	Default confirmation timeout in µs
UINT32	connectTimeout	TCP connection timeout in µs. If the connection is not used for the specified time, it shall be closed.
UINT32	udpPort	Port to be used for UDP MD communication.
UINT32	tcpPort	Port to be used for TCP MD communication.
UINT32	maxNumSessions	Maximum number of replier sessions to prevent DoS attacks

Table 19 Structure `TRDP_MD_CONFIG_T` **- default MD configuration**

Type	Name	Description
CHAR8	*p	Pointer to memory block
UINT32	size	Size of memory block given with p
UINT32	prealloc[VOS_MEM_NBLOCKSIZES]	Configuration of memory block

Table 20 Structure `TRDP_MEM_CONFIG_T` **– indicates the memory configuration**

Value	Name	Description
0x00	TRDP_OPTION_NONE	Invalid, use default value
0x01	TRDP_OPTION_NON_BLOCK	Use non blocking I/O calls, polling necessary
0x02	TRDP_OPTION_TRAFFIC_SHAPING	Do traffic shaping

Table 21 Structure `TRDP_OPTION_T` **– main options for the TRDP process**

7.4.8. tlc_reinitSession

Name	tlc_reinitSession	
Synopsis C	`TRDP_ERR_T tlc_reinitSession(` `  TRDP_APP_T appHandle);`	
Synopsis C++	`TRDP_ERR_T tlc::reinitSession(void);`	
Abstract	Reinitializes a TRDP light session. Shall be called by the application when a link-down/link-up event has occurred during normal operation to re-join the multicast groups.	
Parameters	`appHandle`	(C) handle returned by tlc_openSession()
Returns C	0 if OK, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_NOINIT_ERR handle invalid	
Returns C++	0 if OK, if error exception thrown according to chapter 12.1.	

7.4.9. tlc_closeSession

Name	tlc_closeSession	
Synopsis C	`TRDP_ERR_T tlc_closeSession(` `  TRDP_APP_T appHandle);`	
Synopsis C++	`TRDP_ERR_T tlc::closeSession(void);`	
Abstract	Closes a TRDP light session.	
Parameters	`Handle`	(C) handle returned by tlc_openSession()
Returns C	0 if OK, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_NOINIT_ERR handle invalid	
Returns C++	0 if OK, if error exception thrown according to chapter 12.1.	

7.4.10. tlc_terminate

Name	tlc_terminate	
Synopsis C	`TDRP_ERR_T tlc_terminate(void);`	
Synopsis C++	`TDRP_ERR_T tlc::terminate(void);`	
Abstract	Close all TRDP light sessions and clean up. Mainly used for debugging/test runs	
Parameters	`none`	
Returns	0 if OK, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error	
Returns C++	0 if OK, if error exception thrown according to chapter 12.1.	

7.4.11. tlc_getInterval

Name	tlc_getInterval	

Synopsis C	`TDRP_ERR_T tlc_getInterval(`
	`  TRDP_APP_T  appHandle,`
	`  TRDP_TIME_T  *pInterval,`
	`  TRDP_FDS_T  *pFileDesc,`
	`  INT32  *pNoDesc);`	

Synopsis C++	`TDRP_ERR_T tlc::getInterval (`
	`  TRDP_TIME_T  *pInterval,`
	`  TRDP_FDS_T  *pFileDesc,`
	`  INT32  *pNoDesc );`	

Abstract	Get the lowest time interval for PD's. Return the maximum time interval suitable for 'select()' so that we can send due PD packets in time. If the PD send queue is empty, return zero time.	
Parameters	`appHandle`	(C) handle returned by tlc_openSession()
	`pInterval`	The maximum time interval suitable for 'select()' so that we can send due PD packets in time. If the PD send queue is empty, return NULL
	`pFileDesc`	Pointer to file descriptor set
	`pNoDesc`	Pointer to put number of used descriptors (for select())
Returns	0 if OK, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_NOINIT_ERR handle invalid	
Returns C++	0 if OK, if error exception thrown according to chapter 12.1.	

7.4.12. tlc_setTopoCount

Name	tlc_setTopoCount	
Synopsis C	`void tlc_setTopoCount(UINT32 topoCount);`	
Synopsis C++	`void tlc::setTopoCount(UINT32 topoCount);`	
Abstract	The given topo count value is used for validating outgoing and incoming packets	
Parameters	`topoCount`	New topo count value.
Returns C / C++	-	

7.4.13. tlc_process

Name	**tlc_process**	

Synopsis C	`TDRP_ERR_T tlc_process(`
	`  TRDP_APP_T  appHandle,`
	`  TRDP_FDS_T  *pRfds,`
	`  INT32  *pCount);`	

Synopsis C++	`TDRP_ERR_T tlc::process(`
	`  TRDP_FDS_T  *pRfds,`
	`  INT32  *pCount);`	

Abstract	Work loop of the TRDP handler. Search the queue for pending PD's to send. Search the receive queue for pending PD's (time out)	
Parameters	`appHandle`	(C) handle returned by tlc_openSession()
	`pRfds`	Pointer to set of ready descriptors
	`pCount`	Pointer to number of ready descriptors
Returns C	0 if OK, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_NOINIT_ERR handle invalid	
Returns C++	0 if OK, if error exception thrown according to chapter 12.1.	

Type	**Name**	**Description**
UINT32	fd_count	Number of file descriptors
INT32	fds_bits[MAX_SOCKET_CNT]	File descriptors

Table 22 Structure `TRDP_FDS_T` - file descriptor set compatible with fd_set() / select()

8. Process Data Communication

8.1. Publish and Subscribe mechanism

Exchange of data is done by use of *publish* and *subscribe* mechanisms.

Applications that want to receive a ComId calls TRDP to be registered as a subscriber. There can be many subscribers for a ComId. One application can subscribe for several comId´s.

An application that is the source of a ComId calls TRDP to be registered as publisher of that ComId.

If there is more than one publisher in different devices of the same ComId addressed to the same device the subscription has to be done with source filtering.

8.2. Communication patterns

Process data communication supports the pull and the push pattern.

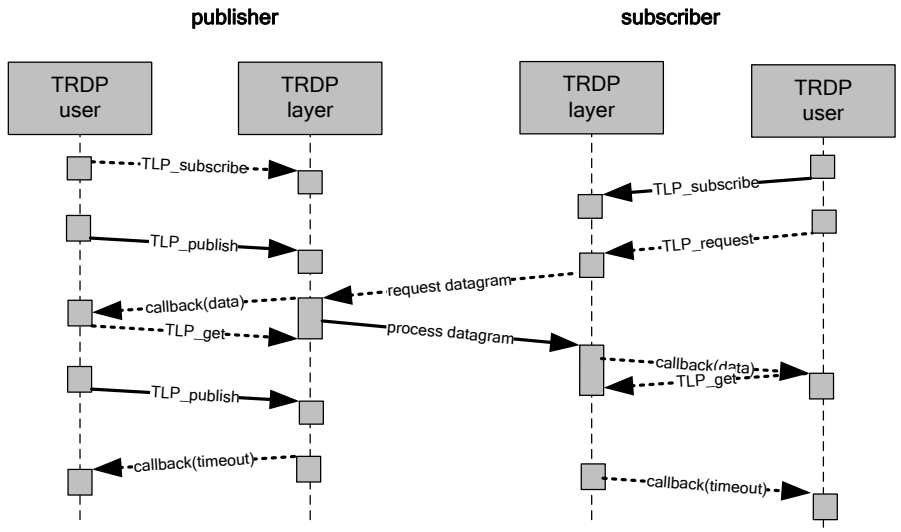

Figure 10 PD Pull Pattern

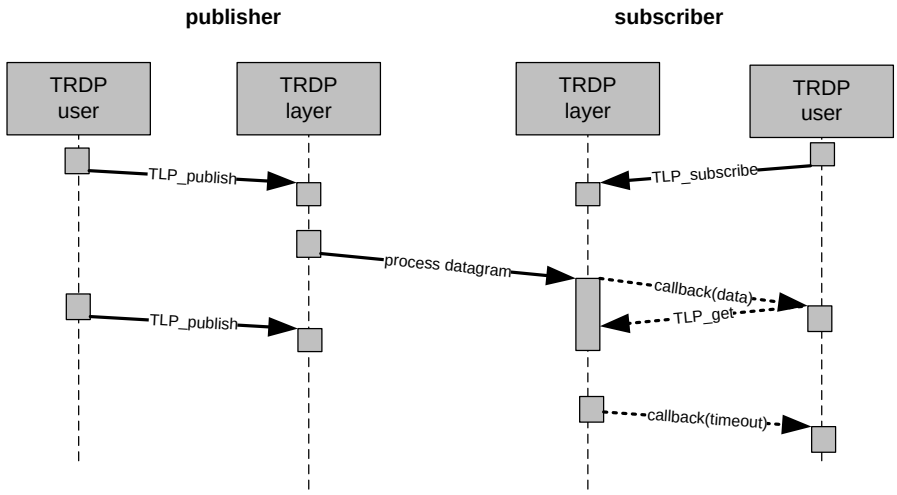

Figure 11 PD Push Pattern

In the push pattern the PD are published in fixed cycle to a given destination address (range). Any application using this address or within this address range can receive the data after a successful subscription

In the pull pattern PD are only sent after receiving a request telegram. The request telegram itself contains the requested ComId and can contain also a destination address (range). It can also contain data. The data can be received if a subscription matching to the telegram was done. If no destination address (range) is given, the source IP address of the received request telegram will be taken as destination address.

8.3. Schedule Group Identity (not in the low layer)

It is crucial that all applications running in a schedule group accesses the same local copy of data. When calling API functions the applications must identify themselves by a *schedule group identity*.

This identity can be any unsigned integer number, but it must be unique for each group within the device. The same number can be used for both subscription and publishing.

Note: A schedule group shall only be used by one thread otherwise data integrity can't be guaranteed.

Tip: To avoid collisions make shure to get unique numbers for the schedule groups within the device.

It is recommended that all PD within the same thread/task, is using the same schedule group identity.

8.4. Marshalling and Unmarshalling

With publish and subscribe can be chosen the optional marshalling/unmarshalling function given in the initialization of the TRDP stack. Choosing this option the PD sent out will be marshalled automatically as well as the received PD will be unmarshalled automatically.

Note: Since a safe data channel can only start after marshalling and ends before unmarshalling, due to the necessary CRC calculations, this option can be used only for unsafe channels (see also chapter 4.4).

Note: Data given to the callback functions are NOT unmarshalled to prevent additional copying. Thus, required unmarshalling needs to be done in the application.

8.5. Callback functionality

For synchronization purposes it might be useful to get one or more PD via call back function. This can be chosen in the related subscription. When receiving matching PD or related timeouts, in the tlc_openSession() given process data call back function will be called.

Note: Data given to the callback functions are NOT unmarshalled to prevent additional copying. Thus, an required unmarshalling needs to be done in the application.

8.6. Validity

When data is copied from the net buffer to the schedule group buffer, by use of the `sink` method, a check is made of the timestamp. If it is older than a specified time it is considered *invalid*. Only data for ComId's that are configured with a time-out time is checked.

Depending on configuration, invalid data are handled in following way:

- replaced by zeroes
- kept as is

8.7. Redundancy

In a redundant system two or more devices can perform the same task. At one single moment one device can take the role as *leader*, and the other ones as *follower*. If the leader becomes defect one follower can take over the role and become the leader. This can also be used for master/slave functionality.

In the configuration data it is possible to specify that a ComId for process data should be handled as redundant data by setting the value greater than zero for the "redundant" attribute of the ComId "pd-parameter" tag. The redundant ComId's are divided into redundancy function groups controlled by the value of the "redundant" attribute.

A ComId marked as redundant will be updated and distributed if the corresponding redundancy function group is leader, but not if it is follower.

To change redundant role there is two methods `tlp_setRedundant` and `tlp_getRedundant` which may be called by any application.

The method `tlp_setRedundant` changes the role (leader/follower) for the specified redundancy function group(s). The method `tlp_getRedundant` reads back the status of the given redundancy groups.

Subscribers of a redundant ComId receive data independent of which device is the leader if no source filter is used. If source filter is used all redundant devices have to be included in the "source-uri" used to resolve the source filter IP address, see chapter 8.10.9 and 10.2.5. Up to two devices can be defined for the "source-uri".

8.8. Train inauguration

After a train inauguration the IP addresses for ETB traffic will be changed, e.g. all IP addresses including the topo counter value. For further details see [Wire].

If the IP address used for sending PD message is changed after an inauguration the publishing has to be renewed. The renewing of publishing of a ComId can be done by unpublish and then publish the ComId.

If the IP address used for filtering received messages or for joining a multicast address is changed after an inauguration the subscription has to be renewed. The renewing of subscription of a ComId can be done by unsubscribe and then subscribe the ComId.

Note: The communicating devices may be changed when a renewing of publishing and subscription is done with the same URI strings before and after an inauguration.

8.9. Use Cases

The use case in the figure below is an example where two applications are using the TLP API as interface to their communication component.

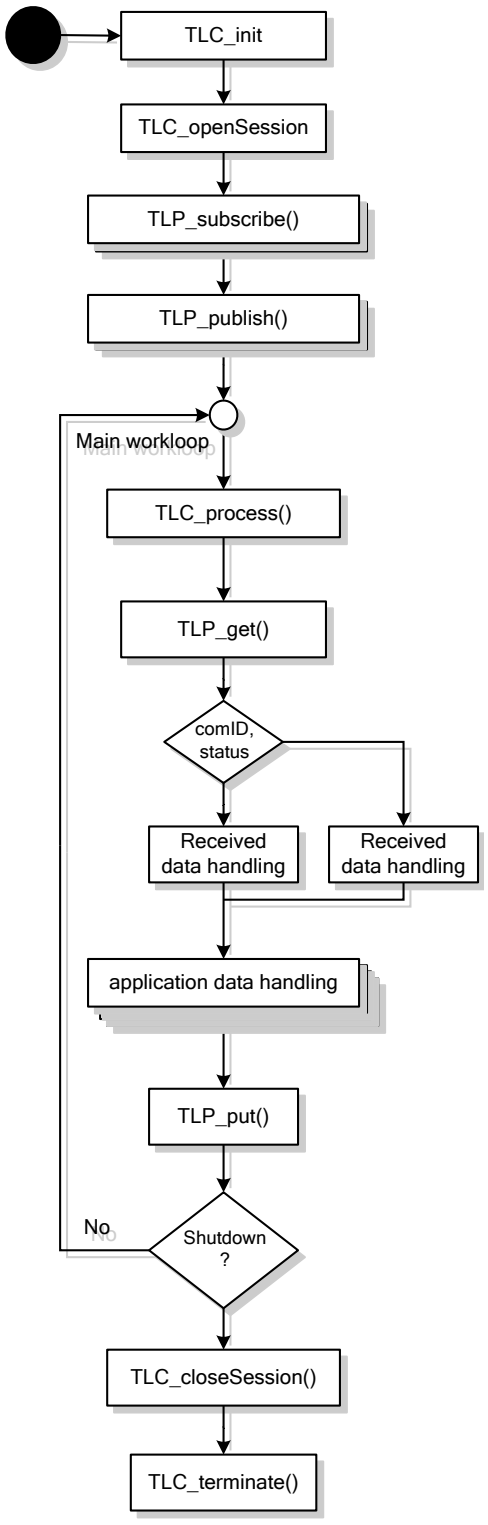

Figure 12 Single Tread PD Polling Workflow

Figure 12 shows a sample sequence of a polling application. After initializing the library including memory management and debug messages with tlc_init() a trdp session related to a defined IP address will be opened. For single thread polling mechanism the session is opened (tlc_openSession()) with the option TRDP_OPTION_NON_BLOCK and not providing callback functions. The application registers telegrams by subscribing and publishing its comId's. These functions return a handle to reference to these elements on successive calls, i. e. get/put or unpublish/unsubscribe.

Within the application's main loop, tlc_process() must be called. tlc_process() will check for any process data pending to be sent and to be received. It will set error states in case packets are overdue (timed out) and will handle basic protocol issues.

Because of the selected polling mode, the application must check for incoming packets itself by calling tlp_get() with a pointer to a local data buffer. Depending on the returned values and possible error (Time Out), the application may branch appropriately.

A sample echoing application using polling without callbacks is shown in chapter 17.

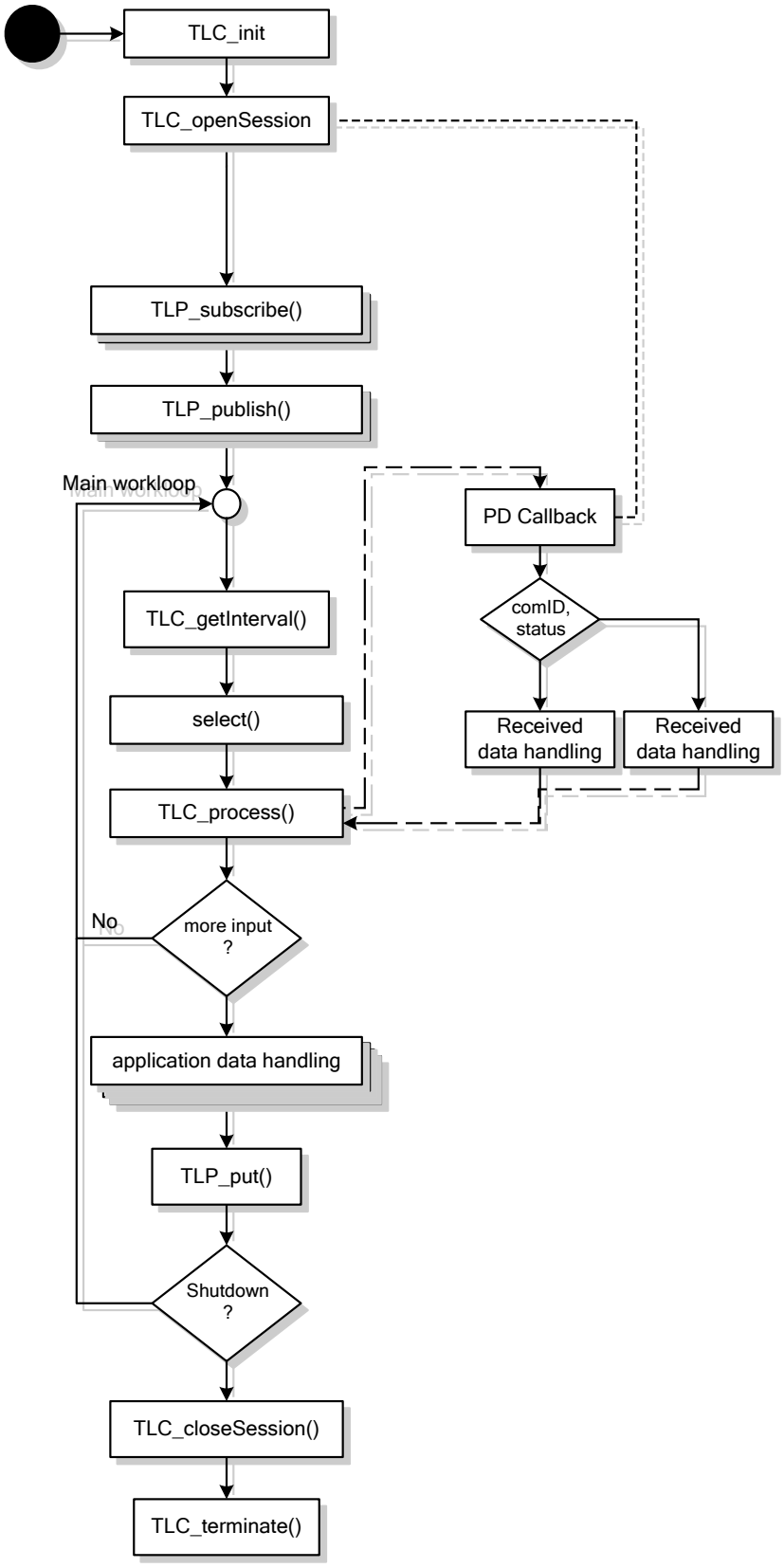

Figure 13 Single Thread PD Callback Workflow

Figure 13 shows a sample sequence of an application using the Posix select() function. After initializing the library including memory management and debug messages with tlc_init() a trdp session related to a defined IP address will be opened (tlc_openSession()). For single thread callback mechanism the session is opened in blocking mode providing callback function pointers. The application registers telegrams by subscribing and publishing its comId's. These functions return a reference handle to be used on successive calls, i. e. get/put or unpublish/unsubscribe.

Within the application main loop, at first tlc_getInterval() is called. It modifies a supplied file descriptor set and a timeout value suitable for use with select() and the standard time macros and functions.

The returned file descriptor set will include the descriptors used by subscribed comId's.

The returned timeout value is the maximum delay time until the next PD must be sent or is due to be received. The application may modify this value before it is handed over to the select() function to allow shorter loop-times.

Select() returns if one of the descriptors are ready for reading (data received) or the timeout was reached.

In any case, tlc_process() must be called then.

tlc_process() will check for any process data pending to be sent or to be received. It will set error states in case packets are overdue (timed out) and will handle basic protocol issues.

It will call the appropriate callbacks if packets have been received or have timed out. The callbacks can be called several times, if several events needed to be handled.

The application may check for incoming packets itself by calling tlp_get() with a pointer to a local data buffer. Always the last received telegram will be returned.

Because of the single threaded handling, there are only few restrictions to the calling sequence. If tlp_subscribe() or tlp_publish() is called from within a callback routine, the callback could be repeated.

tlc_process() must not be called from inside a callback routine!

A sample echoing application using select with callbacks is shown in chapter 17.

8.10. Process Data API - Low Level

8.10.1. Specific types

Definition of the process data call back function see chapter 7.4.4

Type	Name	Description
void *	TRDP_PUB_T	Publish handle

Table 23 Type `TRDP_PUB_T` – publisher handle

Type	Name	Description
void *	TRDP_SUB_T	Subscribe handle

Table 24 Type `TRDP_SUB_T` – subscriber handle

8.10.2. tlp_publish

This function is a low level function of TRDP.

Name	tlp_publish

Synopsis C	`TRDP_ERR_T tlp_publish (`
	`  TRDP_APP_T  appHandle,`
	`  TRDP_PUB_T  *pPubHandle,`
	`  UINT32  comID,`
	`  UINT32  topoCount,`
	`  TRDP_IP_ADDR_T  srcIpAddr,`
	`  TRDP_IP_ADDR_T  destIpAddr,`
	`  UINT32  interval,`
	`  UINT32  redId,`
	`  TRDP_FLAGS_T  pktFlags,`
	`  const TRDP_SEND_PARAM_T  *pSendParam,`
	`  const UINT8  *pData,`
	`  UINT32  dataSize);`

Synopsis C++	`static TRDP_ERR_T tlp::publish (`
	`  TRDP_PUB_T  *pPubHandle,`
	`  UINT32  comID,`
	`  UINT32  topoCount,`
	`  TRDP_IP_ADDR_T  srcIpAddr,`
	`  TRDP_IP_ADDR_T  destIpAddr,`
	`  UINT32  interval,`
	`  UINT32  redId ,`
	`  TRDP_FLAGS_T  pktFlags,`
	`  const TRDP_SEND_PARAM_T  *pSendParam,`
	`  const UINT8  *pData,`
	`  UINT32  dataSize);`

Abstract	Queue a PD message, it will be sent when tlc_process has been called.

Parameters	`appHandle`	(C) handle returned by tlc_openSession()
	`pHandle`	Pointer to publish handle, handle will be returned
	`comId`	ComId
	`topoCount`	Valid topo count, 0 for local consist communication
	`srcIpAddr`	Source IP address. If 0 the IP address given with tlc_openSession() is taken. Only available in the low layer API.
	`destIpAddr`	Destination IP address.
	`interval`	frequency of PD packet (>= 10ms) in microseconds, 0 – only published once (answer on a request)
	`redId`	Redundancy group identifier, set to 0 if not redundant
	`pktFlags`	OPTIONS: TRDP_FLAGS_MARSHALL, TRDP_FLAGS_CALLBACK
	`pSendParam`	Optional setting of QoS and TTL. NULL if default settings shall be used
	`pData`	Data packet
	`dataSize`	size of data packet
Returns C	0 if ok, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_PARAM_ERR parameter error TRDP_MEM_ERR could not insert (out of memory) TRDP_QUEUE_ERR not in queue TRDP_NOINIT_ERR handle invalid	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1.	

Destination IP address:

The destination IP address used can either be, in priority order:

1. If the parameter destId is used, the destination IP configured for a destination Id of the ComId.
2. The destination IP address given in the call.
3. The PD destination IP configured for the ComId.

8.10.3. tlp_unpublish

This function is a low level function of TRDP.

Name	tlp_unpublish	

Synopsis C	`TRDP_ERR_T tlp_unpublish(`
	`  TRDP_APP_T  appHandle,`
	`  TRDP_PUB_T  pubHandle);`	

Synopsis C++	`static TRDP_ERR_T tlp::unpublish(` `  TRDP_PUB_T  pubHandle);`	
Abstract	Stop sending PD.	
Parameters	`appHandle`	(C) Handle returned by tlc_openSession()
	`pubHandle`	Handle returned by tlp_ publish
Returns C	0 if ok, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_PARAM_ERR parameter error TRDP_MEM_ERR could not insert (out of memory) TRDP_QUEUE_ERR not in queue TRDP_NOINIT_ERR handle invalid	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1.	

8.10.4. tlp_put

This function is a low level function of TRDP.

Name	**tlp_put**	

Synopsis C	`TRDP_ERR_T tlp_put(`
	`  TRDP_APP_T  appHandle,`
	`  TRDP_PUB_T  pubHandle`
	`  const UINT8  *pData,`
	`  UINT32  dataSize);`	

Synopsis C++	`static TRDP_ERR_T tlp::put(`
	`  TRDP_PUB_T  pubHandle`
	`  const UINT8  *pData,`
	`  UINT32  dataSize);`	

Abstract	Update the data of a publish.	
Parameters	`appHandle`	(C) Handle returned by tlc_openSession()
	`pubHandle`	Handle returned by tlp_ publish
	`pData`	Pointer to data packet
	`dataSize`	size of data packet
Returns C	0 if ok, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_PARAM_ERR parameter error TRDP_MEM_ERR could not insert (out of memory) TRDP_QUEUE_ERR not in queue TRDP_NOINIT_ERR handle invalid	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1.	

8.10.5. tlp_request

This function is a low level function of TRDP.

Name	tlp_request	

Synopsis C	`TRDP_ERR_T tlp_requestPD(`
	`  TRDP_APP_T  appHandle,`
	`  TRDP_SUB_T  subHandle,`
	`  UINT32  comId,`
	`  UINT32  topoCount,`
	`  TRDP_IP_ADDR_T  srcIpAddr,`
	`  TRDP_IP_ADDR_T  destIpAddr,`
	`  UINT32  redId,`
	`  TRDP_FLAGS  pktFlags,`
	`  const TRDP_SEND_PARAM_T *pSendParam,`
	`  const UINT8 *pData,`
	`  UINT32  dataSize,`
	`  UINT32  replyComId,`
	`  UINT32  replyIpAddr);`	

Synopsis C++	`static TRDP_ERR_T tlp::request(`
	`  TRDP_SUB_T  subHandle,`
	`  UINT32  comId,`
	`  UINT32  topoCount,`
	`  TRDP_IP_ADDR_T  srcIpAddr,`
	`  TRDP_IP_ADDR_T  destIpAddr,`
	`  UINT32  redId,`
	`  TRDP_FLAGS  pktFlags,`
	`  const TRDP_SEND_PARAM_T *pSendParam,`
	`  const UINT8 *pData,`
	`  UINT32  dataSize,`
	`  UINT32  replyComId,`
	`  UINT32  replyIpAddr);`	

Abstract	Send a PD request message, it will be sent when tlc_process has been called.	
Parameters	`appHandle`	(C) Handle returned by tlc_openSession()
	`subHandle`	Returned handle by tlp_subscribe() for the subscription to replyComId
	`comId`	ComId of the packet to be sent
	`topoCount`	Valid topo count.
	`srcIpAddr`	Source IP address. If 0 inserted by the TRDP stack. Only available in the low layer API.
	`destIpaddr`	Destination IP address.
	`redId`	Redundancy group identifier, set to 0 if not redundant
	`pktFlags`	OPTIONS: TRDP_FLAGS_MARSHALL, TRDP_FLAGS_CALLBACK
	`pSendParam`	Optional setting of QoS and TTL. NULL if default settings shall be used
	`pData`	Data packet
	`dataSize`	size of data packet
	`replyComId`	Requested ComId

	`replyIpAddr`	Requested reply IP address
Returns C	0 if ok, !=0 if error, see chapter 12.1.	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1.	

8.10.6. tlp_subscribe

This function is a low level function of TRDP.

Name	tlp_subscribe	

Synopsis C	`TRDP_ERR_T tlp_subscribe(`
	`  TRDP_APP_T  appHandle,`
	`  TRDP_SUB_T  *pSubHandle,`
	`  void  *pUserRef,`
	`  UINT32  comId,`
	`  UINT32  topoCount,`
	`  TRDP_IP_ADDR_T  srcIpAddr1,`
	`  TRDP_IP_ADDR_T  srcIpAddr2,`
	`  TRDP_IP_ADDR_T  destIpAddr,`
	`  TRDP_FLAGS_T  pktFlags,`
	`  UINT32  timeout,`
	`  TRDP_TO_BEHAVIOR_T  toBehavior,`
	`  UINT32  maxDataSize);`	

Synopsis C++	`static TRDP_ERR_T tlp::subscribe(`
	`  TRDP_SUB_T  *pSubHandle,`
	`  void  *pUserRef,`
	`  UINT32  comId,`
	`  UINT32  topoCount,`
	`  TRDP_IP_ADDR_T  srcIpAddr1,`
	`  TRDP_IP_ADDR_T  srcIpAddr2,`
	`  TRDP_IP_ADDR_T  destIpAddr,`
	`  TRDP_FLAGS_T  pktFlags,`
	`  UINT32  timeout,`
	`  TRDP_TO_BEHAVIOR_T  toBehavior,`
	`  UINT32  maxDataSize);`	

Abstract	Prepare for receiving PD messages. Subscribe to a specific PD ComID and source IP address.	
Parameters	`appHandle`	(C) Handle returned by tlc_openSession()
	`pSubHandle`	Returns a unique handle for this subscription
	`pUserRef`	User reference value returned in the info structure.
	`comId`	ComId
	`topoCount`	Topology counter
	`srcIpAddr1`	Source IP address for source filtering, set to zero if not used
	`srcIpAddr2`	Second source IP address for source filtering, set to zero if not used. Used e.g. for source filtering of redundant devices.
	`destIpAddr`	Destination IP address.
	`pktFlags`	OPTIONS: TRDP_FLAGS_MARSHALL, TRDP_FLAGS_CALLBACK
	`timeout`	Timeout in microseconds
	`toBehavior`	Timeout behaviour (set to zero or keep the last value)
	`maxDataSize`	Maximum size of data packet
Returns C	0 if ok, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error	

	TRDP_PARAM_ERR	parameter error
	TRDP_QUEUE_ERR	not in queue
	TRDP_NOINIT_ERR	handle invalid
Returns C++	0 if ok, if error exception thrown, see chapter 12.1.	

8.10.7. tlp_unsubscribe

This function is a low level function of TRDP.

Name	**tlp_unsubscribe**	

Synopsis C	`TRDP_ERR_T tlp_unsubscribe (`
	`  TRDP_APP_T  appHandle,`
	`  TRDP_SUB_T  subHandle);`	

Synopsis C++	`static TRDP_ERR_T tlp::unsubscribe (` `  TRDP_SUB_T  subHandle);`	
Abstract	Stop the referenced subscription	
Parameters	`appHandle`	(C) Handle returned by tlc_openSession()
	`subHandle`	Handle returned by tlp_subscribe
Returns C	0 if ok, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_PARAM_ERR parameter error TRDP_QUEUE_ERR not in queue TRDP_NOINIT_ERR handle invalid	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1.	

8.10.8. tlp_get

Name	tlp_get	

Synopsis C	`TRDP_ERR_T tlp_get(`
	`  TRDP_APP_T  appHandle,`
	`  TRDP_SUB_T  subHandle,`
	`  TRDP_PD_INFO  *pPdInfo,`
	`  UINT8  *pData,`
	`  UINT32  *pDataSize);`	

Synopsis C++	`static TRDP_ERR_T tlp::get(`
	`  TRDP_SUB_T  subHandle,`
	`  TRDP_PD_INFO  *pPdInfo,`
	`  UINT8  *pData,`
	`  UINT32  *pDataSize);`	

Abstract	Get the last valid PD message. This allows polling of PD's instead of event driven handling by callback.	
Parameters	`appHandle`	(C) Handle returned by tlc_openSession()
	`subHandle`	Handle returned by tlp_subscribe
	`pPdInfo`	Pointer to the info structure
	`pData`	Pointer to application buffer
	`pDataSize`	In: size of buffer, Out: data size
Returns C	0 if ok, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_PARAM_ERR parameter error TRDP_QUEUE_ERR not in queue TRDP_NOINIT_ERR handle invalid	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1.	

8.10.9. tlp_setRedundant

Name	***tlp_setRed*undant**	

Synopsis C	`TRDP_ERR_T tlp_setRedundant(`
	`  TRDP_APP_T appHandle,`
	`  UINT32  redId ,`
	`  BOOL8  leader);`	

Synopsis C++	`static TRDP_ERR_T tlp::setRedundant(`
	`  UINT32  redId,`
	`  BOOL8  leader);`	

Abstract	Change the redundant mode for all ComId's configured as redundant with a redundant group id value equal to the parameter `redId`. In leader mode the data will be sent and in follower mode the data will not be sent for published ComId's. The parameter `redId` == 0 means all as redundant defined PD ComId's.	
Parameters	`appHandle`	(C) Handle returned by tlc_openSession()
	`redId`	Redundant ID of the comId's to be controlled
	`leader`	TRUE(1) = leader, FALSE (0) = follower
Returns C	0 if ok, !=0 if error, see chapter 12.1.	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1.	

8.10.10. tlp_getRedundant

Name	***PDComAPI_g*etRedundant**	

Synopsis C	`TRDP_ERR_T tlp_getRedundant(`
	`  TRDP_APP_T  appHandle,`
	`  UINT32  redId,`
	`  BOOL8  *pLeader);`	

Synopsis C++	`static TRDP_ERR_T tlp::getRedundant(`
	`  UINT32  redId,`
	`  BOOL8  *pLeader);`	

Abstract	Get the redundant mode for all ComId's configured as redundant with a redundant Id value equal to the parameter `redId`. The parameter `redId` == 0 means all as redundant defined PD ComId's.	
Parameters	`appHandle`	Handle returned by tlc_openSession()
	`redId`	Redundant ID of the comId's to be controlled.
	`pLeader`	TRUE(1) = active (leader), FALSE (0) = passive (follower)
Returns C	0 if ok, !=0 if error, see chapter 12.1	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1.	

9. Message Data Communication

9.1. Overview – to be updated

9.2. Communication Patterns

Communication between applications can be done in several ways.

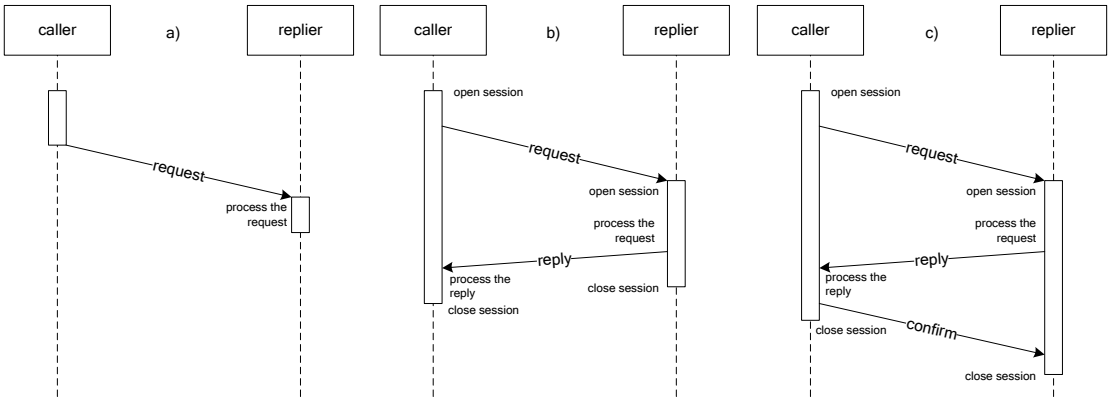

Figure 14 MD communication patterns

The following push patterns are supported

- point to point , sporadic with acknowledge (reply), source knows the sink
- point to point , sporadic without acknowledge (reply), source knows the sink
- point to multipoint, sporadic with acknowledge (reply), source knows the sink
- point to multipoint, sporadic without acknowledge (reply), source knows the sink
- point to multipoint, sporadic with acknowledge (reply), source does not know the sink
- point to multipoint, sporadic without acknowledge (reply), source does not know the sink

The following pull patterns are supported:

- point to point , sporadic with acknowledge (confirm), sink knows the source
- point to point , sporadic without acknowledge (confirm), sink knows the source
- point to multipoint, sporadic with acknowledge (confirm), sink knows the source
- point to multipoint, sporadic without acknowledge (confirm), sink knows the source
- point to multipoint, sporadic on first acknowledge (confirm), sink does not know the source
- point to multipoint, sporadic without acknowledge (confirm), sink does not know the source

9.2.1. Supported Protocols

Message data communication is provided based on two protocols

1. via UDP – for real-time message data limited to 65388 byte length
2. via TCP – for large data up to 65388 byte length

The base protocol can be chosen only once for a MD session.

9.2.2. Unicast Communication

Unicast messages are sent to one device and they are acknowledged. The sending application can be notified about the communication result, i.e. if the sending was ok or if any error has occurred.

The transport layer sends the message to a single receiver device transport layer.

9.2.3. Multicast Communication

Multicast messages are sent to several devices. The sending application can only be notified about the communication result when a reply is requested.

The transport layer sends the message to several receiving devices transport layer.

9.2.4. FRG Multicast Communication

FRG multicast messages are sent to several devices including redundant functions and they are typically only replied by the device with the redundant function activated as leader. The sending application can only be notified about the communication result when a reply is requested.

The transport layer sends the message to several receiving devices transport layer. This is used to send messages to redundant devices. Only one device shall be activated as leader. From the sending application it works as a unicast communication.

9.2.5. Notification Message – Request without Reply

This type of message is send to one or more receiver(s) and no response is expected from the receiver application(s). The application sends a notification message ('Mn'). Both multicast and unicast messages can be send.

In the unicast and the FRG multicast case the sending application can receive a communication result either via a queue (not available in the low layer) message or via a call of a callback.

9.2.6. Request Message – Request with expected Reply

This type of message can send to one or more receiver(s). Each receiver application should return a response message if a request message ('Mr') has been received. Both multicast and unicast request messages can be send.

The sending application is retrieving the received response(s) and sometimes the communication result by reading from the queue (not available in the low layer) or via a call of the callback function. The number of expected responses can be specified in the call, or left unspecified.

If the number of expected responses is specified and there are no or too few responses received after the time-out time expires a communication result will be transmitted to the application.

If the number of expected responses is unspecified the communication result will always be transmitted to the application after the response time-out time expires.

In both cases any received response message will be transmitted to the application before the communication result.

A repetition of an already received request (in case the reply was not received) will be detected during the time the session is open at receiver side and the reply will be repeated without notifying the receiver side application again.

9.2.7. Reply Message without requested Confirmation

This type of message is sent to the sender of a previous received request. The application sends a reply message ('Mp'). A reply message is always sent as unicast containing the URI host part of the replier in the source URI of the reply.

The sending application can receive a communication result either via a queue message by adding a queue identity in the call (not available in the low layer) or via a call of a callback function given in the send reply call.

9.2.8. Reply Message with requested Confirmation

This type of message is sent to the sender of a previous received request. The application sends a reply message ('Mq'). A reply message is always send as unicast containing the URI host part of the replier in the source URI of the reply.

The sending application can receive a communication result either via a queue message (not available in the low layer) or via a call of a callback function.

9.2.9. Confirm message

This type of message is sent to the sender of a previous received reply with confirmation request. The application sends a confirm message ('Mc'). A confirm message is always sent as unicast containing the with the reply received source URI of the replier in the destination URI of the confirm.

The sending application can receive a communication result either via a queue message (not available in the low layer) or via a call of a callback function.

9.2.10. Specified and Unspecified ComID

For each *specified* ComID there is an entry in the configuration database configuring destination, dataset, etc, but not all parameters need to be specified in the configuration database.

Applications may send and listen for ComID's with *unspecified* dataset. Due to lack of information it is not possible for MDCom to perform any marshalling on a message with unspecified dataset. Therefore the message is forwarded to the application as is, without any actions.

For ComId's with *unspecified* destination the application has to provide the destination URI in the method when sending a message.

9.2.11. Echo Server

End devices with MDCom contain echo functions. A system that wants to test communication with an end device sends an echo request (ERQ) to the end device by using the MD request method. The end device activates an echo function that creates an echo response (ERP), copies data from the ERQ to the ERP and then returns it to the sender.

The Message Data Echo is implemented in TRDP.

When message data arrive with the echo ComId (see Table 59) an echo message is created and returned to the source. The data of the returned message is a copy of the received message.

9.3. Adding Listeners

There are a set of listeners that can be used by an application to receive specific message data. There are listeners for ComId's and for user URI's. An application can add listeners for both cases at the same time. A received message will be forwarded to the application if it fulfils the ComId or the URI criteria.

Note: If the listeners are added with different caller references they are considered as separate subscriptions and a received message will be forwarded once for each listener.

9.4. Marshalling and Unmarshalling

With adding a listener or sending MD the optional marshalling/unmarshalling function given in the initialization of the TRDP stack can be chosen. Using this option the MD sent out will be marshalled automatically as well as the received MD will be unmarshalled automatically.

Note: Since a safe data channel can only start after marshalling and ends before unmarshalling, due to the necessary CRC calculations, this option can be used only for unsafe channels (see also chapter 4.4).

Note: Data given to the callback functions are NOT unmarshalled to prevent additional copying. Thus, an required unmarshalling needs to be done in the application.

9.5. Callback functionality

It might be useful to get MD and related error messages via call back function. This can be chosen in the related calls for adding a listener or sending MD. In this case the in tlc_openSession given call back function will be called.

Note: Data given to the callback functions are NOT unmarshalled to prevent additional copying. Thus, an required unmarshalling needs to be done in the application.

9.6. Time-out

There are two types of time-outs used for MD communication, response time-out and confirmation time-out, see chapter 10.2.5. The configured response time-out may be overridden by the application calling the API function for sending a request.

Confirmation time-out is used when waiting for a confirmation in the case the reply type sent out was requesting a confirmation.

Response time-out is used when waiting for response message(s) for a sent request message.

For request messages the response time-out supervision is started after the message is sent.

For reply messages requesting a confirm the time-out supervision is started after the reply message is sent (e.g. the maximum time-out time is equal to the confirm time-out time).

9.7. Redundancy

TRDP supports redundant applications and provides the means to communicate via MD by using multicast addressing for the distribution.

9.7.1. Sending to a Redundant Application

Sending to redundant application normal listener shall be done as unicast or multicast messages. However if unicast is used it has to be send to all redundant devices.

9.7.2. Sending from a Redundant Application

It is the redundant application that is responsible to select when an MD message should be sent or not. Sending a message is done as in the non-redundant case, by calling the TRDP API for sending.

Typically the follower application shall not send any MD, but there might be application specific reasons for this, e.g. implementing a heartbeat function between the redundant pair of applications/devices.

There is thus no special TRDP support for the redundancy leader or the redundancy follower application when sending a message.

9.7.3. Receiving/Sending by a Redundant Application

Depending on the chosen mechanism the redundant application needs to handle the redundancy regarding sending and receiving MD. Typically also the follower application should receive the messages but it should not send out a response.

9.8. Addressing Parameters

This chapter describes how the source and destination addressing parameters transmitted in the wire protocol are determined in the function calls for TRDP message data transfer.

9.8.1. Source IP Address

The source IP address in the MD header is selected by TRDP in one of the following ways, listed in priority order. The source IP address can be used to calculate the host part of the source URI.

1. If the source IP address given by the parameter `srcIpAddr` is not zero, this address shall be used as source IP address.
2. If the parameter `srcIpAddr` is zero, the configured source IP address for the session shall be used as source IP address.

9.8.2. Destination IP Address

The destination IP address in the MD header is selected by TRDP in one of the following ways, listed in priority order. The destination IP address can be used to calculate the host part of the destination URI.

3. If the destination IP address given by the parameter `destIpAddr` is not zero, this address shall be used as destination IP address.
4. If the parameter `destIpAddr` is zero, the configured destination IP address for the given ComId shall be used as destination IP address.

9.8.3. User Part of the Source URI

The user part of the source URI sent in the MD protocol header is defined in one of the following ways, listed in priority order. If the user part of the source URI is used it shall be unique per IP address.

1. If a source URI user part is given by the parameter `srcURI` it shall be used.
2. If the parameter `srcURI` is NULL, the configured source URI user part for the given ComId shall be used for notifications, requests and confirmations.
3. If `srcURI` is NULL, an empty string shall be used as destination URI for notifications, requests and confirmations. For replies TRDP shall use the user reference as hexadecimal value in ASCII characters in a zero terminated string.

9.8.4. User Part of the Destination URI

The user part of the destination URI sent in the MD protocol header is defined in one of the following ways, listed in priority order.

1. If the destination URI user part is given by the parameter `destURI` is not NULL it shall be used.
2. If the parameter `destURI` is NULL, the configured destination URI user part for the given ComId shall be used for notifications and requests.
3. If `destURI` is NULL, an empty string shall be used as destination URI user part for notifications and requests. For replies and confirmations the received source URI user part shall be used.

9.9. Use Cases

The use case in the figure below is an example where two or more applications are using the TLM API as interface to their communication component.

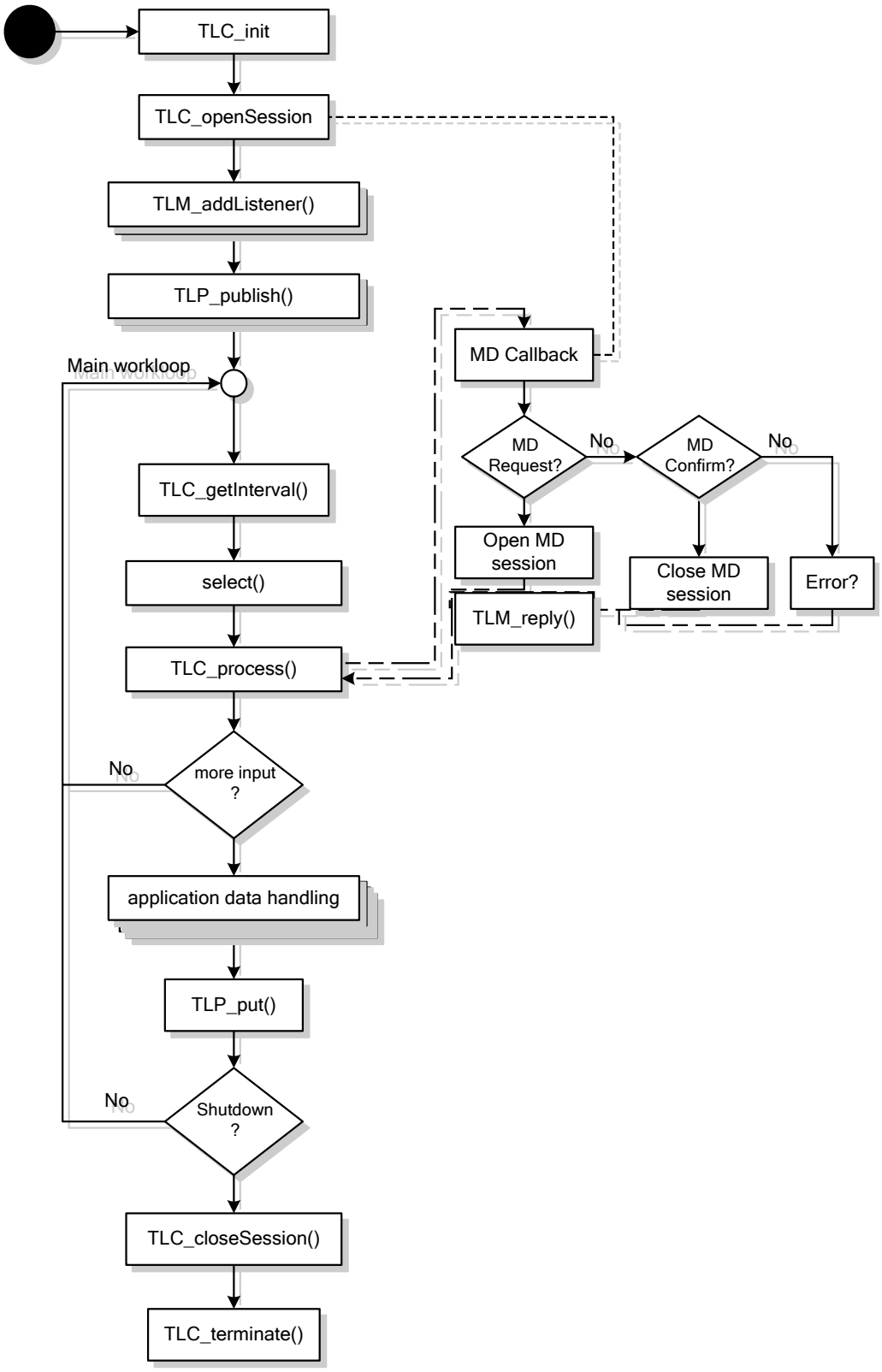

Figure 15 Single Thread MD Callback Workflow

9.10. Message Data API – Low Level

To get received data, request or reply messages or to get a communication results for a sending there are two alternative usages for the application:

- reading a queue (not in low level interface)
- use callback function

In both cases the application will receive two or three items.

- Message Information structure
- The message length
- The message itself, not for communication result.

9.10.1. Specific Types

For the definition of the message data call back function see chapter 7.4.4.

For session handling an identifier of 16 byte length is used. The sessionID is a UUID according to RFC 4122, time based version.

The UUID is used as identity of a “notification”, a “request-reply” or a “request-reply-confirm” session. At caller side it is used to relate a reply message to the original request message. At replier side it is used in combination with the SourceIPAddress to identify a retransmission of the request or confirm message in case the reply message was not received.

For notification messages the receiving application needs to handle duplicate messages (same SessionId, same SourceIPAddress).

Type	Name	Description
VOS_UUID_T	TRDP_UUID_T	Universal unique identifier. Reuse of the VOS definition (see 13.1.1).

Table 25 Type `TRDP_UUID_T` – universally unique identifier

Type	Name	Description
void *	TRDP_LIS_T	Listener handle

Table 26 Type `TRDP_LIS_T` – listener handle

9.10.2. tlm_notify

This method is used to send message one way, i.e. when no response message is expected from the receiving end. The application can be notified about a communication failure via callback function.

Note: No additional communication failure will be delivered to the callback function if the call of `tlm_notify()` already returns an error.

Name	tlm_notify	

Synopsis C	`TRDP_ERR_T tlm_notify(`
	`  TRDP_APP_T  appHandle,`
	`  void  *pUserRef,`
	`  UINT32  comId,`
	`  UINT32  topoCount,`
	`  TRDP_IP_ADDR_T  srcIpAddr,`
	`  TRDP_IP_ADDR_T  destIpAddr,`
	`  TRDP_FLAGS_T  pktFlags,`
	`  const TRDP_SEND_PARAM_T  *pSendParam,`
	`  const UINT8  *pData,`
	`  UINT32  dataSize,`
	`  const TRDP_URI_USER_T  srcURI,`
	`  const TRDP_URI_USER_T  destURI);`	

Synopsis C++	`TRDP_ERR_T tlm::notify(`
	`  void  *pUserRef,`
	`  UINT32  comId,`
	`  UINT32  topoCount,`
	`  TRDP_IP_ADDR_T  srcIpAddr,`
	`  TRDP_IP_ADDR_T  destIpAddr,`
	`  TRDP_FLAGS_T  pktFlags,`
	`  const TRDP_SEND_PARAM_T  *pSendParam,`
	`  const UINT8  *pData,`
	`  UINT32  dataSize,`
	`  const TRDP_URI_USER_T  srcURI,`
	`  const TRDP_URI_USER_T  dstURI);`	

Abstract	Send a MD notify message.	
Parameters	`appHandle`	(C) Handle returned by tlc_openSession()
	`pUserRef`	Caller reference value. Can be used by the application to connect the report of the communication result with the call. The same value will be reported back by the TRDP when the result is reported back to the application.
	`comId`	ComId of the data set
	`topoCount`	Valid topo count
	`srcIpAddr`	Source IP address. Typically set by TRDP stack.
	`dstIpAddr`	Destination IP Address. Used to override any configured destination IP address for the ComId. Set 0 if not used.
	`pktFlags`	OPTIONS: `TRDP_FLAGS_DEFAULT`, TRDP_FLAGS_MARSHALL, TRDP_FLAGS_TCP `Note: using TRDP_FLAGS_DEFAULT` the flags specified in the default MD configuration are used Note: using TRDP_FLAGS_TCP instead of a UDP notification a TCP notification will be sent out
	`pSendParam`	Pointer to optional send parameters QoS and TTL. Set NULL to use default parameters.

	pData	Pointer to data to be sent
	dataSize	Number of data bytes to send
	srcURI	Pointer to the source IP address unique user part of the source URI string. It will override any configured source URI for the ComId. The user part of the source URI string is sent in the MD protocol header. Set to NULL if not used. **Note:** The host part can be created at destination side out of the srcIpAddr Syntax of a complete URI string: [user][@host].
	destURI	Pointer to the user part of the destination URI string Used to define the destination URI string used to override any configured destination URI for the ComId. The user part of the destination URI string is sent in the MD protocol header. Set to NULL if not used. **Note:** The host part can be created at destination side out of the destIpAddr Syntax of a complete URI string: [user][@host].
Returns C	0 if ok, !=0 if error, see chapter 12.1	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1	

9.10.3. tlm_request

This method is used to send message when response is expected from the receiving end application.

The application receives the response(s) and/or the result information, via a callback function. The result information is used, after time-out, when there is any response missing or to report the number of received responses when expected number of responses is unknown.

Note: TRDP will always send response(s) and/or one result information.

Name	tlm_request
Synopsis C	TRDP_ERR_T tlm_notify(TRDP_APP_T appHandle, void *pUserRef, TRDP_UUID_T *pSessionId UINT32 comId, UINT32 topoCount, TRDP_IP_ADDR_T srcIpAddr, TRDP_IP_ADDR_T dstIpAddr, TRDP_FLAGS_T pktFlags, UINT32 numReplies, UINT32 replyTimeout, const TRDP_SEND_PARAM_T *pSendParam, const UINT8 *pData, UINT32 dataSize, const TRDP_URI_USER_T srcURI, const TRDP_URI_USER_T destURI);
Synopsis C++	TRDP_ERR_T tlm::notify (void *pUserRef, TRDP_UUID_T *pSessionId UINT32 comId, UINT32 topoCount, TRDP_IP_ADDR_T srcIpAddr,

	TRDP_IP_ADDR_T destIpAddr, TRDP_FLAGS_T pktFlags, UINT32 numReplies, UINT32 replyTimeout, const TRDP_SEND_PARAM_T *pSendParam, const UINT8 *pData, UINT32 dataSize, const TRDP_URI_USER_T srcURI, const TRDP_URI_USER_T destURI);	
Abstract	Send a MD request message.	
Parameters	appHandle	(C) Handle returned by tlc_openSession()
	pUserRef	User reference value. Can be used by the application to connect the report of the communication result with the call. The same value will be reported back by the TRDP when the result is reported back to the application.
	pSessionId	Pointer to the session identifier given back by this call
	ComId	ComId of the data set
	topoCount	Valid topo count
	srcIpAddr	Source IP address. Typically set by TRDP stack.
	dstIpAddr	Destination IP Address. Used to override any configured destination IP address for the ComId. Set 0 if not used.
	pktFlags	OPTIONS: TRDP_FLAGS_DEFAULT, TRDP_FLAGS_MARSHALL, TRDP_FLAGS_TCP Note: using TRDP_FLAGS_DEFAULT the flags specified in the default MD configuration are used Note: using TRDP_FLAGS_TCP instead of a UDP request a TCP request will be sent out
	numReplies	number of expected replies, 0 if unknown
	replyTimeout	timeout for reply in microseconds
	pSendParam	Pointer to optional send parameters QoS and TTL. Set NULL to use default parameters.
	pData	Pointer to data to be sent
	dataSize	Number of data bytes to send
	srcURI	Pointer to the source IP address unique user part of the source URI string. It will override any configured source URI for the ComId. The user part of the source URI string is sent in the MD protocol header. Set to NULL if not used. **Note:** The host part can be created at destination side out of the srcIpAddr Syntax of a complete URI string: [user][@host].
	destURI	Pointer to the user part of the destination URI string Used to define the destination URI string used to override any configured destination URI for the ComId. The user part of the destination URI string is sent in the MD protocol header. Set to NULL if not used. **Note:** The host part can be created at destination side out of the destIpAddr Syntax of a complete URI string: [user][@host].
Returns C	0 if ok, !=0 if error, see chapter 12.1	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1.	

9.10.4. tlm_reply

This method is used by the application to send a response message on an earlier received request message without requesting a confirmation.

Name	tlm_reply	

Synopsis C	`TRDP_ERR_T tlm_reply(`
	`  TRDP_APP_T  appHandle,`
	`  const void  *pUserRef,`
	`  TRDP_UUID_T  *pSessionId`
	`  UINT32  comId,`
	`  UINT32  topoCount,`
	`  TRDP_IP_ADDR_T  srcIpAddr,`
	`  TRDP_IP_ADDR_T  destIpAddr,`
	`  TRDP_FLAGS_T  pktFlags,`
	`  UINT32  userStatus,`
	`  const TRDP_SEND_PARAM_T  *pSendParam,`
	`  const UINT8  *pData,`
	`  UINT32  dataSize,`
	`  const TRDP_URI_USER_T  srcURI,`
	`  const TRDP_URI_USER_T  destURI);`	

Synopsis C++	`TRDP_ERR_T tlm::reply(`
	`  const void  *pUserRef,`
	`  TRDP_UUID_T  *pSessionId`
	`  UINT32  comId,`
	`  UINT32  topoCount,`
	`  TRDP_IP_ADDR_T  srcIpAddr,`
	`  TRDP_IP_ADDR_T  destIpAddr,`
	`  TRDP_FLAGS_T  pktFlags,`
	`  UINT32  userStatus,`
	`  const TRDP_SEND_PARAM_T  *pSendParam,`
	`  const UINT8  *pData,`
	`  UINT32  dataSize,`
	`  const TRDP_URI_USER_T  srcURI,`
	`  const TRDP_URI_USER_T  destURI);`	

Abstract	Send a MD reply message without requesting a confirmation.	
Parameters	`appHandle`	(C) Handle returned by tlc_openSession()
	`pUserRef`	User reference value. Can be used by the application to connect the report of the communication result with the call. The same value will be reported back by the TRDP when the result is reported back to the application.
	`pSessionId`	Pointer to session identifier of the related request
	`ComId`	ComId of the telegram
	`topoCount`	Valid topo count
	`srcIpAddr`	Source IP address. Typically set by TRDP stack.
	`destIpAddr`	Destination IP Address. Used to override any configured destination IP address for the ComId. Set 0 if not used.
	`pktFlags`	OPTIONS: `TRDP_FLAGS_DEFAULT`, `TRDP_FLAGS_MARSHALL` `Note: using TRDP_FLAGS_DEFAULT` the flags specified in the default MD configuration are used
	`userStatus`	Info for requester about application errors

	`pSendParam`	Pointer to optional send parameters QoS and TTL. Set NULL to use default parameters.
	`pData`	Pointer to data to be sent
	`dataSize`	Number of data bytes to send
	`srcURI`	Pointer to the source IP address unique user part of the source URI string of the replier. It will override any configured source URI for the ComId. The user part of the source URI string is sent in the MD protocol header. Set to NULL if not used. **Note:** The host part can be created at destination side out of the `srcIpAddr` Syntax of a complete URI string: [user][@host].
	`destURI`	Pointer to the user part of the destination URI string received in the request. Used to override any configured destination URI for the ComId. The user part of the destination URI string is sent in the MD protocol header. Set to NULL if not used. **Note:** The host part can be created at destination side out of the `destIpAddr` Syntax of a complete URI string: [user][@host].
Returns C	0 if ok, !=0 if error, see chapter 12.1	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1.	

9.10.5. tlm_replyQuery

This method is used by the application to send a response message on an earlier received request message requesting a confirmation.

Name	**tlm_replyQuery**

Synopsis C	`TRDP_ERR_T tlm_replyQuery(`
	`  TRDP_APP_T  appHandle,`
	`  const void  *pUserRef,`
	`  TRDP_UUID_T  *pSessionId`
	`  UINT32  comId,`
	`  UINT32  topoCount,`
	`  TRDP_IP_ADDR_T  srcIpAddr,`
	`  TRDP_IP_ADDR_T  destIpAddr,`
	`  TRDP_FLAGS_T  pktFlags,`
	`  UINT32  userStatus,`
	`  UINT32  confirmTimeout,`
	`  const TRDP_SEND_PARAM_T *pSendParam,`
	`  const UINT8  *pData,`
	`  UINT32  dataSize,`
	`  const TRDP_URI_USER_T  srcURI,`
	`  const TRDP_URI_USER_T  destURI);`

Synopsis C++	`TRDP_ERR_T tlm::replyQuery(`
	`  const void  *pUserRef,`
	`  TRDP_UUID_T  *pSessionId`
	`  UINT32  comId,`
	`  UINT32  topoCount,`
	`  TRDP_IP_ADDR_T  srcIpAddr,`

	TRDP_IP_ADDR_T destIpAddr, TRDP_FLAGS_T pktFlags, UINT32 userStatus, UINT32 confirmTimeout, const TRDP_SEND_PARAM_T *pSendParam, const UINT8 *pData, UINT32 dataSize, const TRDP_URI_USER_T srcURI, const TRDP_URI_USER_T destURI);	
Abstract	Send a MD reply message requesting a confirmation.	
Parameters	appHandle	(C) Handle returned by tlc_openSession()
	pUserRef	User reference value. Can be used by the application to connect the report of the communication result with the call. The same value will be reported back by the TRDP when the result is reported back to the application.
	pSessionId	Pointer to session identifier of the related request
	comId	ComId of the telegram
	topoCount	Valid topo count
	srcIpAddr	Source IP address. Typically set by TRDP stack.
	destIpAddr	Destination IP Address. Used to override any configured destination IP address for the ComId. Set 0 if not used.
	pktFlags	OPTIONS: TRDP_FLAGS_DEFAULT, TRDP_FLAGS_MARSHALL Note: using TRDP_FLAGS_DEFAULT the flags specified in the default MD configuration are used
	userStatus	Info for requester about application errors
	confirmTimeout	timeout for confirmation in microseconds
	pSendParam	Pointer to optional send parameters QoS and TTL. Set NULL to use default parameters.
	pData	Pointer to data to be sent
	dataSize	Number of data bytes to send
	srcURI	Pointer to the source IP address unique user part of the source URI string of the replier. It will override any configured source URI for the ComId. The user part of the source URI string is sent in the MD protocol header. Set to NULL if not used. **Note:** The host part can be created at destination side out of the srcIpAddr Syntax of a complete URI string: [user][@host].
	destURI	Pointer to the user part of the destination URI string received in the request. Used to override any configured destination URI for the ComId. The user part of the destination URI string is sent in the MD protocol header. Set to NULL if not used. **Note:** The host part can be created at destination side out of the destIpAddr. Syntax of a complete URI string: [user][@host].
Returns C	0 if ok, !=0 if error, see chapter 12.1	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1.	

9.10.6. tlm_replyErr

This method is used to send an error message on an earlier received request message. The call is typically used from the TRDP layers itself.

Name	**tlm_replyErr**	

Synopsis C	`TRDP_ERR_T tlm_replyErr(`
	`  TRDP_APP_T  appHandle,`
	`  TRDP_UUID_T  *pSessionId`
	`  UINT32  comId,`
	`  UINT32  topoCount,`
	`  TRDP_IP_ADDR_T  srcIpAddr,`
	`  TRDP_IP_ADDR_T  destIpAddr,`
	`  TRDP_REPLY_STATUS_T  replyStatus,`
	`  const TRDP_SEND_PARAM_T *pSendParam,`
	`  const TRDP_URI_USER_T  srcURI,`
	`  const TRDP_URI_USER_T  destURI);`	

Synopsis C++	`TRDP_ERR_T tlm::replyErr(`
	`  TRDP_APP_T  appHandle,`
	`  TRDP_UUID_T  *pSessionId`
	`  UINT32  comId,`
	`  UINT32  topoCount,`
	`  TRDP_IP_ADDR_T  srcIpAddr,`
	`  TRDP_IP_ADDR_T  destIpAddr,`
	`  TRDP_REPLY_STATUS_T  replyStatus,`
	`  const TRDP_SEND_PARAM_T *pSendParam,`
	`  const TRDP_URI_USER_T  srcURI,`
	`  const TRDP_URI_USER_T  destURI);`	

Abstract	Send a MD error reply message. The call is typically used from the TRDP layers itself.	
Parameters	`appHandle`	(C) Handle returned by tlc_openSession()
	`sessionId`	Session identifier of the related request
	`topoCount`	Valid topo count
	`ComId`	ComId of the telegram
	`srcIpAddr`	Source IP address. Typically set by TRDP stack.
	`destIpAddr`	Destination IP Address. Used to override any configured destination IP address for the ComId. Set 0 if not used.
	`replyStatus`	Info for requester about stack errors
	`pSendParam`	Pointer to optional send parameters QoS and TTL. Set NULL to use default parameters.
	`srcURI`	Pointer to the source IP address unique user part of the source URI string of the replier. It will override any configured source URI for the ComId. The user part of the source URI string is sent in the MD protocol header. Set to NULL if not used. **Note:** The host part can be created at destination side out of the `srcIpAddr` Syntax of a complete URI string: [user][@host].

	destURI	Pointer to the user part of the destination URI string received in the request. Used to override any configured destination URI for the ComId. The user part of the destination URI string is sent in the MD protocol header. Set to NULL if not used. **Note:** The host part can be created at destination side out of the destIpAddr. Syntax of a complete URI string: [user][@host].
Returns C	0 if ok, !=0 if error, see chapter 12.1	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1.	

9.10.7. tlm_confirm

This method is used by the application to send a confirm message on an earlier received reply message requesting a confirmation.

Name	tlm_confirm	
Synopsis C	TRDP_ERR_T tlm_confirm(TRDP_APP_T appHandle, void *pUserRef, TRDP_UUID_T *pSessionId UINT32 comId, UINT32 topoCount, TRDP_IP_ADDR_T srcIpAddr, TRDP_IP_ADDR_T destIpAddr, TRDP_FLAGS_T pktFlags, const TRDP_SEND_PARAM_T *pSendParam, const TRDP_U RI_USER_T srcURI, const TRDP_URI_USER_T destURI);	
Synopsis C++	TRDP_ERR_T tlm::confirm(void *pUserRef, TRDP_UUID_T *pSessionId UINT32 comId, UINT32 topoCount, TRDP_IP_ADDR_T srcIpAddr, TRDP_IP_ADDR_T destIpAddr, TRDP_FLAGS_T pktFlags, const TRDP_SEND_PARAM_T *pSendParam, const TRDP_U RI_USER_T srcURI, const TRDP_URI_USER_T destURI);	
Abstract	Send a MD confirm message.	
Parameters	appHandle	(C) Handle returned by tlc_openSession()
	pUserRef	User reference value. Can be used by the application to connect the report of the communication result with the call. The same value will be reported back by the TRDP when the result is reported back to the application.
	pSessionId	Pointer to the session identifier given back by the received reply
	ComId	ComId of the data set
	topoCount	Valid topo count

	srcIpAddr	Source IP address. Typically set by TRDP stack.
	destIpAddr	Destination IP Address. Used to override any configured destination IP address for the ComId. Set 0 if not used.
	pktFlags	OPTIONS: TRDP_FLAGS_DEFAULT Note: using TRDP_FLAGS_DEFAULT the flags specified in the default MD configuration are used.
	pSendParam	Pointer to optional send parameters QoS and TTL. Set NULL to use default parameters.
	srcURI	Pointer to the device unique user part of the source URI string. It will override any configured source URI for the ComId. The user part of the source URI string is sent in the MD protocol header. Set to NULL if not used. **Note:** The host part can be created at destination side out of the srcIpAddr Syntax of a complete URI string: [user][@host].
	destURI	Pointer to the user part of the destination URI string received in the reply. Used to override any configured destination URI for the ComId. The user part of the destination URI string is sent in the MD protocol header. Set to NULL if not used. **Note:** The host part can be created at destination side out of the destIpAddr. Syntax of a complete URI string: [user][@host].
Returns C	0 if ok, !=0 if error, see chapter 12.1	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1.	

9.10.8. tlm_addListener

Name	tlm_addListener	

Synopsis C	`TRDP_ERR_T tlm_addListener(`
	`  TRDP_APP_T  appHandle,`
	`  TRDP_LIS_T  *pListenHandle,`
	`  const void  *pUserRef,`
	`  const UINT32  comId,`
	`  const UINT32  topoCount,`
	`  TRDP_IP_ADDR_T  destIpAddr,`
	`  TRDP_FLAGS_T  *pktFlags,`
	`  const TRDP_URI_USER_T  destURI);`	

Synopsis C++	`TRDP_ERR_T tlm::addListener(`
	`  TRDP_LIS_T  *pListenHandle,`
	`  const UINT32  comId,`
	`  const void  *pUserRef,`
	`  const UINT32  topoCount,`
	`  TRDP_IP_ADDR_T  destIpAddr,`
	`  TRDP_FLAGS_T  *pktFlags,`
	`  const TRDP_URI_USER_T  destURI);`	

Abstract	Add a listener for message data with the given ComId and/or given destination (defined by IP address and URI user part. TRDP will put received messages on the common callback function. Join multicast IP address for given multicast destinations. When a listener is added for both ComId's and destination URI strings using the same caller reference a received message matching both ComId and destination URI string will only be put on the queue once and the callback function will only be called once.	
Parameters	`appHandle`	(C) Handle returned by tlc_openSession()
	`pListenHandle`	Pointer to listener handle to be given back by this call.
	`pUserRef`	User reference value. Can be used by the application to connect a received message with an added listener. The same value will be reported back by the TRDP when a message is received via the queue and/or when the callback function is called.
	`comId`	ComId to listen to. In case of URI listener set 0..
	`topoCount`	Pointer to callback function to be added to the listener list Set to NULL if not used.
	`destIpAddr`	Destination IP Address. Used to join multicast IP addresses. Set 0 if not used.
	`pktFlags`	OPTIONS: `TRDP_FLAGS_DEFAULT`, TRDP_FLAGS_MARSHALL, TRDP_FLAGS_TCP `Note: Using TRDP_FLAGS_DEFAULT` the flags specified in the default MD configuration are used. Note: Using TRDP_FLAGS_TCP instead of a UDP listener a TCP listener will be created.
	`destURI`	Pointer to destination URI string user part. The user part of the URI will be used to match received message, see the following table. Set 0 in case of a ComId listener. Syntax of a complete URI string: user[@[host]].
Returns C	0 if ok, !=0 if error, see chapter 12.1	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1.	

Received destination URI string, user part	Given destination URI string, user part			
	“instX.funcN”	**“aInst.funcN ” or “funcN”**	**“instX.aFunc”**	**“aInst.aFunc ” or “aFunc”**
“instX.funcN”	Yes	Yes	Yes	Yes
“instY.funcN”	No	Yes	No	Yes
“instX.funcM”	No	No	Yes	Yes
“instY.funcM”	No	No	No	Yes
“aInst.funcN” or “funcN”	Yes	Yes	Yes	Yes
“aInst.funcM” or “funcM”	No	No	Yes	Yes
“instX.aFunc”	Yes	Yes	Yes	Yes
“instY.aFunc”	No	Yes	No	Yes
“aInst.aFunc” or “aFunc”	Yes	Yes	Yes	Yes
“”	No	No	No	No

Table 27 Matching of received destination URI strings and URI strings given by the parameter pDestURI.

Joining multicast IP address:

For MD sent as multicast messages the multicast address has to be joined by TRDP.

TRDP will use the destination IP address to check if a multicast address has to be joined.

9.10.9. tlm_delListener

Name	**tlm_delListener**	

Synopsis C	`TRDP_ERR_T tlm_delListener(`
	`  TRDP_APP_T  appHandle,`
	`  TRDP_LIS_T  handle) ;`	

Synopsis C++	`TRDP_ERR_T tlm::delListener(` `  TRDP_LIS_T  handle);`	
Abstract	Remove the specified listener(s) using the given queue	
Parameters	`appHandle`	(C) Handle returned by tlc_openSession()
	`handle`	Listener reference
Returns C	0 if ok, !=0 if error, see chapter 12.1	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1	

9.10.10. tlm_abortSession

Name	tlm_abortSession	

Synopsis C	`TRDP_ERR_T tlm_abortSession(`
	`  TRDP_APP_T  appHandle,`
	`  UINT32  *pSessionId) ;`	

Synopsis C++	`TRDP_ERR_T tlm::abortSession (` `  UINT32  *pSessionId);`	
Abstract	Cancel the referenced session, drop pending messages and set the session id to zero	
Parameters	`appHandle`	(C) Handle returned by tlc_openSession()
	`pSessionId`	Pointer to session id
Returns C	0 if ok, !=0 if error, see chapter 12.1	
Returns C++	0 if ok, if error exception thrown, see chapter 12.1.	

10. Configuration Data

In order to exchange data over TCN, TRDP needs to be configured, e.g. information on how to handle the data, where to send data etc. This information is stored in a *Configuration Database.*

The database can be populated from configuration file(s) in XML-format or with API functions. The database is usually read at start up from a configuration file in XML-format. The file(s) is created by an off-line tool and downloaded as a DLU (Downloadable Unit).

The structure of this XML file is described in a XML Schema Definition (.XSD). The file can be found in the TRDP API directory. By use of XML schema definition it is possible to validate a configuration file. The XSD only describes the formal structure of the configuration and should be used to validate the configurations by TRDP.

The optional utility functions for reading the XML configuration file(s) are described in chapter 14.1.

The definition files can be common for different communication concepts. In the chapters below only features relevant to TRDP are described. The TRDP configuration includes:

- Device Configuration Parameters
- Bus Interface Parameters including related Telegram Parameters (Exchange Parameters)
- Mapped device parameters
- Communication Parameters
- Dataset Parameters
- Debug Parameters

Note: The column “Optional/Required” in the tables below is used for what is optional and required for TRDP and not what is optional and required for the XML file or for other use of the XML file, e.g. tools.

```
<device attributes>
   <device-configuration attributes>
         :
   </device-configuration>
   <bus-interface-list>
      <bus-interface attributes>
         <telegram attributes >
             :
         </telegram>
             :
      </bus-interface>
          :
   </bus-interface-list>
   <mapped-device-list>
      <mapped-device attributes>
         <mapped-bus-interface attributes>
            <mapped-telegram attributes/>
                  :
         </mapped-bus-interface>
               :
      </mapped-device>
             :
   </mapped-device-list>
   <com-parameter-list>
      <com-parameter attributes/>
          :
   </com-parameter-list>
   <data-set-list>
      <data-set attributes >
          :
      </data-set>
          :
   </data-set-list>
   <debug attributes/>
</device>
```

Name	Data Type	Optional/ Required	Description
host-name	STRING	Required	Device host name
leader-name	STRING	Optional	Leader host name, depending on redundancy concept
type	STRING	Optional	Device type, for information only

Table 28 Attributes for device tag

10.1. Device Configuration Parameters

The parameters given with the device configuration are needed tlc_init() call (see chapter 7.4.6).

Device configuration parameters may be provided for:

- Memory configuration

```
<device>
    <device-configuration attributes>
        <mem-block-list>
            <mem-block attributes/>
            <mem-block attributes/>
                 :
        </mem-block-list>
    </device-configuration>
</device>
```

The tag “device-configuration” is optional.
All attributes that can be specified for the tags are described in the following table.

Name	Data Type	Optional/ Required	Description
memory-size	UINT32	Optional	Size of TRDP total memory, default size: 4 MBytes

Table 29 Attributes for device-configuration tag

10.1.1. Memory configuration

The TRDP uses a dynamic memory allocation, see chapter 13.2.

Configuration of memory size and/or configuration of memory blocks may be specified in the XML configuration file. If values is not specified TRDP default values will be used.

```
<device>
    <device-configuration attributes>
        <mem-block-list>
            <mem-block attributes>
                  :
        </mem-block-list>
             :
    </device-configuration>
</device>
```

The tag “mem-block” is optional.
All attributes that can be specified for the tags are described the following table.

Name	Data Type	Optional/ Required	Description
size	UINT32	Required	Size of memory block
preallocate	UINT32	Optional	Number of pre-allocated memory block with the defined size

Table 30 Attributes for mem-block tag

NOTE: TRDP can use 15 different memory block sizes for the size parameter. So only the position and not the defined size itself is relevant. The default configuration for the sizes is shown in Table 65.

10.2. Bus Interface List

The Bus interface list contains the configuration parameter needed for the specific interfaces.

```
<device>
    <bus-interface-list>
        <bus-interface attributes>
            <trdp-process attributes/>
            <pd-com-parameter attributes/>
            <pd-com-parameter attributes/>
             <telegram attributes>
                    :
             </telegram>
                 :
        </bus-interface>
           :
   </bus-interface-list>
</device>
```

10.2.1. Bus Interface Configuration

The parameters given with the bus interface configuration are needed for the interface specific tcl_openSession() call (see chapter 7.4.7).

For each bus interfaces parameters may be provided for:

- Interface configuration
- Thread/task configuration
- Default PD Communication
- Default MD Communication
- Telegram configuration

```
<device>
    <bus-interface-list>
        <bus-interface attributes>
            <trdp-process attributes/>
            <pd-com-parameter attributes/>
            <pd-com-parameter attributes/>
             <telegram attributes >
                      :
             </telegram>
                 :
        </bus-interface>
              :
    </bus-interface-list>
</device>
```

All attributes that can be specified for the tag “bus-interface” are described in the following table.

Name	Data Type	Optional/ Required	Description
network-id	UINT8	Required	Ip interface (1..4) on the device
name	STRING	Required	Interface name
host-ip	UINT32	Optional	Host ip address in case of redundancy
leader-ip	UINT32	Optional	Leader ip address in case of redundancy

Table 31 Attributes for bus-interface tag

10.2.2. Process Configuration

At start up the TRDP main process is spawned for sending and receiving MD and PD. The thread/task priority and for cyclic thread/task the cycle time may be specified in the XML configuration file. If not specified TRDP default values will be used.

```
<device>
    <bus-interface-list>
        <bus-interface attributes>
            <trdp-process attributes/>
            <pd-com-parameter attributes/>
            <pd-com-parameter attributes/>
             <telegram attributes >
                        :
             </telegram>
                  :
        </bus-interface>
             :
     </bus-interface-list>
</device>
```

The tag “trdp-process” is optional.
All attributes that can be specified are described in the following table.

Name	Data Type	Optional/ Required	Description
cycle-time	UINT32	Optional	Cycle time–in µs. Only for cyclic threads/tasks
blocking	STRING	Optional	NO – non - blocking, YES – blocking
traffic-shaping	STRING	Optional	OFF – no traffic shaping, ON – traffic shaping
priority	UINT32	Optional	Priority for trdp process task. Values:1-255, 1 = highest, 0 = default

Table 32 Attributes for trdp-process tag

Name	Value
cycle-time	10000
blocking	NO
traffic-shaping	ON
priority	64

Table 33 Default values for thread/task

10.2.3. PD Communication Parameters

There are a number of parameters that may be configured for the PD communication in the XML configuration file. If not specified TRDP hard coded default values will be used.

```
<device>
    <bus-interface-list>
        <bus-interface attributes>
            <trdp-process attributes/>
            <pd-com-parameter attributes/>
            <md-com-parameter attributes/>
             <telegram attributes >
                      :
             </telegram>
                 :
        </bus-interface>
           :
    </bus-interface-list>
</device>
```

The tag “pd-com-parameter” is optional.

All attributes that can be specified for the tag “pd-com-parameter” are described in the following table.

.

Name	Data Type	Optional/ Required	Description
timeout-value	UINT32	Optional	Timeout value in µs, before considering received process data as invalid. Disabled if 0 or not specified.
validity-behaviour	STRING	Optional	Behaviour when received process data is invalid. ZERO = zero values KEEP = keep last value
ttl	UINT32	Optional	Default time To live for PD.
qos	UINT32	Optional	Default quality of service for PD.
marshall	STRING	Optional	ON/OFF = enable/disable internal marshalling/unmarshalling
callback	STRING	Optional	ON/OFF = enable/disable call back
port	UINT32	Optional	Port to be used for PD communication

Table 34 Attributes for pd-com-parameter tag

Name	Value
timeout-value	100000
validity-behaviour	ZERO
ttl	64
qos	5
marshall	OFF
callback	OFF
port	20548

Table 35 Default values for pd-com-parameter

10.2.4. MD Communication Parameters

There are a number of parameters that may be configured for the MD communication in the XML configuration file. If not specified TRDP hard coded default values will be used.

The here configured default values are used if nothing else is configured for the ComId, see 0

The default time-out time for waiting for a confirm message is used when no confirm time-out value is specified for a ComId.

The default time-out time for waiting for a reply message(s) is used when no reply time-out value is specified for a ComId.

TRDP stores received sequence numbers per source IP address, to detect resend messages that already have been received. More details are described in reference document [Wire]. The maximum number of stored sequence numbers per IP address may be configured.

```
<device>
    <bus-interface-list>
        <bus-interface attributes>
            <trdp-process attributes/>
           <pd-com-parameter attributes/>
           <md-com-parameter attributes/>
            <telegram attributes >
                     :
            </telegram>
                :
        </bus-interface>
           :
    </bus-interface-list>
</device>
```

The tag “md-com-parameter” is optional.

All attributes that can be specified for the tag “md-com-parameter” are described in the following table..

Name	Data Type	Optional/ Required	Description
confirm-timeout	UINT32	Optional	Default time-out time in µs for receiving a confirm message
reply-timeout	UINT32	Optional	Default time-out time in µs for receiving response message(s)
connect-timeout	UINT32	Optional	Default time-out time in µs to close a not used TCP connection
ttl	UINT32	Optional	Default time to live for MD.
qos	UINT32	Optional	Default quality of service for MD.
protocol	STRING	Optional	TCP/UDP = default protocol
marshall	STRING	Optional	ON/OFF = enable/disable internal marshalling/unmarshalling
callback	STRING	Optional	ON/OFF = enable/disable call back
udp-port	UINT32	Optional	Port to be used for UDP MD communication
tcp-port	UINT32	Optional	Port to be used for TCP MD communication
num-sessions	UINT32	Optional	Maximal number of replier sessions to prevent DoS attacks

Table 36 Attributes for md-com-parameter tag

Name	Value
confirm-timeout	1 000 000 µs
reply-timeout	5 000 000 µs
connect-timeout	60 000 000 µs
ttl	64
qos	3
retries	2
protocol	UDP
marshall	OFF
callback	ON
udp-port	20550
tcp-port	20550
num-sessions	1000

Table 37 Default values for md-com-parameter

10.2.5. Telegram Configuration (ExchgPar)

The Telegram configuration contains the central elements for data exchange – the exchange parameters. It is identified by the central key *ComID*.

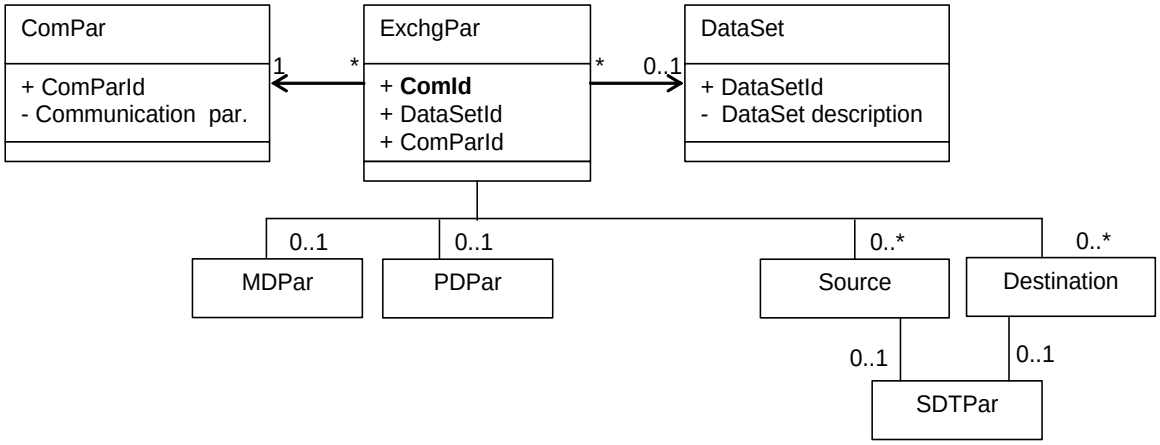

Figure 16 Exchange Parameters with the central key ComId

Each Exchange Parameter references one or no DataSet and one Communication Parameter. (Many Exchange Parameters can reference the same DataSet and Communication Parameter.) The Exchange Parameter also describes how to handle any Process and/or Message Data.

An Exchange Parameter is described in the XML file as a “telegram” tag.

Structure:

```
<device>
    <bus-interface-list>
       <bus-interface attributes>
           <trdp-process attributes/>
           <pd-com-parameter attributes/>
           <md-com-parameter attributes/>
           <telegram attributes >
               <md-parameter attributes />
               <pd-parameter attributes />
               <source attributes />
               <destination attributes />
           </telegram>
              :
        </bus-interface>
         :
    </bus-interface-list>
</device>
```

The tags “sdt-parameter“, “md-parameter”, “pd-parameter”, “source” and “destination” are optional.

The tags “sdt-parameter”, “md-parameter” and “pd-parameter” can only be used one time for each Comid.

For each ComId the tags “source” and “destination” may be used several times with different values.

For some attributes there are default values.

All attributes that can be specified for the telegram tag are described in the following table.

Name	Data Type	Optional/ Required	Description
name	STRING	Optional	Required for tooling and debugging
com-id	UINT32	Required	ID for exchange parameter **Note:** Only the first found configuration of a ComId will be considered.
data-set-id	UINT32	Required for PD Optional for MD	ID for dataset to be exchanged.
com-parameter-id	UINT32	Optional	ID for communication parameter to be used, if not given the default parameters for MD/PD communication will be used

Table 38 Attributes for telegram tag

Note: ComId's 1-1000 are reserved for special purpose (see Table 59).

10.2.5.1. MD Parameters

```
<device>
    <bus-interface-list>
        <bus-interface attributes>
                :
            <telegram attributes >
                    :
                <md-parameter attributes/>
                    :
            </telegram>
                :
        </bus-interface>
    </bus-interface-list>
</device>
```

Name	Data Type	Optional/ Required	Description
confirm-timeout	UINT32	Optional	Confirm time-out time in µs Default value will be used at absence of the tag, see Table 37.
reply-timeout	UINT32	Optional	Response time-out time in µs Default value will be used at absence of the tag, see Table 37.
marshall	STRING	Optional	ON/OFF = enable/disable internal marshalling/unmarshalling
callback	STRING	Optional	ON/OFF = enable/disable callback
protocol	STRING	Optional	UDP/TCP

Table 39 Attributes for md-parameter tag

10.2.5.2. PD Parameters

```
<device>
    <bus-interface-list>
        <bus-interface attributes>
                :
            <telegram attributes >
                    :
                <pd-parameter attributes/>
                    :
            </telegram>
              :
        </bus-interface>
           :
    </bus-interface-list>
</device>
```

Name	Data Type	Optional/ Required	Description
timeout	UINT32	Optional	Timeout value in µs, before considering received process data as invalid. Disabled if 0 or no specified value. Default value will be used at absence of the tag, see Table 35.
validity-behaviour	STRING	Optional	Behaviour when received process data is invalid. ZERO = zero values KEEP = keep the last value Default value will be used at absence of the tag, see .
cycle	UINT32	Required	Cycle time in µs, describing how often a process data should be transmitted. TRDP will round this value to a multiple of the cycle time of the TRDP process thread. Default value will be used at absence of the tag, see Table 35.
redundant	UINT32	Optional	>0 if process data is redundant, i.e. it should be transmitted in leader mode, and not in follower mode in a redundant system. See also chapter 8.10.9. Default value = 0 (no redundancy);
marshall	UINT32	Optional	ON/OFF - enable/disable internal marshalling/unmarshalling Default value will be used at absence of the tag, see Table 35.
callback	STRING	Optional	ON/OFF = enable/disable callback
offset-address	UINT16	Optional	Offset-address for PD in traffic store for ladder topology

Table 40 Attributes for pd-parameter tag

10.2.5.3. Source Parameters

```
<device>
    <bus-interface-list>
        <bus-interface attributes>
                :
            <telegram attributes >
                :
                <source attributes>
                   <sdt-parameter attributes />
                </source>
                :
            </telegram>
            :
        </bus-interface>
         :
    </bus-interface-list>
</device>
```

Name	Data Type	Optional/ Required	Description
id	UINT32	Required	Source identifier.
uri1	STRING	Required	Source URI for process and message data. Used for PD filtering at receiver side to only receive data from a specific end device(s) or for MD as information provided to the receiving side. Syntax : [[user@]host] **Note:** If not specified here or overridden in the subscribe call, then no filtering for PD used. If not specified here or overridden in any MD send call, the URI (host part) of the own device is used. **Note:** Only URI strings that are resolved as unicast IP addresses can be used.
uri2	STRING	Optional	For process data a second source URI string can be given for source filtering, e.g. for redundant devices. Syntax : [host] **Note:** If not specified here or overridden in the subscribe call, then no filtering for PD used. If not specified here or overridden in any MD send call, the URI (host part) of the own device is used. **Note:** Only URI strings that are resolved as unicast IP addresses can be used.
name	STRING	Optional	Optional name for the connection

Table 41 Attributes for source tag

10.2.5.4. Destination Parameters

```
<device>
    </bus-interface-list>
        <bus-interface attributes>
                :
            <telegram attributes >
                    :
               <destination attributes
                   <sdt-parameter attributes />
               </destination>
                    :
            </telegram>
              :
        </bus-interface>
           :
    </bus-interface-list>
</device>
```

Name	Data Type	Optional/ Required	Description
id	UINT32	Required	Destination identifier.
uri	STRING	Required	Destination URI for process and message data. Syntax: [user@]host If not specified here it has to be set/overridden in any send/publish call. The URI user part is used in the MD header frame to address a URI listener. The URI host part is used to resolve the destination IP address used for sending and at receiver side to check if the IP address is a multicast address that has to be joined.
name	STRING	Optional	Optional name for the connection

Table 42 Attributes for destination tag

10.2.5.5. SDT Parameters

```
<device>
   <bus-interface-list>
      <bus-interface attributes>
                :
         <telegram attributes >
                :
             <source attributes
                <sdt-parameter attributes />
             </source>
                :
             <destination attributes
                <sdt-parameter attributes />
             </destination>
                :
         </telegram>
             :
      </bus-interface>
   </bus-interface-list>
</device>
```

The tag “sdt-parameter” is optional.

Name	Data Type	Optional/ Required	Description
smi1	UINT32	Required	Safe message identifier – unique for this message at consist level
smi2	UINT32	Optional	Safe message identifier for a redundant device – unique for this message at consist level
udv	UINT16	Required	User data version
rx-period	UINT16	Required	Sink cycle time
tx-period	UINT16	Required	Source cycle time
n-rxsafe	UINT8	Optional	Timeout cycles
n-guard	UINT16	Optional	Initial timeout cycles
cm-thr	UINT32	Optional	Channel monitoring threshold

Table 43 Attributes for sdt-parameter tag

Name	Value
smi2	0
n-rxsafe	3
n-guard	100
cm-thr	10

Table 44 Default values for sdt-parameter tag

10.3. Mapped Device Parameters

There might be the requirement to have the same configuration for several identical devices (e.g. the different door controllers of a consist). This is supported by the mapped device tag, containing the differences for the specific mapped devices.

```
<device>
   <device-configuration>
      :
   </device-configuration>
   <mapped-device-list>
      <mapped-device attributes>
         <mapped-bus-interface attributes>
            <mapped-telegram attributes>
                  :
            </mapped-telegram attributes>
                  :
         </mapped-bus-interface>
               :
      </mapped device>
            :
   </mapped-device-list>
         :
</device>
```

The tags “mapped-device-list” and “mapped-device” are optional.

All attributes that can be specified for the tag “mapped-device” are described in the following table.

.

Name	Data Type	Optional/ Required	Description
host-name	STRING	Required	Device name
leader-name	STRING	Optional	Leader name, optional for redundant devices

Table 45 Attributes for mapped-device tag

10.3.1. Mapped Bus Interface Parameters

For each bus interface of a mapped device the different attributes regarding the process data, sources, destinations and related SDT parameters of the telegrams can be specified.

```
<device>
   <device-configuration>
      :
   </device-configuration>
   <mapped-device list>
      <mapped-device attributes>
         <mapped-bus-interface attributes>
            <mapped-telegram attributes>
               <mapped-pd-parameter attributes>
               <mapped-source attributes>
                  <mapped-sdt-parameter attributes>
               </mapped-source>
                     :
               <mapped-destination attributes>
                  <mapped-sdt-parameter attributes>
               </mapped-destination>
                     :
            </mapped-telegram>
                  :
         </mapped-bus-interface>
               :
      </mapped device>
   </mapped-device-list>
</device>
```

The tag “mapped-bus-interface” is optional.

All attributes that can be specified for the tag “mapped-telegram” are described in the following table.

Name	Data Type	Optional/ Required	Description
name	STRING	Required	Name for this interface
host-ip	UINT32	Optional	IP address for this interface
leader-ip	UINT32	Optional	Leader IP address in case of redundancy

Table 46 Attributes for mapped-bus-interface tag

The tag “mapped-telegram” is optional.

All attributes that can be specified for the tag “mapped-telegram” are described in the following table.

Name	Data Type	Optional/ Required	Description
com-id	UINT32	Required	ComId of the mapped telegram
name	STRING	Optional	Optional different name for the telegram

Table 47 Attributes for mapped-telegram tag

The tag “mapped-pd-parameter” is optional.

Name	Data Type	Optional/ Required	Description
offset-address	UINT16	Optional	Offset-address for PD in traffic store for ladder topology

Table 48 Attributes for mapped-pd-parameter tag

The tag “mapped-source” is optional.

All attributes that can be specified for the tag “mapped-source” are described in the following table.

Name	Data Type	Optional/ Required	Description
Id	UINT32	Required	Identifier of the source of the telegram
uri1	STRING	Required	Source URI for process and message data. Used for PD filtering at receiver side to only receive data from a specific end device(s) or for MD as information provided to the receiving side. For process data the source URI string can include up to two comma separated URI strings, e.g. redundant devices. Syntax: [[user@]host[, host]] **Note:** If not specified here or overridden in the subscribe call, then no filtering for PD used. If not specified here or overridden in any MD send call, the URI (host part) of the own device is used. **Note:** Only URI strings that are resolved as unicast IP addresses can be used.
uri2	STRING	Optional	For process data a second source URI string can be given for source filtering, e.g. for redundant devices. Syntax : [host] **Note:** If not specified here or overridden in the subscribe call, then no filtering for PD used. If not specified here or overridden in any MD send call, the URI (host part) of the own device is used. **Note:** Only URI strings that are resolved as unicast IP addresses can be used.
name	STRING	Optional	Optional different name for the connection

Table 49 Attributes for mapped-source tag

The tag “mapped-destination” is optional.

All attributes that can be specified for the tag “mapped-source” are described in the following table.

Name	Data Type	Optional/ Required	Description
Id	UINT32	Required	Identifier of the destination of the telegram
uri	STRING	Required	Destination URI for process and message data. Syntax: [user@]host If not specified here it has to be set/overridden in any send/publish call. The URI user part is used in the MD header frame to address a URI listener. The URI host part is used to resolve the destination IP address used for sending and at receiver side to check if the IP address is a multicast address that has to be joined.
name	STRING	Optional	Optional different name for the connection

Table 50 Attributes for mapped-destination tag

The tag “mapped-sdt-parameter” is optional.

All attributes that can be specified for the tag “mapped-sdt parameters” are described in the following table.

Name	Data Type	Optional/ Required	Description
smi1	UINT32	Required	Required for SDT telegrams.
smi2	UINT32	Optional	Required for redundant SDT telegrams.

Table 51 Attributes for mapped-sdt-parameter tag

10.4. Communication Parameters (ComPar)

The Communication Parameter information describes in which way the communication should take place. In most cases one set of parameters is sufficient but it is possible to specify specific communication parameters for special situations.

A Communication Parameter is described in the XML file as a “com-parameter” tag.

Structure:

```
<device>
   <com-parameter-list>
      <com-parameter attributes/>
      :
   </com-parameter-list>
</device>
```

All attributes that can be specified for the tag are described in Table 52.

Name	Data Type	Optional/ Required	Description
id	UINT32	Required	ID for communication parameter **Note:** Only the first configuration of a com-parameter-id will be considered.
qos	UINT32	Required	Quality of Service, what priority level data should be sent with, 0..7
ttl	UINT32	Optional	Time To Live, how many jumps a message should live, 0..255, default = 64.
retries	UINT32	Optional	Number of retries used for MD request

Table 52 Attributes for com-parameter tag

10.4.1. Default Communication Parameters

Most communication can be done with a small set of communication parameters. To simplify for the TRDP user there are always two sets of default communication parameters available.

Com-parameter-id	qos	time-to-live	retries	Description
1	5	64	0	Suitable for PD communication
2	3	64	3	Suitable for MD communication

Table 53 Default communication parameters

10.5. DataSet Parameters

Data communication over the IP Train is done with *DataSets*.

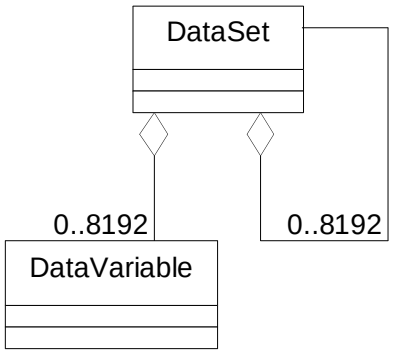

Figure 17 DataSet structure

A DataSet is a container of data items, like a *structure* in C language. A DataSet can contain up to 8192 DataVariables or other DataSets. A *DataVariable* is data of one of the basic data types (UINT8, REAL32 etc.), see chapter 4.2 Each DataVariable or DataSet can be single or multiple instances. Multiple instances (arrays) can be of fix or variable size.

The maximum depth of a dataset within a dataset is 5.

The DataSet configuration information describes all datasets to be used in data exchange. Many Exchange Parameters can use one single DataSet.

The DataSet describes the structure of the data, i.e. which data variables and other datasets it contain. By use of this information TRDP can send and receive data in such a way that it complies with the data formats required on the TCN. Applications do not have to do any further adjustment to be able to use the data.

A DataSet is described in the XML file as a “data-set” tag. Inside this there are “element” tags.

Structure:

```
<device>
   <data-set-list>
       <data-set attributes >
          <element attributes />
          :
       </data-set>
       :
   </data-set-list>
</device>
```

All attributes that can be specified for the tags are described in Table 54.

Name	Data Type	Optional/ Required	Description
name	STRING	Optional	Required for tooling and debugging
id	UINT32	Required	ID for dataset **Note:** Only the first found configuration of a data-set-id will be considered.

Table 54 Attributes for data-set tag

10.5.1. DataSet Element

To be able to perform marshalling of datasets, all end devices must have information about structure and content of the datasets. This information is stored in a Dataset Formatting Table in the configuration database. This table describes for each data item of the DataSet its *type* and *array size*. For visualization purposes each element can have the additional attributes unit, scale and offset.

Each element of a DataSet can be either a DataVariable or DataSet. Up to 8192 elements are possible.

```
<device>
   <data-set-list>
       <data-set attributes >
          <element attributes />
          :
       </data-set>
       :
   </data-set-list>
</device>
```

Name	Data Type	Optional/ Required	Description
name	STRING	Optional	Required for tooling and debugging
type	STRING	Required	Data type of data variable (INT8, INT16 ..., see Table 3) or dataset identifier (1...)
array-size	UINT32	Optional	0 = Array with dynamic size 1 = Single >1 Number of instances in array Default = 1
unit	STRING	Optional	Unit text for visualisation purposes.
scale	FLOAT	Optional	Factor for visualisation purposes (val = scale*x+offset).
offset	INT32	Optional	Offset for visualisation purposes (val = scale*x+offset).

Table 55 Attributes for element tag

10.5.1.1. DataSet Element "type"

If the type is another DataSet the *Type* is the ID of the dataset (positive integer, 1..).

For a DataVariable the *Type* tells the marshalling software the size of the data but also how to treat it. A 4-byte string should not be treated the same way as 4-byte integer or real. The Datatype constants currently available are described in Table 3 on page 22.

The table describes all data types that can be used on the TCN. For IEC-61131 software only a sub-set of types with fixed size can be used.

Note: The C++ type "BOOL" is often not resolved into a UINT8 variable by the compiler because of that here BOOL8 is defined and used.

10.5.1.2. DataSet Element "array-size"

The *array-size* describes the number of instances for the data item.

Array-size	Instances
1	Single
2..n	Array, fixed size 2..n
0	Array, dynamic size

Table 56 Use of element array size

A dynamic sized array must be preceded by a UINT8, UNIT16 or UINT32, containing current size of the array, i.e. number of items in the array (not number of bytes). See also example in chapter 10.5.2.4.

Dynamic sized arrays can only be used for message data and not for process data.

10.5.2. Examples of DataSets

The examples below show different types of datasets with example of included DataVariables and DataSets, single and multiple instances, fixed or variable.

10.5.2.1. DataSet with DataVariables

This dataset contains single instances of basic data types.

C Declaration	Dataset Formatting Table		
	Type	Number	Comment
`struct DS57 {`			
`UINT32 a;`	UINT32	1	Single
`UINT8 b;`	UINT8	1	Single
`INT16 c;`	INT16	1	Single
`REAL32 d;`	REAL32	1	Single
`};`			

XML configuration example:

```
<device>
   <data-set-list>
       <data-set id=”57” >
          <element type=”UINT32” array-size=”1” name=”a” />
          <element type=”UINT8” array-size=”1” name=”b” />
          <element type=”INT16” array-size=”1” name=”c” />
          <element type=”REAL32” array-size=”1” name=”d” />
       </data-set>
       :
   </data-set-list>
</device>
```

10.5.2.2. DataSet with other DataSet

In this example a dataset DS3 contains another dataset DS2.

To specify that a dataset contain another dataset, the column for Type is set to the ID of that dataset. The marshalling software will recognize this by the positive value. Codes for basic data types are negative.

C Declaration	Dataset Formatting Table		
	Type	Number	Comment
`struct DS2 {`			
`UINT32 a1;`	UINT32	1	Single
`INT32 b1;`	INT32	1	Single
`INT32 c1;`	INT32	1	Single
`};`			
`struct DS3 {`			
`UINT32 a;`	UINT32	1	Single
`INT32 b;`	INT32	1	Single
`struct DS2 c;`	ID of DS2	1	Single
`};`			

XML configuration example:

```
<device>
   <data-set-list>
       <data-set id=”2” >
          <element type=”UINT32” array-size=”1” name=”a1” />
          <element type=”INT32” array-size=”1” name=”b1” />
          <element type=”INT32” array-size=”1” name=”c1” />
       </data-set>
       <data-set id=”3 >
          <element type=”UINT32” array-size=”1” name=”a” />
          <element type=”INT32” array-size=”1” name=”b” />
          <element type=”2” array-size=”1” name=”c” />
       </data-set>
       :
   </data-set-list>
</device>
```

10.5.2.3. DataSet with Fixed Sized Arrays

In this example a dataset contains multiple instances of DataVariables and DataSets, i.e. arrays of data.

C Declaration	**Dataset Formatting Table**		
	Type	Number	Comment
`struct DS4 {`			
`INT32 a[17];`	INT32	17	Array of integers
`struct DS2 b[3];`	ID of DS2	3	Array of datasets
`};`			

The first data item is an array of 17 instances of INT32;

The second data item is an array of 3 instances of dataset DS2.

XML configuration example:

```
<device>
   <data-set-list>
       <data-set id=”2” >
          <element type=”UINT32” array-size=”1” name=”a1” />
          <element type=”INT32” array-size=”1” name=”b1” />
          <element type=”INT32” array-size=”1” name=”c1” />
       </data-set>
       <data-set id=”4” >
          <element type=”INT32” array-size=”17” name=”a” />
          <element type=”2” array-size=”3” name=”b” />
       </data-set>
       :
   </data-set-list>
</device>
```

10.5.2.4. DataSet with Dynamic sized arrays

In this example a dataset contains instances of arrays, but the size of the arrays is not predefined in the configuration database but dynamically specified at runtime.

A dynamically sized array must be preceded with a size of type UINT8, UINT16 or UINT32 that contains the current size of the array. These must be loaded with current size at run-time. Size should be set to number of items in the array.
Dynamic size of arrays means that there is no information in the configuration database about the size of the array.

Note: Dynamic sized datasets can only be used in MD communication, not in PD communication.

Declaration	Dataset Formatting Table		
	Type	Number	Comment
`struct DS5 {`			
`UINT32 aSize;`	UINT32	1	Size of array b (=5)
`INT32 b[5];`	INT32	0	
`};`			

Declaration	Dataset Formatting Table		
	Type	Number	Comment
`struct DS6 {`			
`UINT32 aSize;`	UINT32	1	Size of array b (=5)
`INT32 b[5];`	INT32	0	
`UINT16 cSize;`	UINT16	1	Size of array d (=9)
`struct DS2 d[9];`	ID of DS2	0	
`};`			

XML configuration example:

```
<device>
   <data-set-list>
      <data-set id=”2” >
         <element type=”UINT32” array-size=”1” name=”a1” />
         <element type=”INT32” array-size=”1” name=”b1” />
         <element type=”INT32” array-size=”1” name=”c1” />
      </data-set>
      <data-set id=”5” >
         <element type=”UINT32” array-size=”1” name=”a” />
         <element type=”IN32” array-size=”0” name=”b” />
      </data-set>
      <data-set id=”6” >
         <element type=”UINT32” array-size=”1” name=”a” />
         <element type=”INT32” array-size=”0” name=”b” />
         <element type=”UIN16” array-size=”1” name=”c” />
         <element type=”2” array-size=”0” name=”d” />
      </data-set>
      :
   </data-set-list>
</device>
```

The example of structure for data set 6 in the figure above is not very useful unless there are several similar structure used for sending with the same dataset ID. When this type of structure is used where a dynamic array is followed by any other variable type the size variables has to be set to exactly the size of the corresponding array. See the code example below for a more useful way to use this type of dynamic dataset. Another possibility, as for data set 5, is to only use one dynamic array in the end of the dataset then the size variable can be set to the current used size of the array.

Code example for dataset 6:

```
char buf[2000];
UINT16 *p;
UINT32 len;

p = (UINT16 *) buf;         /* Set pointer to start of buffer */
*p++ = currAsize;           /* Store current size of array a */
memcpy(p, aBuffer, currAsize * sizeof(INT32));
p = (char *) p + currAsize * sizeof(INT32); /* Move pointer */

*p++ = currBsize;           /* Store current size of array b */
memcpy(p, bBuffer, currBsize * sizeof(struct DS2));
p = (char *) p + currBsize * sizeof(struct DS2);

len = (char *) p - buf;     /* length of buffer, no. of bytes */
```

10.5.2.5. DataSet for Transparent Communication

In some cases applications want to exchange data without any interference of marshalling software, e.g. for uploading of recorded binary data. This can be achieved by sending a dataset that contains only one data item: a dynamic sized array of bytes, unsigned 8 bit integers or for by sending a comId without any specified dataset. As mentioned before dynamic data set can only be sent as message data.

Declaration	**Dataset Formatting Table**		
	Type	Number	Comment
`struct DS5 {`			
`  UINT32 bufSize;`	UINT32	1	Size of array buf (=n)
`  UINT8 buf[n];`	UINT8	0	Array of bytes
`};`			

The sending buffer `buf` must be preceded with the size of the buffer to send, loaded with the number of items in `buf`, i.e. 'n'.

10.6. Controlling the Trace Output

To control the trace output there is a "debug" tag with four attributes "level-trdp", "info-trdp" "file-size" and "file-name" in the configuration XML file. Since the implementation of the logging functionality itself is up to the application, this part of the XML-File is an optional service for the application which can be used or not.

```
<device>
   <debug attributes />
</device>
```

The tag "debug" is optional. All attributes that can be specified for the tag "debug" are described in the following table.

Name	Data Type	Optional/ Required	Description
level	STRING	Optional	Debug output level: Blank, “” OR “ ” - turned off E OR e - errors I OR i - information W OR w - warnings
info	STRING	optional	Debug info: Blank , “” or “ “ - Show only error/warning text D,T,d OR t - Show date and time F OR f - Show source file name and line C OR c - Show category
file-size	UINT32	Optional	The maximum file size for storing trace outputs before overwriting old values.
file-name	STRING	Optional	“” OR “ “ means only standard IO output except for TRDP daemon, otherwise filename and path needs to be given.

Table 57 Attributes for debug tag

Name	Value
Level	“E”
Info	“”
file-size	65536
file-name	“” (stdio)

Table 58 Default values for debug-parameter

Example:

```
<debug file-name=”c:/temp/debug.txt” file-size=”16000” level=”w” info=”LFD”/>
```

10.7. Populating the Configuration Database

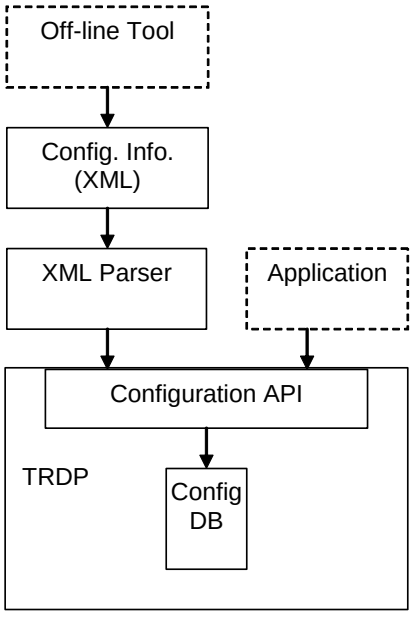

Figure 18 Configuration data tool chain

To be able to function correctly TRDP needs configuration data in its configuration database. This can be populated in two ways: with configuration file(s) at start up or by application via an API during run-time.

10.7.1. XML Configuration File

An off-line configuration tool can generate the TRDP XML configuration file(s). The file(s) can be downloaded to the end device and stored in its file system.

More than one configuration file may be used. The XML configuration file(s) is parsed by the *XML Parser* at call of the related methods described in chapter 14.1. The retrieved data can be used in the call of `tlc_openSession (C)` or `tlc::openSession (C++)`.

10.7.2. Example XML Configuration File

Examples of XML configuration file are provided in the archive file trdp-example.zip.

10.8. Reserved ComId's

The ComId's 1-1000 are reserved for system level. The usage is listed in the following table. For application use only ComId's >1000 shall be used.

ComId	Use
10	Echo
31	Request to retrieve TRDP statistics information
35	Reply of TRDP global statistics (see Table 101)
37	Reply of PD subscription statistics (see Table 102)
39	Reply of PD publish statistics (see Table 103)
41	Reply of Redundancy statistics (see Table 105)
43	Reply of Join
45	Statistics reply of UDP MD listener statistics (see Table 104)
46	Statistics reply of TCP MD listener statistics (see Table 104)
100	PD push Inauguration state and topo count telegram
101	PD pull request telegram to retrieve dynamic train configuration information
102	PD pull reply telegram with dynamic train configuration information
103	MD request telegram to retrieve static consist and car information
104	MD reply telegram with static consist and car information
105	MD request telegram to retrieve device information for a given consist/car/device
106	MD reply telegram with device information for a given consist/car/device
107	MD request telegram to retrieve consist and car properties for a given consist/car
108	MD reply telegram with consist and car properties for a given consist/car
109	MD request telegram to retrieve device properties for a given consist/car/device
110	MD reply telegram with device properties for a given consist/car/device
111	MD request telegram for manual insertion of a given consist/car
112	MD reply telegram for manual insertion of a given consist/car
120..129	Switch Control&Monitoring Interface
125	MD Data (Version) Request Telegram
126	MD Counter Telegram
127	MD Dynamic Configuration Telegram
128	MD Dynamic Configuration Telegram Response
129	PD Dynamic Configuration Telegram (redundant TS to TS IPC)
400..415	SDTv2 validation test
1000	Test telegram with data

Table 59 Reserved ComId's

10.9. Reserved Dataset Id's

The dataset id's 1-1000 are reserved for system level. The usage is listed in the following table. For application use only dataset id's >1000 shall be used.

Dataset Id	Use
1..30	Basic data types of Table 3
31	Request to retrieve TRDP statistics information
32	Memory statistics (see Table 98)
33	PD global statistics (see Table 99)
34	MD global statistics (see Table 100)
35	TRDP global statistics (see Table 101)
36	PD subscription statistics (see Table 102)
37	PD subscription statistics array
38	PD publish statistics (see Table 103)
39	PD publish statistics array
40	Redundancy statistics (see Table 105)
41	Redundancy statistics array
42	Join statistics MD
43	Join statistics MD array
44	Listener statistics (see Table 104)
45	Listener statistics array
100	PD push Inauguration state and topo count telegram
101	PD pull request telegram to retrieve dynamic train configuration information
102	PD pull reply telegram with dynamic train configuration information
103	MD request telegram to retrieve static consist and car information
104	MD reply telegram with static consist and car information
105	MD request telegram to retrieve device information for a given consist/car/device
106	MD reply telegram with device information for a given consist/car/device
107	MD request telegram to retrieve consist and car properties for a given consist/car
108	MD reply telegram with consist and car properties for a given consist/car
109	MD request telegram to retrieve device properties for a given consist/car/device
110	MD reply telegram with device properties for a given consist/car/device
111	MD request telegram for manual insertion of a given consist/car
112	MD reply telegram for manual insertion of a given consist/car
120..129	Switch Control&Monitoring Interface
125	MD Data (Version) Request Telegram
126	MD Counter Telegram
127	MD Dynamic Configuration Telegram
128	MD Dynamic Configuration Telegram Response
129	PD Dynamic Configuration Telegram (redundant TS to TS IPC)
400..415	SDTv2 validation test
1000	Test dataset

Table 60 Reserved Dataset Id's

11. Marshalling

11.1. Marshalling Rules

The *marshalling rules* describe how data types should be converted from the internal representation to the representation on the TCN.

Note: In case marshalling is disabled then there is no conversion done and the data is sent on the wire as it is stored in the memory of the host.

11.1.1. Data Representation

All data on the TCN are Big Endian.

REAL32 is transmitted as IEEE 32 bit float format.

STRING is transmitted as zero-terminated 8 bit ASCII characters.

UNICODE16 string is transmitted as array of UINT16 (with marshalling). If a UINT16 zero is found it is considered end of string. BOM (Byte Order Marker) is supported.

11.1.2. Packing

Packing is performed with the following rules:

1. Only relevant data bytes are transmitted, padding bytes added by the compiler are suppressed.
2. For strings of type STRING or UNICODE16 only characters up to and including the zero-termination are transmitted.
3. Resulting datagram is padded with trailing bytes to give a total size of a multiple of 32 bits. These bytes are set zero.

11.1.3. Example

A dataset is described in following way:

```
struct {
   INT8  a;
   INT16 b;
   INT8  c, d;
   INT32 e;
   INT8  f;
};
```

On a 32 bit, little-endian processor (e.g. Intel PC) this is probably stored in memory as:

Address	**0**	**+1**	**+2**	**+3**
0	a		b (LSB)	b (MSB)
4	c	d		
8	e (LSB, byte 0)	e (byte 1)	e (byte 2)	e (MSB, byte 3)
12	f			

(blank cells = padded bytes, unknown state)

When sending a dataset with TRDP the marshalled datagram will be like this:

MSB			LSB
IP Header			
UDP Header			
RTP Header			
PD Header			
a	b (MSB)	b (LSB)	c
d	e (MSB, byte 0)	e (byte 1)	e (byte 2)
e (LSB, byte 3)	f	P	P
FCS (for DataSet)			

P = padding, set to 0.

Note: On the wire, the MSB is numbered as byte 0 and is sent first in a stream. The most significant bit in a byte is numbered as bit 0. See also [Wire].

Actual byte stream sent:

<all header bytes>
a
b (MSB)
b (LSB)
C
D
e (MSB, byte 0)
e (byte 1)
e (byte 2)
e (LSB, byte 3)
F
P
P
FCS (MSB, byte 0)
FCS (byte 1)
FCS (byte 2)
FCS (LSB, byte 3)

11.2. Marshalling Software

The marshalling software must convert data from the internal representation to the standard representation on the TCN. To its use it has the information in the configuration database, which describes the content and structure of the datasets. This is not completely sufficient. The software must also know how data is handled in the present environment. This can be depending on hardware, compiler and operating system.

The TRDP software can never be 100 % portable but the current implementation automatically takes care of most problems of a port to a new platform. At start up TRDP checks that alignment conditions are as expected. Below is described how TRDP handles different items.

11.2.1. Order of Bytes

In multi-byte data, e.g. INT32, the order in which data is stored could differ. Computers that store Least Significant Part on Least Significant Address are said to be “Little Endian”. Computers that store Most Significant Part on Least Significant Address are said to be “Big Endian”. All data on the TCN shall be Big Endian.

TRDP marshalling automatically handles reordering of Little Endian data.

11.2.2. Alignment

Compilers assign memory for variables and structures to get fast execution speed. This means that subsequent variables not necessarily are positioned on subsequent bytes. Intermediate bytes may be skipped, *padded*. This is called *alignment*.

TRDP marshalling automatically handles alignment, if the environment is using *natural* alignment, which most compilers do.

Natural alignment means that a variable is aligned depending on the size of the variable. A 1-byte variable is aligned to any address, a 2-byte variable is aligned to an even address, and a 4-byte variable is aligned to an address that is a multiple of 4. Datatypes with sizes larger than 4 bytes are aligned according the return value of the *alignof()* operator.

A structure is aligned depending on the largest alignment of any of its components.

11.2.3. Data Representation

Some data types can be represented differently. One example is *floats*, where there are alternative methods of storing decimal values.

TRDP marshalling currently does not perform any change in data representation.

12. Error Handling

TRDP can in some situations not fulfil its task, e.g. because of lack of memory, illegal buffer sizes. In such cases TRDP has two ways of indicating this to the calling application:

- Method return code (C / C++ methods)
- Raising exception (C++ methods)

12.1. Return / Error Codes

Return codes are used in the C / C++ interface. TRDP methods return a code as an integer with values according to following table. As well as exceptions are used in the C++ interface, and are raised if according to following table. The return codes -1 .. -29 are mirrored over from VOS level.

Value	Name	Description
0	TRDP_NO_ERR	No error
-1	TRDP_PARAM_ERR	Parameter missing or out of range
-2	TRDP_INIT_ERR	Call without valid initialisation
-3	TRDP_NOINIT_ERR	Call with invalid handle
-4	TRDP_TIMEOUT_ERR	Timeout
-5	TRDP_NODATA_ERR	Non blocking mode: no data received
-6	TRDP_SOCK_ERR	Socket error / option not supported
-7	TRDP_IO_ERR	Socket IO error, data can't be received/sent
-8	TRDP_MEM_ERR	No more/not enough memory available
-9	TRDP_SEMA_ERR	Semaphore not available
-10	TRDP_QUEUE_ERR	Queue empty
-11	TRDP_QUEUE_FULL_ERR	Queue full
-12	TRDP_MUTEX_ERR	Mutex not available
-13	TRDP_THREAD_ERR	Thread not available
-14	TRDP_INTEGRATION_ERR	Alignment or endianess for selected target wrong
-15	TRDP_NO_CONN_ERR	No TCP connection
-16..-29	Reserved	
-30	TRDP_NO_SESSION_ERR	No valid SessionId
-31	TRDP_SESSION_ABORT_ERR	Session aborted
-32	TRDP_NOSUB_ERR	No subscriber
-33	TRDP_NOPUB_ERR	No publisher
-34	TRDP_NOLIST_ERR	No listener
-35	TRDP_CRC_ERR	CRC check error
-36	TRDP_WIRE_ERR	
-37	TRDP_TOPO_ERR	Topo counter not valid
-38	TRDP_COMID_ERR	No valid ComId
-39	TRDP_STATE_ERR	Call in wrong state
-40	TRDP_APP_TIMEOUT_ERR	Application timeout
-41	TRDP_APP_REPLYTO_ERR	Application reply sent timeout
-42	TRDP_APP_CONFIRMTO_ERR	Application confirm sent timeout
-43	TRDP_REPLYTO_ERR	Reply timeout
-44	TRDP_CONFIRMTO_ERR	Confirm timeout
-45	TRDP_REQCONFIRMTO_ERR	Request confirm timeout
-46	TRDP_PACKET_ERR	Incomplete MD packet

-99	TRDP_UNKNOWN_ERR	Unspecified error

Table 61 Enumeration `TRDP_ERR_T` – Error code definitions for TRDP

The TLC methods may throw an exception of the class “tlc_exception”. The TLP methods may throw an exception of the class “tlp_exception”. The TLM methods may throw an exception of the class “tlm_exception”.

Example of C++ code:

```
try
{
   :
   tlp::subscribe(...);
   :
}
catch (tlp_exception ev)
{
   cout << ev.getErrorString();
}
```

13. TRDP Virtual OS

To be able to handle various operating systems there is an abstraction layer inside TRDP called Vos. This tackles differences for e.g. following features:

- Memory allocation
- Queue handling
- Socket handling
- Mutex handling
- Thread handling
- Debug support
- CRC calculation

Some of the methods in Vos are made public for use by related applications. This includes methods for semaphores, thread start and timers.

Memory allocation is also included. This memory is taken from same memory pool as the rest of TRDP. This should be considered when dimensioning the size of the memory pool.

13.1. VOS Types, Initialisation and Support Functions

13.1.1. Definitions

Value	Name	Description
0	VOS_NO_ERR	No error
-1	VOS_PARAM_ERR	Necessary parameter missing or out of range.
-2	VOS_INIT_ERR	Call without valid initialization.
-3	VOS_NOINIT_ERR	The supplied handle/reference is not valid.
-4	VOS_TIMEOUT_ERR	Timeout
-5	VOS_NODATA_ERR	Non blocking mode: no data received.
-6	VOS_SOCK_ERR	Socket error/option not supported
-7	VOS_IO_ERR	Socket IO error, data can't be received/sent
-8	VOS_MEM_ERR	Not enough/no more memory available
-9	VOS_SEMA_ERR	Semaphore not available
-10	VOS_QUEUE_ERR	Queue empty
-11	VOS_QUEUE_FULL_ERR	Queue full
-12	VOS_MUTEX_ERR	Mutex not available
-13	VOS_THREAD_ERR	Thread creation error
-14	TRDP_INTEGRATION_ERR	Alignment or endianess for selected target wrong
-15	TRDP_NO_CONN_ERR	No TCP connection
-99	VOS_UNKNOWN_ERR	Unknown error

Table 62 Enumeration `VOS_ERR_T` – Error code definitions for VOS

Value	Name	Description
0	VOS_LOG_ERR	This is a critical error
1	VOS_LOG_WARNING	This is a warning.
2	VOS_LOG_INFO	This is an information.
3	VOS_LOG_DBG	This is an additional debug information.

Table 63 Enumeration `VOS_LOG_T` – Log type definitions

Type	Name	Description
UINT8	VOS_UUID_T[16]	universal unique identifier according to RFC 4122, time based version

Table 64 Type `VOS_UUID_T` - universal unique identifier according to RFC 4122, time based version

13.1.2. VOS_PRINT_DBG_T

Name	VOS_PRINT_DBG_T	

Synopsis C	`typedef void (*VOS_PRINT_DBG_T)(`
	`  void  *pRefCon,`
	`  VOS_LOG_T  category,`
	`  const CHAR8 *pTime,`
	`  const CHAR8 *pFile,`
	`  UINT16  lineNumber,`
	`  const CHAR8 *pMsgStr);`	

Synopsis C++	`typedef void (*VOS_PRINT_DBG_T)(`
	`  void  *pRefCon,`
	`  VOS_LOG_T  category,`
	`  const CHAR8 *pTime,`
	`  const CHAR8 *pFile,`
	`  UINT16  lineNumber,`
	`  const CHAR8 *pMsgStr);`	

Abstract	Function definition for error/debug output. The function will be called for logging and error message output. The user can decide what kind of info will be logged by filtering the category.	
Parameters	`pRefCon`	Pointer to user context.
	`Category`	Log category acc. Table 63
	`pTime`	Pointer to NULL-terminated string of time stamp
	`pFile`	Pointer to NULL-terminated string of source module name
	`lineNumber`	Line number in source module
	`pMsgStr`	Pointer to NULL-terminated message string
Returns C/C++	None	

13.1.3. vos_snprintf

Name	vos_snprintf	
Synopsis C	`EXT_DECL UINT32 vos_snprintf ( CHAR8 *str, UINT32 size, CHAR8 *pFormat, . . . );`	
Synopsis C++	`EXT_DECL VOS_ERR_T vos::snprintf ( CHAR8 *str, UINT32 size, CHAR8 *pFormat, . . . );`	
Abstract	Safe sprintf() function.	
Parameters	`str`	Destination string
	`size`	Destination string size
	`pFormat`	Format string according printf() definition
	`. . .`	Arguments according printf() definition
Returns C/C++	Number of characters written in the string str.	

13.1.4. vos_printLog

Name	vos_printLog	
Synopsis C	`EXT_DECL void vos_printLog ( CHAR8 *str, UINT32 size, CHAR8 *pFormat, . . . );`	
Synopsis C++	`EXT_DECL void vos::printLog ( CHAR8 *str, UINT32 size, CHAR8 *pFormat, . . . );`	
Abstract	Safe log print function for debug strings calling the debug output function given by the TRDP user .	
Parameters	`level`	Debug level according Table 63
	`pFormat`	Format string according printf() definition
	`. . .`	Arguments according printf() definition
Returns C/C++	None	

13.1.5. vos_init

Name	**vos_init**	

Synopsis C	`EXT_DECL VOS_ERR_T vos_init (`
	`  void  *pRefCon,`
	`  VOS_PRINT_DBG_T  *pDebugOutput);`	

Synopsis C++	`EXT_DECL VOS_ERR_T vos::init (`
	`  void  *pRefCon,`
	`  VOS_PRINT_DBG_T  *pDebugOutput);`	

Abstract	Initialize the vos library. This is used to set the output function for all VOS error and debug output.	
Parameters	`pRefCon`	Pointer to user context
	`pDebugOutput`	Pointer to debug output function.
Returns C/C++	VOS_NO_ERR no error VOS_INIT_ERR not supported	

13.1.6. vos_crc32

Name	**vos_crc32**	

Synopsis C	`EXT_DECL UINT32 vos_crc32 (`
	`  const CHAR8  *pBuf,`
	`  const UINT32  bufSize);`	

Synopsis C++	`EXT_DECL UINT32 vos_crc32 (`
	`  const CHAR8  *pBuf,`
	`  const UINT32  bufSize);`	

Abstract	Calculate CRC for the given buffer and length. For TRDP FCS CRC calculation the CRC32 according to IEEE802.3 with start value 0xffffffff is used.	
Parameters	`pBuf`	Pointer to a buffer to calculate the CRC.
	`bufSize`	Size of the buffer.
Returns C/C++	Calculated CRC	

13.1.7. vos_bsearch

Name	vos_bsearch	

Synopsis C	`EXT_DECL void *vos_bsearch (`
	`  const void  *pKey,`
	`  const void  *pBuf,`
	`  UINT32  num,`
	`  UINT32  size,`
	`  int  (*compare)(`
	`  const  void *,`
	`  const  void *))`	

Synopsis C++	`EXT_DECL void *vos_bsearch (`
	`  const void  *pKey,`
	`  const void  *pBuf,`
	`  UINT32  num,`
	`  UINT32  size,`
	`  int  (*compare)(`
	`  const  void *,`
	`  const  void *))`	

Abstract	Binary search in a sorted array. This is just a wrapper for the standard qsort function	
Parameters	`pKey`	Pointer to a key to search for
	`pBuf`	Pointer to the array to sort
	`num`	number of elements
	`size`	size of one element
	`compare`	Pointer to the compare function, first argument contains is the pointer to the key the second one the pointer to the array element
Returns C/C++	Pointer to found element or NULL	

13.1.8. vos_qsort

Synopsis C	`EXT_DECL void vos_qsort (`
	`  void  *pBuf,`
	`  UINT32  num,`
	`  UINT32  size,`
	`  int  (*compare)(`
	`  const  void *,`
	`  const  void *))`	

Synopsis C++	`EXT_DECL void vos_qsort (`
	`  void  *pBuf,`
	`  UINT32  num,`
	`  UINT32  size,`
	`  int  (*compare)(`
	`  const  void *,`
	`  const  void *))`	

Abstract	Sort an array using quick sort algorithm. This is just a wrapper for the standard qsort function.	
Parameters	`pBuf`	Pointer to the array to sort
	`num`	number of elements
	`size`	size of one element
	`compare`	Pointer to the compare function to compare the arrays given in the two parameters. return -n if arg1 < arg2, return 0 if arg1 == arg2, return +n if arg1 > arg2 where n is an integer != 0
Returns C/C++	Pointer to found element or NULL	

13.2. Memory Allocation and Queue Handling

In a real time system dynamic memory allocation must be handled with care. If not taken care of properly it could lead to non-deterministic behaviour during e.g. garbage collection. TRDP has its own memory allocation component. At start up a large memory area is allocated with a default size or with a size specified in the XML configuration file.

Whenever a program within TRDP needs a dynamic memory area it calls an allocation method, which allocates a fixed size memory block that is equal or greater in size. There is a fixed set of memory block sizes.

Allocation is done in the following steps:

- First try to allocate any returned memory blocks with the closest size equal or greater than requested size
- Then try to create a new block from the free, unblocked memory area with a fixed size from the set of memory block sizes that is equal or greater than requested size
- Then try to allocate any returned memory blocks with a greater size
- Then return error.

At start up there is a number of free memory blocks pre-allocated for some of the fixed set of memory block sizes. This is mainly used to ensure that there are at least a minimum number of blocks of the larger sizes.

If the program does not need the memory area any more it releases it calling the freeing method. This will add this memory block to a list of available memory blocks, which can be reused at later stage.

The memory size is configured via parameters specified in the XML configuration file or by default values.

13.2.1. Definitions

#define VOS_MEM_NBLOCKSIZES 15

Internally memory is allocated always by the 15 pre-configured block sizes.

#define VOS_MEM_PREALLOCATE {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 4, 0, 0}

Default pre-allocation of free memory blocks. To avoid problems with too many small blocks and no large one, specify how many blocks of each size should be pre-allocated (and freed!) to pre-segment the memory area.

Block Index	Block Size	Description
0	48	tlc_init(), tlc_openSession()
1	72	tlm_addListener()
2	128	tlc_openSession(), tlp_publish(), tlp_subscribe(),
3	180	tlm_request(), tlm_notify()
4	256	Block size
5	512	tlc_openSession()
6	1024	tlc_openSession() without MD
7	1480	tlp_publish(), tlp_subscribe()
8	2048	Block size
9	4096	Block size
10	11520	tlc_openSession()
11	16384	Block size
12	32768	Block size
13	65536	Block size
14	1311072	Block size

Table 65 Enumeration `VOS_MEM_BLK_T` **- indicates the memory block size**

Type	Name	Description
void *	VOS_QUEUE_T	Hidden queue handle definition

Table 66 Type `VOS_QUEUE_T` – queue handle

Type	Name	Description
void *	VOS_SHRD_T	Hidden shared memory handle definition

Table 67 Type `VOS_SHRD_T` – shared memory handle

13.2.2. vos_memInit

Name	vos_memInit	

Synopsis C	`VOS_ERR_T vos_memInit(`
	`  UINT8  *pMemoryArea,`
	`  UINT32  size,`
	`  const UINT32  fragMem[VOS_MEM_NBLOCKSIZES]);`	

Synopsis C++	`VOS_ERR_T  vos::memInit (`
	`  UINT8  *pMemoryArea,`
	`  UINT32  size,`
	`  const UINT32  fragMem[VOS_MEM_NBLOCKSIZES]);`	

Abstract	Initialise the provided memory block and prepare it for use with vos_alloc() and vos_dealloc(). The used block sizes can be supplied and will be pre-allocated.	
Parameters	`pMemoryArea`	Pointer to memory area to use
	`size`	size of the provided memory area
	`fragMem`	List of pre-allocate block sizes, used to fragment memory for large blocks
Returns C/C++	VOS_NO_ERR	no error
	VOS_PARAM_ERR	parameter out of range/invalid
	VOS_MEM_ERR	no memory available

13.2.3. vos_memDelete

Name	vos_memDelete	
Synopsis C	`void vos_memDelete(` `  UINT8  *pMemoryArea);`	
Synopsis C++	`void vos:: memDelete(` `  UINT8  *pMemoryArea);`	
Abstract	This will eventually invalidate any previously allocated memory blocks! It should be called last before the application quits. No further access to the memory blocks is allowed after this call.	
Parameters	`pMemoryArea`	Pointer to memory area to use
Returns C/C++		

13.2.4. vos_memAlloc

Name	vos_memAlloc	
Synopsis C	`UINT8 *vos_memAlloc(` `  UINT32  size);`	
Synopsis C++	`UINT8 *vos::memAlloc(` `  UINT32  size);`	
Abstract	Allocate memory of the requested size.	
Parameters	`size`	Size of the requested memory.
Returns C/C++	Pointer to memory area or NULL if no memory available	

13.2.5. vos_memFree

Name	vos_memFree	
Synopsis C	`void vos_memFree(` `  void  *pMemoryArea);`	
Synopsis C++	`void vos::memFree(` `  void  *pMemoryArea);`	
Abstract	Free the memory area.	
Parameters	`pMemoryArea`	Pointer to memory area to use
Returns C/C++		

13.2.6. vos_memCount

Name	vos_memCount	

Synopsis C	`VOS_ERR_T  vos_memCount(`
	`  UINT32  *pAllocatedMemory,`
	`  UINT32  *pFreeMemory,`
	`  UINT32  *pFragMem[VOS_MEM_NBLOCKSIZES]);`	

Synopsis C++	`VOS_ERR_T  vos::memCount (`
	`  UINT32  *pAllocatedMemory,`
	`  UINT32  *pFreeMemory,`
	`  UINT32  *pFragMem[VOS_MEM_NBLOCKSIZES]);`	

Abstract	Return used and available memory (of memory area above)	
Parameters	`pAllocatedMemory`	Pointer to allocated memory size
	`pFreeMemory`	Pointer to free memory size
	`pFragMem`	Pointer to list of used memory blocks
Returns C/C++	VOS_NO_ERR	no error
	VOS_INIT_ERR	module not initialized
	VOS_PARAM_ERR	parameter out of range/invalid

13.2.7. vos_sharedOpen

Name	**vos_ sharedOpen**	

Synopsis C	`VOS_ERR_T vos_sharedOpen (`
	`  const CHAR8 *pKey,`
	`  VOS_SHRD_HNDL_T *pHandle,`
	`  UINT8 **ppMemoryArea,`
	`  UINT32 *pSize);`	

Synopsis C++	`VOS_ERR_T vos::sharedOpen (`
	`  const CHAR8 *pKey,`
	`  VOS_SHRD_HNDL_T *pHandle,`
	`  UINT8 **ppMemoryArea,`
	`  UINT32 *pSize);`	

Abstract	Create a shared memory area or attach to existing one. The first call with the specified key will create a shared memory area with the supplied size and will return a handle and a pointer to that area. If the area already exists, the area will be attached. This function is not available in each target implementation.	
Parameters	`pKey`	Unique identifier (file name)
	`pHandle`	Pointer to the memory handle
	`ppMemoryArea`	Pointer to pointer of the memory area
	`pSize`	Pointer to the size of the memory to allocate, returns the actual attached size
Returns C/C++	VOS_NO_ERR	no error
	VOS_INIT_ERR	module not initialized
	VOS_PARAM_ERR	parameter out of range/invalid
	VOS_MEM_ERR	no memory available

13.2.8. vos_sharedClose

Name	**vos_ sharedClose**	

Synopsis C	`VOS_ERR_T vos_sharedClose (`
	`  VOS_SHRD_T handle,`
	`  UINT8 *pMemoryArea);`	

Synopsis C++	`VOS_ERR_T vos::sharedClose (`
	`  VOS_SHRD_T handle,`
	`  UINT8 *pMemoryArea);`	

Abstract	Close connection to the shared memory area. If the area was created by the calling process, the area will be closed (freed). If the area was attached, it will be detached. This function is not available in each target implementation.	
Parameters	`handle`	Memory handle
	`pMemoryArea`	Pointer to the memory area
Returns C/C++	VOS_NO_ERR	no error
	VOS_INIT_ERR	module not initialized
	VOS_NOINIT_ERR	invalid handle
	VOS_PARAM_ERR	invalid handle/parameter out of range/invalid

13.2.9. vos_queueCreate

Name	vos_ queueCreate	

Synopsis C	`VOS_ERR_T vos_queueCreate (`
	`const CHAR8 *pKey,`
	`VOS_QUEUE_T *pQueue,`
	`UINT32 maxNoMsg,`
	`UINT32 maxLength) ;`	

Synopsis C++	`VOS_ERR_T vos::queueCreate (`
	`const CHAR8 *pKey,`
	`VOS_QUEUE_T *pQueue,`
	`UINT32 maxNoMsg,`
	`UINT32 maxLength) ;`	

Abstract	Initialize a message queue and return a handle for further calls	
Parameters	`pKey`	Unique identifier (file name)
	`pQueue`	Pointer to returned queue handle
	`maxNoMsg`	maximum number of messages
	`maxLength`	maximum size of messages
Returns C/C++	VOS_NO_ERR	no error
	VOS_INIT_ERR	module not initialized
	VOS_NOINIT_ERR	invalid handle
	VOS_PARAM_ERR	parameter out of range/invalid
	VOS_INIT_ERR	not supported
	VOS_QUEUE_ERR	error creating queue

13.2.10. vos_queueDestroy

Name	vos_ queueDestroy	
Synopsis C	`VOS_ERR_T vos_queueDestroy (` `VOS_QUEUE_T queue);`	
Synopsis C++	`VOS_ERR_T vos::queueDestroy (` `VOS_QUEUE_T queue);`	
Abstract	Delete a message queue and free all resources used by this queue	
Parameters	`queue`	Queue handle
Returns C/C++	VOS_NO_ERR	no error
	VOS_INIT_ERR	module not initialized
	VOS_NOINIT_ERR	invalid handle
	VOS_PARAM_ERR	parameter out of range/invalid

13.2.11. vos_queueSend

Name	**vos_ queueSend**	

Synopsis C	`VOS_ERR_T vos_queueSend (`
	`  VOS_QUEUE_T  queue,`
	`  const UINT8  *pMsg,`
	`  UINT32  size);`	

Synopsis C++	`VOS_ERR_T vos::queueSend (`
	`  VOS_QUEUE_T  queue,`
	`  const UINT8  *pMsg,`
	`  UINT32  size);`	

Abstract	Put a message in the queue	
Parameters	`queue`	Queue handle
	`pMsg`	Pointer to the message
	`size`	Size of the message
Returns C/C++	VOS_NO_ERR	no error
	VOS_INIT_ERR	module not initialized
	VOS_NOINIT_ERR	invalid handle
	VOS_PARAM_ERR	invalid handle/parameter out of range/invalid
	VOS_QUEUE_ERR	queue is full

13.2.12. vos_queueReceive

Name	**vos_ queueReceive**	

Synopsis C	`VOS_ERR_T vos_queueReceive (`
	`  VOS_QUEUE_T  queue,`
	`  const UINT8  **ppMsg,`
	`  UINT32  *pSize,`
	`  UINT32  *timeout);`	

Synopsis C++	`VOS_ERR_T vos::queueReceive (`
	`  VOS_QUEUE_T  queue,`
	`  const UINT8  *ppMsg,`
	`  UINT32  *pSize`
	`  UINT32  *timeout);`	

Abstract	Get a message from the queue	
Parameters	`queue`	Queue handle
	`ppMsg`	Pointer to the message pointer
	`pSize`	Pointer to the size of the message
	`timeout`	Maximum waiting time for a message in µsec
Returns C/C++	VOS_NO_ERR	no error
	VOS_INIT_ERR	module not initialized
	VOS_NOINIT_ERR	invalid handle
	VOS_PARAM_ERR	invalid handle/parameter out of range/invalid
	VOS_QUEUE_ERR	queue is empty

13.2.13. vos_strnicmp

Name	vos_ strnicmp	
Synopsis C	`INT32 vos_strnicmp ( const CHAR8 *pStr1, const CHAR8 *pStr2, UINT32 count );`	
Synopsis C++	`INT32 vos::strnicmp ( const CHAR8 *pStr1, const CHAR8 *pStr2, UINT32 count );`	
Abstract	Case insensitive string compare.	
Parameters	`pStr1`	Null terminated string to compare
	`pStr2`	Null terminated string to compare
	`count`	Maximum number of characters to compare
Returns C/C++	0 - equal <0 - string1 less than string 2 >0 - string 1 greater than string 2	

13.2.14. vos_strncpy

Name	vos_ strncpy	
Synopsis C	`void _T vos_strncpy ( const CHAR8 *pStrDst, const CHAR8 *pStrSrc, UINT32 count );`	
Synopsis C++	`void vos::strncpy ( const CHAR8 *pStrDst, const CHAR8 *pStrSrc, UINT32 count );`	
Abstract	String copy function with fixed length..	
Parameters	`pStrDst`	Destination string
	`pStrSrc`	Null terminated string to copy
	`count`	Maximum number of characters to copy
Returns C/C++	None	

13.3. Socket Handling

13.3.1. Definitions

Type	Name	Description
`UINT8`	`qos`	quality/type of service 0…7
`UINT8`	`ttl`	time to live for unicast (default 64)
`UINT8`	`ttlMulticast`	time to live for multicast (default 64)
`BOOL8`	`reuseAddrPort`	allow reuse of address and port
`BOOL8`	`nonBlocking`	use non blocking calls

Table 68 Structure `VOS_SOCK_OPT_T` – socket options

13.3.2. vos_sockInit

Name	vos_ sockInit	
Synopsis C	`VOS_ERR_T vos_sockInit (void);`	
Synopsis C++	`VOS_ERR_T vos::sockInit (void);`	
Abstract	Initialize the socket library. Must be called once before any other call of the socket library.	
Parameters		
Returns C/C++	VOS_NO_ERR VOS_SOCK_ERR	no error sockets not supported

13.3.3. vos_dottedIP

Name	vos_dottedIP	
Synopsis C	`UINT32 vos_dottedIP(const CHAR8 *pDottedIP);`	
Synopsis C++	`UINT32 vos::dottedIP(const CHAR8 *pDottedIP);`	
Abstract	Convert dotted IP address to UINT32	
Parameters	`pDottedIP`	Pointer to dotted IP address
Returns C/C++	IP address in host endianess	

13.3.4. vos_ipDotted

Name	vos_ipDotted	
Synopsis C	`const CHAR8 *vos_ipDotted(UINT32 ipAddress);`	
Synopsis C++	`const CHAR8 *vos::ipDotted(UINT32 ipAddress);`	
Abstract	Convert UINT32 to dotted IP address	
Parameters	`ipAddress`	IP address in host endianess
Returns C/C++	Dotted IP address	

13.3.5. vos_isMulticast

Name	vos_isMulticast	
Synopsis C	`BOOL8 vos_isMulticast(UINT32 ipAddress);`	
Synopsis C++	`BOOL8 vos::isMulticast(UINT32 ipAddress);`	
Abstract	Check if given IP address is a multicast address	
Parameters	`ipAddress`	IP address
Returns C/C++	TRUE FALSE	address is multicast address address is not a multicast address

13.3.6. vos_getInterfaces

Name	vos_getInterfaces	

Synopsis C	`VOS_ERR_T vos_getInterfaces(`
	`  UINT32 *pAddrCnt,`
	`  VOS_IF_REC_T ifAddrs[]);`	

Synopsis C++	`VOS_ERR_T vos::getInterfaces(`
	`  UINT32 *pAddrCnt,`
	`  VOS_IF_REC_T ifAddrs[]);`	

Abstract	Get interface addresses..	
Parameters	pAddrCnt	IN: Pointer to number of elements in array of interface records OUT: Pointer to number of interface records read
	ifAddrs	Array of interface records
Returns C/C++	VOS_NO_ERR	no error
	VOS_INIT_ERR	sockets not initialized

Type	Name	Description
CHAR8	name[VOS_MAX_IF_NAME_SIZE]	Interface adapter name
VOS_IP4_ADDR_T	ipAddr	IP address
VOS_IP4_ADDR_T	netMask	Subnet mask
UINT8	mac[VOS_MAC_SIZE]	Interface adapter MAC address

Table 69 Type VOS_IF_REC_T – interface record definition

13.3.7. vos_sockGetMac

Name	vos_sockGetMac	
Synopsis C	`VOS_ERR_T vos_sockGetMac(UINT8 pMAC[VOS_MAC_SIZE]);`	
Synopsis C++	`VOS_ERR_T vos::sockGetMac(UINT8 pMAC[VOS_MAC_SIZE]);`	
Abstract	Get MAC address of the default interface..	
Parameters	pMAC	Pointer to MAC address
Returns C/C++	VOS_NO_ERR	no error
	VOS_INIT_ERR	sockets not initialized

13.3.8. vos_sockOpenUDP

Name	**vos_ sockOpenUDP**	

Synopsis C	`VOS_ERR_T vos_sockOpenUDP (`
	`  INT32  *pSock,`
	`  const VOS_SOCK_OPT_T  *pOptions);`	

Synopsis C++	`VOS_ERR_T  vos::sockOpenUDP (`
	`  INT32  *pSock,`
	`  const VOS_SOCK_OPT_T  *pOptions);`	

Abstract	Create an UDP socket. Return a socket descriptor for further calls. The socket options are optional and can be applied later. **Note:** Some target systems might not support every option.	
Parameters	`pSock`	Pointer to socket descriptor returned
	`pOptions`	Pointer to socket options (optional)
Returns C/C++	VOS_NO_ERR	no error
	VOS_INIT_ERR	module not initialized
	VOS_NOINIT_ERR	invalid handle
	VOS_PARAM_ERR	parameter out of range/invalid, pSock == NULL
	VOS_SOCK_ERR	socket not available or option not supported

13.3.9. vos_sockOpenTCP

Name	**vos_ sockOpenTCP**	

Synopsis C	`VOS_ERR_T vos_sockOpenTCP (`
	`  INT32  *pSock,`
	`  const VOS_SOCK_OPT_T  *pOptions);`	

Synopsis C++	`VOS_ERR_T  vos::sockOpenTCP (`
	`  INT32  *pSock,`
	`  const VOS_SOCK_OPT_T  *pOptions);`	

Abstract	Create a TCP socket. Return a socket descriptor for further calls. The socket options are optional and can be applied later. **Note:** Some target systems might not support each option.	
Parameters	`pSock`	Pointer to socket descriptor returned
	`pOptions`	Pointer to socket options (optional)
Returns C/C++	VOS_NO_ERR	no error
	VOS_INIT_ERR	module not initialized
	VOS_NOINIT_ERR	invalid handle
	VOS_PARAM_ERR	parameter out of range/invalid, pSock == NULL
	VOS_SOCK_ERR	socket not available or option not supported

13.3.10. vos_sockClose

Name	vos_ sockClose	
Synopsis C	`VOS_ERR_T vos_ sockClose ( INT32 sock);`	
Synopsis C++	`VOS_ERR_T vos::sockClose ( INT32 sock);`	
Abstract	Close a socket. Release any resource initialised by this socket.	
Parameters	`sock`	Socket descriptor
Returns C/C++	VOS_NO_ERR	no error
	VOS_INIT_ERR	module not initialized
	VOS_NOINIT_ERR	invalid handle
	VOS_PARAM_ERR	parameter out of range/invalid

13.3.11. vos_sockSetOptions

Name	vos_ sockSetOptions	
Synopsis C	`VOS_ERR_T vos_ sockSetOptions ( INT32 sock, const VOS_SOCK_OPT_T *pOptions);`	
Synopsis C++	`VOS_ERR_T vos::sockSetOptions ( INT32 sock, const VOS_SOCK_OPT_T *pOptions);`	
Abstract	Set socket options. **Note:** Some target systems might not support each option.	
Parameters	`sock`	Socket descriptor
	`pOptions`	Pointer to socket options (optional)
Returns C/C++	VOS_NO_ERR	no error
	VOS_INIT_ERR	module not initialized
	VOS_NOINIT_ERR	invalid handle
	VOS_PARAM_ERR	parameter out of range/invalid
	VOS_SOCK_ERR	socket not available or option not supported

13.3.12. vos_sockJoinMC

Name	**vos_ sockJoinMC**	

Synopsis C	`VOS_ERR_T vos_sockJoinMC (`
	`  INT32  sock,`
	`  UINT32  mcAddress,`
	`  UINT32  ipAddress);`	

Synopsis C++	`VOS_ERR_T  vos::sockJoinMC (`
	`  INT32  sock,`
	`  UINT32  mcAddress,`
	`  UINT32  ipAddress);`	

Abstract	Join a multicast group. **Note:** Some target systems might not support this option.	
Parameters	`sock`	Socket descriptor
	`mcAddress`	Multicast address to join
	`ipAddress`	Depicts interface on which to join, default 0 for any
Returns C/C++	VOS_NO_ERR	no error
	VOS_INIT_ERR	module not initialized
	VOS_NOINIT_ERR	invalid handle
	VOS_PARAM_ERR	parameter out of range/invalid
	VOS_SOCK_ERR	socket not available or option not supported

13.3.13. vos_sockLeaveMC

Name	**vos_ sockLeaveMC**	

Synopsis C	`VOS_ERR_T vos_sockLeaveMC (`
	`  INT32  sock,`
	`  UINT32  mcAddress,`
	`  UINT32  ipAddress);`	

Synopsis C++	`VOS_ERR_T  vos::sockLeaveMC (`
	`  INT32  sock,`
	`  UINT32  mcAddress,`
	`  UINT32  ipAddress);`	

Abstract	Leave a multicast group. **Note:** Some target systems might not support this option.	
Parameters	`sock`	Socket descriptor
	`mcAddress`	Multicast address to leave
	`ipAddress`	Depicts interface on which to leave, default 0 for any
Returns C/C++	VOS_NO_ERR	no error
	VOS_INIT_ERR	module not initialized
	VOS_NOINIT_ERR	invalid handle
	VOS_PARAM_ERR	parameter out of range/invalid
	VOS_SOCK_ERR	socket not available or option not supported

13.3.14. vos_sockSendUDP

Name	vos_ sockSendUDP	

Synopsis C	`VOS_ERR_T vos_sockSendUDP (`
	`  INT32 sock,`
	`  const UINT8 *pBuffer,`
	`  UINT32 *pSize,`
	`  UINT32 ipAddress,`
	`  UINT16 port);`	

Synopsis C++	`VOS_ERR_T vos::sockSendUDP (`
	`  INT32 sock,`
	`  const UINT8 *pBuffer,`
	`  UINT32 *pSize,`
	`  UINT32 ipAddress,`
	`  UINT16 port);`	

Abstract	Send UDP data to the given address and port.	
Parameters	`sock`	Socket descriptor
	`pBuffer`	Pointer to the data to send
	`pSize`	Pointer to the size of the data buffer. Returns the size of the data sent
	`ipAddress`	Destination IP address
	`port`	Destination port
Returns C/C++	VOS_NO_ERR	no error
	VOS_PARAM_ERR	parameter out of range/invalid
	VOS_IO_ERR	data could not be sent
	VOS_BLOCK_ERR	call would have blocked blocking mode

13.3.15. vos_sockReceiveUDP

Name	**vos_ sockReceiveUDP**	

Synopsis C	`VOS_ERR_T vos_sockReceiveUDP (`
	`  INT32  sock,`
	`  UINT8  *pBuffer,`
	`  UINT32  *pSize,`
	`  UINT32  *pSrcIpAddress,`
	`  UINT16  *pSrcIpPort,`
	`  UINT32  *pDstIpAddr);`	

Synopsis C++	`VOS_ERR_T  vos::sockReceiveUDP (`
	`  INT32  sock,`
	`  UINT8  *pBuffer,`
	`  UINT32  *pSize,`
	`  UINT32  *pSrcIpAddress,`
	`  UINT16  *pSrcIpPort,`
	`  UINT32  *pDstIpAddr);`	

Abstract	Receive UDP message data from the given port with the given destination IP address. The caller must provide a sufficient sized buffer. If the supplied buffer is smaller than the bytes received, *pSize will reflect the number of copied bytes and the call should be repeated until *pSize is 0 (zero). If the socket was created in blocking-mode (default), then this call will block and will only return if data has been received or the socket was closed or an error occurred. If called in non-blocking mode, and no data is available, VOS_NODATA_ERR will be returned.	
Parameters	`sock`	Socket descriptor
	`pBuffer`	Pointer to the application data buffer
	`pSize`	Pointer to the size of the data buffer. Returns the size of the received data.
	`pSrcIpAddress`	Pointer to source IP address
	`pSrcIpPort`	Pointer to source IP port
	`pDstIpAddress`	Pointer to destination IP address
Returns C/C++	VOS_NO_ERR	no error
	VOS_PARAM_ERR	parameter out of range/invalid
	VOS_IO_ERR	data could not be read
	VOS_MEM_ERR	resource error
	VOS_NODATA_ERR	no data
	VOS_BLOCK_ERR	call would have blocked blocking mode

13.3.16. vos_sockBind

Name	vos_ sockBind	

Synopsis C	`VOS_ERR_T vos_sockBind(`
	`  INT32  sock,`
	`  UINT32  ipAddress,`
	`  UINT16  port);`	

Synopsis C++	`VOS_ERR_T  vos::sockBind (`
	`  INT32  sock,`
	`  UINT32  ipAddress,`
	`  UINT16  port);`	

Abstract	Bind a socket to an address and port.	
Parameters	`Sock`	Socket descriptor
	`ipAddress`	Source IP address to receive from, 0 for any
	`Port`	Source port to receive from
Returns C/C++	VOS_NO_ERR	no error
	VOS_PARAM_ERR	parameter out of range/invalid
	VOS_SOCK_ERR	binding failed

13.3.17. vos_sockListen

Name	vos_ sockListen	

Synopsis C	`VOS_ERR_T vos_sockListen(`
	`  INT32  sock,`
	`  UINT32  backlog);`	

Synopsis C++	`VOS_ERR_T  vos::sockListen (`
	`  INT32  sock,`
	`  UINT32  backlog);`	

Abstract	Listen for incoming TCP connection.	
Parameters	`sock`	Socket descriptor
	`backlog`	Maximum connection attempts if system is busy
Returns C/C++	VOS_NO_ERR	no error
	VOS_PARAM_ERR	parameter out of range/invalid
	VOS_SOCK_ERR	receiving not possible

13.3.18. vos_sockAccept

Name	**vos_ sockAccept**	

Synopsis C	`VOS_ERR_T vos_sockAccept(`
	`  INT32  sock,`
	`  INT32  pSock,`
	`  UINT32  *pIpAddress,`
	`  UINT16  *pPort);`	

Synopsis C++	`VOS_ERR_T  vos::sockAccept (`
	`  INT32  sock,`
	`  INT32  pSock,`
	`  UINT32  *pIpAddress,`
	`  UINT16  *pPort);`	

Abstract	Accept an incoming TCP connection.	
Parameters	`sock`	Socket descriptor
	`pSock`	Pointer to new socket descriptor to be returned
	`pIpAddress`	Source IP address to receive on, 0 for any
	`pPort`	Source port to receive on
Returns C/C++	VOS_NO_ERR	no error
	VOS_PARAM_ERR	parameter out of range/invalid

13.3.19. vos_sockConnect

Name	**vos_ sockConnect**	

Synopsis C	`VOS_ERR_T vos_sockConnect(`
	`  INT32  sock,`
	`  UINT32  ipAddress,`
	`  UINT16  port);`	

Synopsis C++	`VOS_ERR_T  vos:: sockConnect (`
	`  INT32  sock,`
	`  UINT32  ipAddress,`
	`  UINT16  port);`	

Abstract	Open a TCP connection.	
Parameters	`sock`	Socket descriptor
	`ipAddress`	Destination IP address
	`port`	Destination port
Returns C/C++	VOS_NO_ERR	no error
	VOS_PARAM_ERR	parameter out of range/invalid
	VOS_IO_ERR	Input/output error

13.3.20. vos_sockSendTCP

Name	vos_ sockSendTCP	

Synopsis C	`VOS_ERR_T vos_sockSendTCP (`
	`  INT32  sock,`
	`  const UINT8 *pBuffer,`
	`  UINT32  size);`	

Synopsis C++	`VOS_ERR_T  vos:: sockSendTCP (`
	`  INT32  sock,`
	`  const UINT8 *pBuffer,`
	`  UINT32  size);`	

Abstract	Send TCP data to the given socket.	
Parameters	`sock`	Socket descriptor
	`pBuffer`	Pointer to the data to send
	`size`	size of the data to send
Returns C/C++	VOS_NO_ERR	no error
	VOS_PARAM_ERR	parameter out of range/invalid
	VOS_IO_ERR	data could not be sent
	VOS_NOCONN_ERR	no TCP connection
	VOS_BLOCK_ERR	call would have blocked in blocking mode

13.3.21. vos_sockReceiveTCP

Name	vos_ sockReceiveTCP	

Synopsis C	`VOS_ERR_T vos_sockReceiveTCP (`
	`  INT32  sock,`
	`  const UINT8 *pBuffer,`
	`  UINT32  *pSize);`	

Synopsis C++	`VOS_ERR_T  vos:: sockReceiveTCP (`
	`  INT32  sock,`
	`  const UINT8 *pBuffer,`
	`  UINT32  *pSize);`	

Abstract	Receive TCP data. The caller must provide a sufficient sized buffer. If the supplied buffer is smaller than the bytes received, *pSize will reflect the number of copied bytes and the call should be repeated until *pSize is 0 (zero). If the socket was created in blocking-mode (default), then this call will block and will only return if data has been received or the socket was closed or an error occurred. If called in non-blocking mode, and no data is available, VOS_NODATA_ERR will be returned.	
Parameters	`sock`	Socket descriptor
	`pBuffer`	Pointer to the applications data buffer
	`pSize`	Pointer to the received data size
Returns C/C++	VOS_NO_ERR	no error
	VOS_PARAM_ERR	parameter out of range/invalid
	VOS_BLOCK_ERR	call would have blocked in blocking mode
	VOS_IO_ERR	data could not be read
	VOS_MEM_ERR	resource error
	VOS_NODATA_ERR	no data in non-blocking

13.3.22. vos_sockSetMulticastIf

Name	vos_ sockSetMulticastIf	
Synopsis C	`VOS_ERR_T vos_sockSetMulticastIf ( INT32 sock, UINT32 usingMulticastIfAddress );`	
Synopsis C++	`VOS_ERR_T vos:: sockSetMulticastIf ( INT32 sock, UINT32 usingMulticastIfAddress );`	
Abstract	Set the interface to be used for multicast	
Parameters	`sock`	Socket descriptor
	`usingMulticastIfAddress`	IP address of interface to be used
Returns C/C++	VOS_NO_ERR	no error
	VOS_PARAM_ERR	parameter out of range/invalid

13.4. Thread and Mutex Handling

13.4.1. Definitions

Name	Description
VOS_THREAD_POLICY_OTHER	Default for the target system
VOS_THREAD_POLICY_FIFO	First come, first serve
VOS_THREAD_POLICY_RR	Round robin

Table 70 Enumeration VOS_THREAD_POLICY_T - thread policy matching pthread/Posix

Value	Name	Description
0	VOS_SEMA_EMPTY	Semaphore empty
1	VOS_SEMA_FULL	Semaphore full

Table 71 Enumeration VOS_SEMA_STATE_T – initial state of a semaphore

Type	Name	Description
UINT32	tv_sec	full seconds
UINT32	tv_usec	micro seconds (max. value 999999)

Table 72 Structure VOS_TIME_T – select/timeval compatible time definition

Type	Name	Description
UINT8	VOS_THREAD_PRIORITY_T	Thread priority range from 1 (highest) to 255 (lowest), 0 default of the target system.

Table 73 Type VOS_THREAD_PRIORITY_T – thread priority

Type	Name	Description
void *	VOS_THREAD_T	Hidden thread handle definition

Table 74 Type VOS_THREAD_T – thread handle

Type	Name	Description
void *	VOS_MUTEX_T	Hidden mutex handle definition

Table 75 Type VOS_MUTEX_T – mutex handle

Type	Name	Description
void *	VOS_SEMA_T	Hidden semaphore handle definition

Table 76 Type VOS_SEMA_T – semaphore handle

13.4.2. VOS_THREAD_FUNC_T

Name	VOS_THREAD_FUNC_T	
Synopsis C	`typedef void(__cdecl *VOS_THREAD_FUNC_T)(` `  void *pArg);`	
Abstract	Thread function prototype.	
Parameters	`pArg`	*arguments.*
Returns C	void	

13.4.3. vos_threadInit

Name	vos_threadInit	
Synopsis C	`VOS_ERR_T vos_threadInit (void);`	
Synopsis C++	`VOS_ERR_T  vos::threadInit (void);`	
Abstract	Initialize the thread library. Must be called once before any other call of this library.	
Parameters		
Returns C/C++	VOS_NO_ERR VOS_INIT_ERR	no error threading not supported

13.4.4. vos_threadCreate

Name	**vos_ threadCreate**	

Synopsis C	`VOS_ERR_T vos_threadCreate (`
	`  VOS_THREAD_T  *pThread,`
	`  const CHAR8  *pName,`
	`  VOS_THREAD_POLICY_T  policy,`
	`  VOS_THREAD_PRIORITY_T  priority,`
	`  UINT32  interval,`
	`  UINT32  stackSize,`
	`  VOS_THREAD_FUNC_T  pFunction,`
	`  void  *pArgumen ts);`	

Synopsis C++	`VOS_ERR_T vos::threadCreate (`
	`  VOS_THREAD_T  *pThread,`
	`  const CHAR8  *pName,`
	`  VOS_THREAD_POLICY_T  policy,`
	`  VOS_THREAD_PRIORITY_T  priority,`
	`  UINT32  interval,`
	`  UINT32  stackSize,`
	`  VOS_THREAD_FUNC_T  pFunction,`
	`  void  *pArgumen ts);`	

Abstract	Create a thread and return a thread handle for further requests. Not each parameter may be supported by all target systems!	
Parameters	`pThread`	Pointer to returned thread handle
	`pName`	Pointer to name of the thread (optional)
	`policy`	Scheduling policy (FIFO, Round Robin or other)
	`priority`	Scheduling priority (1...255 (highest), default 0)
	`interval`	Interval for cyclic threads in µs (optional)
	`stackSize`	Minimum stacksize, default 0: 16kB
	`pFunction`	Pointer to the thread function parameters
	`pArguments`	Pointer to the received data size
Returns C/C++	VOS_NO_ERR no error VOS_INIT_ERR module not initialized/no threads available VOS_PARAM_ERR parameter out of range/invalid	

13.4.5. vos_threadTerminate

Name	**vos_threadTerminate**	
Synopsis C	`VOS_ERR_T vos_threadTerminate (` `  VOS_THREAD_T  thread);`	
Synopsis C++	`VOS_ERR_T vos::threadTerminate (` `  VOS_THREAD_T  thread);`	
Abstract	This call will terminate the thread with the given identifier and release all resources. Depending on the underlying architectures, it may just block until the thread ran out.	
Parameters	`thread`	Thread handle
Returns C/C++		

13.4.6. vos_threadIsActive

Name	vos_ threadIsActive	
Synopsis C	`VOS_ERR_T vos_threadIsActive (` `  VOS_THREAD_T  thread);`	
Synopsis C++	`VOS_ERR_T  vos::threadIsActive (` `  VOS_THREAD_T  thread);`	
Abstract	This call will return VOS_NO_ERR if the thread is still active, VOS_PARAM_ERR in case it ran out.	
Parameters	`thread`	Thread handle
Returns C/C++	VOS_NO_ERR VOS_INIT_ERR VOS_NOINIT_ERR	no error module not initialized invalid handle

13.4.7. vos_threadDelay

Name	vos_ threadDelay	
Synopsis C	`VOS_ERR_T vos_threadDelay (` `  UINT32  delay);`	
Synopsis C++	`VOS_ERR_T  vos::threadDelay (` `  UINT32  delay);`	
Abstract	Delay the execution of the current thread by the given delay in µs	
Parameters	`delay`	Delay in µs
Returns C/C++	VOS_NO_ERR VOS_INIT_ERR VOS_PARAM_ERR	no error module not initialized parameter out of range/invalid

13.4.8. vos_getTime

Name	vos_ getTime	
Synopsis C	`void vos_getTime (` `  VOS_TIME_T  *pTime);`	
Synopsis C++	`void vos::getTime (` `  VOS_TIME_T  *pTime);`	
Abstract	Return the current time in sec and µs	
Parameters	`pTime`	Pointer to time value
Returns C/C++		

13.4.9. vos_clearTime

Name	vos_ clearTime	
Synopsis C	`void vos_clearTime (` `  VOS_TIME_T  *pTime);`	
Synopsis C++	`void vos::clearTime (` `  VOS_TIME_T  *pTime);`	
Abstract	Clear the time stamp	
Parameters	`pTime`	Pointer to time value
Returns C/C++		

13.4.10. vos_addTime

Name	vos_ addTime	

Synopsis C	`void vos_addTime (`
	`  VOS_TIME_T  *pTime,`
	`  const VOS_TIME_T  *pAdd);`	

Synopsis C++	`void vos::addTime (`
	`  VOS_TIME_T  *pTime,`
	`  const VOS_TIME_T  *pAdd);`	

Abstract	Add the time	
Parameters	`pTime`	Pointer to time value
	`pAdd`	Pointer to time value to add
Returns C/C++		

13.4.11. vos_subTime

Name	vos_ subTime	

Synopsis C	`void vos_subTime (`
	`  VOS_TIME_T  *pTime,`
	`  const VOS_TIME_T  *pSub);`	

Synopsis C++	`void vos::subTime (`
	`  VOS_TIME_T  *pTime,`
	`  const VOS_TIME_T  *pSub);`	

Abstract	Substract the time	
Parameters	`pTime`	Pointer to time value
	`pSub`	Pointer to time value to substract
Returns C/C++		

13.4.12. vos_mulTime

Name	vos_ mulTime	

Synopsis C	`void vos_mulTime (`
	`  VOS_TIME_T  *pTime,`
	`  const VOS_TIME_T  *pMul);`	

Synopsis C++	`void vos::mulTime (`
	`  VOS_TIME_T  *pTime,`
	`  const VOS_TIME_T  *pMul);`	

Abstract	Multiply the time	
Parameters	`pTime`	Pointer to time value
	`pMul`	Pointer to time value to multiply
Returns C/C++		

13.4.13. vos_divTime

Name	vos_ divTime	

Synopsis C	`void vos_divTime (`
	`  VOS_TIME_T *pTime,`
	`  UINT32  divisor);`	

Synopsis C++	`void vos::divTime (`
	`  VOS_TIME_T *pTime,`
	`  UINT32  divisor);`	

Abstract	Divide the time	
Parameters	`pTime`	Pointer to time value
	`divisor`	Divisor
Returns C/C++		

13.4.14. vos_cmpTime

Name	vos_ cmpTime	

Synopsis C	`INT32 vos_divTime (`
	`  const VOS_TIME_T  *pTime,`
	`  const VOS_TIME_T  *pCmp);`	

Synopsis C++	`INT32 vos::divTime (`
	`  const VOS_TIME_T  *pTime,`
	`  const VOS_TIME_T  *pCmp);`	

Abstract	Compare the time	
Parameters	`pTime`	Pointer to time value
	`pCmp`	Pointer to time value to compare

Returns C/C++	`0  pTime == pCmp`
	`-1  pTime < pCmp`
	`1  pTime > pCmp`	

13.4.15. vos_getTimeStamp

Name	vos_ getTimeStamp	
Synopsis C	`CHAR8 *vos_getTimeStamp (void);`	
Synopsis C++	`CHAR8 * vos::getTimeStamp (void);`	
Abstract	Return the current time in readable format as yyyymmdd-hh:mm:ss:ms. Depending on the used OS / hardware the time might not be a real-time stamp but relative from start of system.	
Parameters	`none`	
Returns C/C++	VOS_NO_ERR no error VOS_INIT_ERR module not initialized VOS_PARAM_ERR parameter out of range/invalid	

13.4.16. vos_getUuid

Name	vos_getUuid	
Synopsis C	`void vos_getUuid ( VOS_UUID_T pUuid);`	
Synopsis C++	`void vos::getUuid ( VOS_UUID_T pUuid);`	
Abstract	Get a unique identifier	
Parameters	`delay`	Delay in µs
Returns C/C++		

13.4.17. vos_mutexCreate

Name	vos_mutexCreate	
Synopsis C	`VOS_ERR_T vos_mutexCreate ( VOS_MUTEX_T *pMutex);`	
Synopsis C++	`VOS_ERR_T vos::mutexCreate ( VOS_MUTEX_T *pMutex);`	
Abstract	Create a mutex. Return a mutex handle. The mutex will be available at creation.	
Parameters	`pMutex`	Pointer to mutex handle
Returns C/C++	VOS_NO_ERR VOS_INIT_ERR VOS_PARAM_ERR VOS_MUTEX_ERR	no error module not itialised pMutex == NULL no mutex available

13.4.18. vos_mutexDelete

Name	vos_mutexDelete	
Synopsis C	`void vos_mutexDelete ( VOS_MUTEX_T mutex);`	
Synopsis C++	`void vos::mutexDelete ( VOS_MUTEX_T mutex);`	
Abstract	Delete a mutex.	
Parameters	`mutex`	Mutex handle
Returns C/C++		

13.4.19. vos_mutexLock

Name	vos_mutexLock	
Synopsis C	`VOS_ERR_T vos_mutexLock ( VOS_MUTEX_T mutex);`	
Synopsis C++	`VOS_ERR_T vos::mutexLock ( VOS_MUTEX_T mut ex);`	
Abstract	Take a mutex. Wait for the mutex to become available (lock).	
Parameters	`mutex`	Mutex handle
Returns C/C++	VOS_NO_ERR VOS_INIT_ERR VOS_NOINIT_ERR	no error module not initialized invalid handle

13.4.20. vos_mutexTryLock

Name	vos_mutexTryLock	
Synopsis C	`VOS_ERR_T vos_mutexTryLock (` `  VOS_MUTEX_T mutex);`	
Synopsis C++	`VOS_ERR_T vos::mutexTryLock (` `  VOS_MUTEX_T mutex);`	
Abstract	Try to take a mutex. If mutex can't be taken VOS_MUTEX_ERR is returned.	
Parameters	`mutex`	Mutex handle
Returns C/C++	VOS_NO_ERR VOS_INIT_ERR VOS_NOINIT_ERR VOS_MUTEX_ERR	no error module not initialized invalid handle no mutex available

13.4.21. vos_mutexUnlock

Name	vos_mutexUnlock	
Synopsis C	`void vos_mutexUnlock (` `  VOS_MUTEX_T mutex);`	
Synopsis C++	`void vos::mutexUnlock (` `  VOS_MUTEX_T mutex);`	
Abstract	Release a mutex.	
Parameters	`mutex`	Mutex handle
Returns C/C++		

13.4.22. vos_semaCreate

Name	vos_semaCreate	

Synopsis C	`VOS_ERR_T vos_semaCreate (`
	`  VOS_SEMA_T  *pSema,`
	`  VOS_SEMA_STATE_T  initialState);`	

Synopsis C++	`VOS_ERR_T  vos::semaCreate (`
	`  VOS_SEMA_T  *pSema,`
	`  VOS_SEMA_STATE_T  initialState);`	

Abstract	Create a semaphore. Return a semaphore handle. Depending on the initial state the semaphore will be available or not on creation.	
Parameters	`pSema`	Pointer to semaphore handle
	`initialState`	The initial state of the semaphore
Returns C/C++	VOS_NO_ERR VOS_INIT_ERR VOS_PARAM_ERR VOS_SEMA_ERR	no error module not initialized parameter out of range/invalid no semaphore available

13.4.23. vos_semaDelete

Name	**vos_semaDelete**	
Synopsis C	`void vos_semaDelete (` `  VOS_SEMA_T  sema);`	
Synopsis C++	`void vos::semaDelete (` `  VOS_SEMA_T  sema);`	
Abstract	Delete a semaphore. This will eventually release any processes waiting for the semaphore.	
Parameters	`sema`	Semaphore handle
Returns C/C++		

13.4.24. vos_semaTake

Name	**vos_semaTake**	

Synopsis C	`VOS_ERR_T vos_semaTake (`
	`  VOS_SEMA_T  sema,`
	`  UINT32  timeout);`	

Synopsis C++	`VOS_ERR_T  vos::semaTake (`
	`  VOS_SEMA_T  sema,`
	`  UINT32  timeout);`	

Abstract	Take a semaphore. Try to get (decrease) a semaphore.	
Parameters	`sema`	Semaphore handle
	`timeout`	Timeout in µs
Returns C/C++	VOS_NO_ERR	no error
	VOS_INIT_ERR	module not initialized
	VOS_NOINIT_ERR	invalid handle
	VOS_PARAM_ERR	parameter out of range/invalid
	VOS_SEMA_ERR	no semaphore available

13.4.25. vos_semaGive

Name	**vos_semaGive**	
Synopsis C	`void_T vos_semaGive (` `  VOS_SEMA_T  sema);`	
Synopsis C++	`void vos::semaGive (` `  VOS_SEMA_T  sema);`	
Abstract	Give a semaphore back. Try to release (increase) a semaphore.	
Parameters	`sema`	Semaphore handle
Returns C/C++		

14. Utilities

14.1. TRDP Read XML Configuration API

TRDP support configuration via an XML file like it is described in chapter 10. To minimize the overhead for reading the XML configuration file(s), the tau_prepareXmlDoc() and tau_freeXmlDoc() methods are provided to access a configuration file.

14.1.1. tau_prepareXmlDoc

Name	tau_prepareXmlDoc	

Synopsis C	`TRDP_ERR_T tau_prepareXmlDoc(`
	`  const CHAR8 *pFileName,`
	`  TRDP_XML_DOC_HANDLE_T *pDocHnd`
	`);`	

Synopsis C++	`TRDP_ERR_T tau::prepareXmlDoc(`
	`  const CHAR8 *pFileName,`
	`  TRDP_XML_DOC_HANDLE_T *pDocHnd`
	`);`	

Abstract	Load XML file into DOM tree, prepare XPath context.	
Parameters	`pFileName`	`Path and filename of the xml configuration file`
	`pDocHnd`	`Handle of the parsed XML file`
Returns	0 if OK, !=0 if error	

Type	Name	Description
`void`	`pXmlDocument`	Pointer to parsed XML document
`void`	`pRootElement`	Pointer to the document root element
`void`	`pXPathContext`	Pointer to prepared XPath context

Table 77 Structure `TRDP_XML_DOC_HANDLE_T` – TRDP process configuration parameters

14.1.2. tau_freeXmlDoc

Name	tau_freeXmlDoc	

Synopsis C	`TRDP_ERR_T tau_freeXmlDoc(`
	`  TRDP_XML_DOC_HANDLE_T *pDocHnd`
	`);`	

Synopsis C++	`TRDP_ERR_T tau::freeXmlDoc(`
	`  TRDP_XML_DOC_HANDLE_T *pDocHnd`
	`);`	

Abstract	Free all the memory allocated by tau_prepareXmlDoc.	
Parameters	`pDocHnd`	`Handle of the parsed XML file`
Returns	0 if OK, !=0 if error	

14.1.3. tau_readXmlConfig

Name	tau_readXmlDeviceConfig	

Synopsis C	`TRDP_ERR_T tau_readXmlDeviceConfig(`
	`  const TRDP_XML_DOC_HANDLE_T *pDocHnd,`
	`  TRDP_MEM_CONFIG_T *pMemConfig,`
	`  TRDP_DBG_CONFIG_T *pDbgConfig,`
	`  UINT32 numComPar,`
	`  TRDP_COM_PAR_T **ppComPar,`
	`  UINT32 *pNumIf,`
	`  TRDP_IF_CONFIG_T **ppIfConfig`
	`);`	

Synopsis C++	`TRDP_ERR_T tau::readXmlDeviceConfig(`
	`  const TRDP_XML_DOC_HANDLE_T *pDocHnd,`
	`  TRDP_MEM_CONFIG_T *pMemConfig,`
	`  TRDP_DBG_CONFIG_T *pDbgConfig,`
	`  UINT32 numComPar,`
	`  TRDP_COM_PAR_T **ppComPar,`
	`  UINT32 *pNumIf,`
	`  TRDP_IF_CONFIG_T **ppIfConfig`
	`);`	

Abstract	Read the communication relevant parameters (except data set configuration) out of the configuration file	
Parameters	pDocHnd	Handle of the parsed XML file
	pMemConfig	Memory configuration
	pDbgConfig	Pointer to debug printout configuration for application use
	numComPar	Number of configured com parameters
	ppComPar	Pointer to an array of com parameters
	pNumIf	Number of configured interfaces
	ppIfConfig	Pointer to an array of interface parameter sets
Returns	0 if OK, !=0 if error	

Value	Name	Description
0	TRDP_DBG_DEFAULT	Printout default
0x01	TRDP_DBG_OFF	Printout off
0x02	TRDP_DBG_ERR	Printout only error messages
0x04	TRDP_DBG_WARN	Printout only warning and error messages
0x08	TRDP_DBG_INFO	Printout only info, warning and error messages
0x10	TRDP_DBG_TIME	Printout time stamp
0x20	TRDP_DBG_LOC	Printout file name and line
0x40	TRDP_DBG_CAT	Printout category (INFO, WARN, ERR)

Table 78 Enumeration `TRDP_DBG_OPTION_T` – Debug printout options

Type	Name	Description
TRDP_DEBUG_OPTION_T	option	Debug printout options for application use
UINT32	maxFileSize	Maximal file size
TRDP_FILE_NAME_T	fileName	Debug file name and path

Table 79 Structure `TRDP_DBG_CONFIG_T` – Debug printout configuration

Type	Name	Description
UINT32	Id	Com parameter identifier
TRDP_SEND_PARAM_T	sendParam	Send parameter (TTL, QoS, retries)

Table 80 Structure `TRDP_COM_PAR_T` – Common communication parameter

Type	Name	Description
TRDP_LABEL_T	ifName	Interface name
UINT8	networkId	Used network on the device (1…4)
TRDP_IP_ADDR	hostIp	Host IP address
TRDP_IP_ADDR	leaderIp	Leader IP address dependant on redundancy concept

Table 81 Structure `TRDP_IF_CONFIG_T` – TRDP interface configuration parameters

14.1.4. tau_readXmlInterfaceConfig

Name	tau_readXmlInterfaceConfig	

Synopsis C	`TRDP_ERR_T tau_readXmlInterfaceConfig(`
	`  const TRDP_XML_DOC_HANDLE_T *pDocHnd,`
	`  const CHAR8 *pIfName,`
	`  TRDP_PROCESS_CONFIG_T *pProcessConfig,`
	`  TRDP_PD_CONFIG_T *pPdConfig,`
	`  TRDP_MD_CONFIG_T *pMdConfig,`
	`  UINT32 numExchgPar,`
	`  TRDP_EXCHG_PAR_T **ppExchgPar`
	`);`	

Synopsis C++	`TRDP_ERR_T tau::readXmlInterfaceConfig(`
	`  const TRDP_XML_DOC_HANDLE_T *pDocHnd,`
	`  const CHAR8 *pIfName,`
	`  TRDP_PROCESS_CONFIG_T *pProcessConfig,`
	`  TRDP_PD_CONFIG_T *pPdConfig,`
	`  TRDP_MD_CONFIG_T *pMdConfig,`
	`  UINT32 numExchgPar,`
	`  TRDP_EXCHG_PAR_T **ppExchgPar`
	`);`	

Abstract	Read the interface relevant telegram parameters (except data set configuration) out of the configuration file.	
Parameters	pDocHnd	Handle of the parsed XML file
	pIfName	Interface name
	pProcessConfig	TRDP main process configuration
	pPdConfig	PD default configuration
	pMdConfig	MD default configuration
	pNumExchgPar	Number of configured telegrams
	ppExchgPar	Pointer to an array of telegram configurations
Returns	0 if OK, !=0 if error	

Type	Name	Description
TRDP_LABEL_T	hostName	Host name
TRDP_LABEL_T	leaderName	Leader name dependant on redundancy concept
UINT32	cycleTime	TRDP main process cycle time in µs
UINT32	priority	TRDP main process cycle time (0-255, 0=default, 255=highest)
TRDP_OPTION_T	options	TRDP options

Table 82 Structure TRDP_PROCESS_CONFIG_T – TRDP process configuration parameters

Type	Name	Description
UINT32	comId	ComId **Note:** ComId's 1-1000 are reserved for special purpose (see Table 59).
UINT32	datasetId	Dataset ID
UINT32	comParId	Communication parameter ID
TRDP_MD_PAR_T	mdPar	Structure with the MD send parameters see Table 85.
TRDP_PD_PAR_T	pdPar	Structure with the PD receive parameters see Table 86.
UINT32	destCnt	Number of destination URI's
TRDP_DEST_T	*pDest	Pointer to a destination handled as a list
UINT32	srcCnt	Number of source URI's
TRDPC_T	*pSrc	Pointer to a source handled as a list

Table 83 Structure TRDP_EXCHG_PAR_T – communication exchange parameters

Type	Name	Description
UINT32	smi1	Safe message identifier – unique for this message at consist level
UINT32	smi2	Safe message identifier for a redundant device – unique for this message at consist level
UINT32	cmThr	Channel monitoring threshold
UINT16	udv	User data version
UINT16	rxPeriod	Sink cycle time
UINT16	txPeriod	Source cycle time
UINT16	nGuard	Initial timeout cycles
UINT8	nRxSafe	Timeout cycles
UINT8	reserved1	Reserved for future use
UINT16	Reserved2	Reserved for future use

Table 84 Structure TRDP_SDT_PAR_T – SDT communication parameter

Type	Name	Description
UINT32	confirmTimeout	Acknowledge time-out in µs
UINT32	replyTimeout	Response time-out in µs
TRDP_FLAGS_T	flags	TRDP_FLAGS_MARSHALL, TRDP_FLAGS_TCP

Table 85 Structure TRDP_MD_PAR_T – MD communication parameter

Type	Name	Description
UINT32	cycle	PD cycle time
UINT32	redundant	0 = not redundant, >0 redundancy group
UINT32	timeout	Timeout value in µs, before considering received process data invalid
TRDP_TO_BEHAVIOR	toBehav	Behaviour when received process data is invalid/timed out.
TRDP_FLAGS_T	flags	TRDP_FLAGS_MARSHALL, TRDP_FLAGS_CALLBACK

Table 86 Structure `TRDP_PD_PAR_T` – PD communication parameter

Type	Name	Description
UINT32	id	Destination identifier
TRDP_SDT_PAR_T	*pSdtPar	Parameter for safe data transmission see Table 84
TRDP_URI_USER_T	*pUriUser	Pointer to URI user part
TRDP_URI_HOST_T	*pUriHost	Pointer to URI host part or IP

Table 87 Structure `TRDP_DEST_T` – Destination addresses

Type	Name	Description
UINT32	id	Source filter identifier
TRDP_SDT_PAR_T	*pSdtPar	Parameter for safe data transmission see Table 84
TRDP_URI_USER_T	*pUriUser	Pointer to URI user part
TRDP_URI_HOST_T	*pUriHost	Pointer to URI host part or IP
TRDP_URI_HOST_T	*pRedUriHost	Pointer to URI host part or IP of redundant source

Table 88 Structure `TRDP_SRC_T` – Source addresses

14.1.5. tau_readXmlDatasetConfig

Name	tau_readXmlDatasetConfig	

Synopsis C	`TRDP_ERR_T tau_readXmlDatasetConfig(`
	`  const TRDP_XML_DOC_HANDLE_T *pDocHnd,`
	`  UINT32 *pNumComId`
	`  TRDP_COMID_DSID_MAP_T **ppComIdDsIdMap,`
	`  UINT32 *pNumDataset,`
	`  TRDP_DATASET_T **ppDataset);`	

Synopsis C++	`TRDP_ERR_T tau::readXmlDatasetConfig(`
	`  const TRDP_XML_DOC_HANDLE_T *pDocHnd,`
	`  UINT32 *pNumComId`
	`  TRDP_COMID_DSID_MAP_T **ppComIdDsIdMap,`
	`  UINT32 *pNumDataset,`
	`  TRDP_DATASET_T ***papDataset);`	

Abstract	Reads all dataset configurations out of the given XML file. Allocated memory for the returned lists needs to be freed with vos_memFree().	
Parameters	`pDocHnd`	`Handle of the parsed XML file`
	`pNumComId`	Number of entries in the ComId DatasetId mapping list.
	`ppComIdDsIdMap`	Pointer to the ComId DatasetId mapping list.
	`pNumDataset`	Pointer to number of datasets read
	`papDataset`	Pointer to an array of structures of type `TRDP_DATASET_T`, see Table 5.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

Type	Name	Description
`UINT32`	`comId`	Communication parameter identifier.
`UINT32`	`dsId`	Dataset identifier.

Table 89 Structure `TRDP_COMID_DSID_MAP_T` – comId – data set mapping element

14.2. TRDP Train Configuration Information API

TRDP contains some general utilities functions to access the train configuration information.

The train configuration information will be delivered within different telegrams from the ETBN:

1. PD push telegram with safe topo count and ETB-state (pushed at least in 1000ms cycle)
2. PD pull telegram with dynamic train information (pulled after train inauguration by ETBN itself as a multicast to all devices in the consist)
3. MD telegram (call/reply pattern) to retrieve static consist and car information
4. MD telegram (call/reply pattern) to retrieve device information for one or all devices in a given car in a consist
5. MD telegram (call reply pattern) to retrieve consist and car properties
6. MD telegram (call reply pattern) to retrieve device properties for one or all devices in a given car in a consist
7. MD telegram for manual insertion of consists

The train configuration info can be retrieved with the following algorithm adding input and return value checks:

```
void getTrainConfig(
   TRDP_DEV_INFO_T    *pTrnInfo,
   UINT32             *topoCnt)
{
   UINT16 dev;
   UINT16 car;
   UINT16 cst;

   TRDP_DEV_INFO_T *pDevInfo
   TRDP_CAR_INFO_T *pCarInfo
   TRDP_CST_INFO_T *pCstInfo

   /* free memory of old configuration */
   for (cst=0, cst< pTrnInfo->cstCnt, cst++)
   {
      pCstInfo= &(pTrnInfo->pCstInfo[cst]);

      for (car=0, car<pCstInfo->carCnt, car++)
      {
         pCarInfo=&(pTrnInfo->pCstInfo[cst].pCarInfo[car]);

         for (dev=0, dev < pCarInfo->devCnt, dev++)
         {
            /* free  detailed device info, if needed */
            vos_memFree(pCarInfo->pDevInfo[dev].pProp);
            vos_memFree(pCarInfo->pDevInfo[dev].pFctNo);
         }
         vos_memFree(pCarInfo->pProp);
         vos_memFree(pCarInfo->pDevInfo);
      }
      vos_memFree(pCstInfo->pProp);
```

```
      vos_memFree(pCstInfo->pCarInfo);
      vos_memFree(pCstInfo->pFct);
   }
   vos_memFree(pTrnInfo->pCstInfo);

   /* retrieve new train configuration info */
   tau_getTrnInfo(pTrnInfo, pTopoCnt);

   /* provide memory for basic consist info */
   pTrnInfo->pCstInfo = vos_memAlloc(pTrnInfo.cstCnt * sizeof(TRDPCST_INFO_T));

   tau_getCstBasicInfo(pTrnInfo->pCstInfo, pTopoCnt, "acst", trnInfo.cstCnt);

   for (cst=0, cst< pTrnInfo->cstCnt, cst++)
   {
      pCstInfo= &(pTrnInfo->pCstInfo[cst]);

      /* provide memory for basic car info */
      pCstInfo->pCarInfo=vos_memAlloc(pCstInfo->carCnt*sizeof(TRDP_CAR_INFO_T));

      /* provide memory, if consist properties needed */
      pCstInfo->pProp=vos_memAlloc(pCstInfo->propLen*sizeof(UINT8));

      /* provide memory, if function information needed */
      pCstInfo->pFct=vos_memAlloc(pCstInfo->fctCnt*sizeof(TRDP_FCT_INFO_T));
   }
   tau_getCstDetailInfo(pTrnInfo->pCstInfo, pTopoCnt, NULL, pTrnInfo->cstCnt);

   /* get more detailed car information, which is typically not necessary */
   for (cst=0, cst< pTrnInfo->cstCnt, cst++)
   {
      pCstInfo= &(pTrnInfo->pCstInfo[cst]);

      for (car=0, car<pCstInfo->carCnt, car++)
      {
         pCarInfo=&(pCstInfo->pCarInfo[car]);

         /* provide memory, if car properties needed */
         pCarInfo->pProp=vos_memAlloc(pCarInfo->propLen*sizeof(UINT8));

         /* provide memory, if device information needed */
         pCarInfo->pDevInfo=vos_memAlloc(pCarInfo->devCnt*sizeof(TRDP_DEV_INFO_T));

         tau_getCarDetailInfo(
            pCarInfo,
            pTopoCnt,
            pCstInfo->id,
            NULL,
            "grpAll",
            pCstInfo->carCnt);

         /* retrieve detailed device info, if needed */
```

```
            for (dev=0, dev < pCarInfo->devCnt, dev++)
            {
               pDevInfo =  &(pCarInfo->pDevInfo[dev]);

               /* provide memory, if device information needed */
               pDevInfo->pProp=vos_memAlloc(pDevInfo->propLen*sizeof(UINT8));

               /* provide memory, if function information needed */
               pDevInfo->pFctNo=vos_memAlloc(pDevInfo->fctCnt*sizeof(UINT32));

               tau_getDevDetailInfo(
                  pDevInfo,
                  pTopoCnt,
                  pCstInfo->id,
                  pCarInfo->id,
                  NULL,
                  pCstInfo->devCnt);
            }
         }
      }
   }
}
```

14.2.1. Definitions

Value	Name	Description
0	TRDP_INAUG_INVALID	Ongoing inauguration, DNS not yet available, no address transformation possible
1	TRDP_INAUG_FAULT	Error in train inauguration, DNS not available, train wide communication not possible
2	TRDP_INAUG_NOLEAD_UNCONF	inauguration done, no leading vehicle set, inauguration unconfirmed
3	TRDP_INAUG_LEAD_UNCONF	inauguration done, leading vehicle set, inauguration unconfirmed
4	TRDP_INAUG_LEAD_CONF	inauguration done, leading vehicle set, inauguration confirmed

Table 90 Enumeration `TRDP_INAUG_STATE_T` – ETB inauguration states

Value	Name	Description
0	TRDP_FCT_INVALID	Invalid type
1	TRDP_FCT_LOCAL	Device local function
2	TRDP_FCT_CAR	Car control function
3	TRDP_FCT_CST	Consist control function
4	TRDP_FCT_TRAIN	Train control function

Table 91 Enumeration `TRDP_FCT_T` – function type

Type	Name	Description
TRDP_LABEL_T	id	function identifier (name)
TRDP_FCT_T	type	function type
UINT32	no	unique function number in consist, should be the list index number
TRDP_IP_ADDR	addr	Device IP address / multicast address
UINT8	ecnId	Ethernet consist network id.
UINT8	etbId	Ethernet train backbone id.

Table 92 Structure `TRDP_FCT_INFO_T` – function configuration information

Type	Name	Description
TRDP_IP_ADDR	addr1	First device IP address
TRDP_IP_ADDR	addr2	Second device IP address
TRDP_LABEL_T	id	device id / host name
TRDP_LABEL_T	type	device type (reserved key words ETBN, ETBR, FCT)
UINT8	orient	device orientation 0=opposite, 1=same related to car
TRDP_LABEL_T	redId	redundant device Id if available
UINT8	ecnId1	First Ethernet consist network id. 0 means no connection.
UINT8	ecnId2	Second Ethernet consist network id. 0 means no connection.
UINT8	etbId1	First Ethernet train backbone id. 0 means no connection.
UINT8	etbId2	Second Ethernet train backbone id. 0 means no connection.
UINT16	fctCnt	Number of public functions on the device
UINT32	*pFctNo	Pointer to device function number list for application use and convenience
UINT16	propVer	Properties version
UINT16	propLen	Length of device properties
UINT8	*pProp	Pointer to device properties for application use and convenience

Table 93 Structure `TRDP_DEV_INFO_T` – device configuration information

Type	Name	Description
TRDP_LABEL_T	id	car id
TRDP_LABEL_T	type	car type
UINT8	orient	0 == opposite, 1 == same orientation related to consist
UINT8	lead	0 == car is not leading
UINT8	leadDir	0 == leading direction 1, 1 == leading direction 2
UINT8	no	Car number in consist
UINT8	iecNo	IEC car number in train
UINT8	reachable	0 == car not reachable, inserted manually
UINT16	devCnt	number of devices in the car
TRDP_DEVICE_INFO_T	*pDevInfo	Pointer to device info list for application use and convenience.
UINT16	propVer	Properties version
UINT16	propLen	Length of car properties
UINT8	*pProp	Pointer to car properties for application use and convenience

Table 94 Structure `TRDP_CAR_INFO_T` – car configuration information

Type	Name	Description
TRDP_LABEL_T	id	Unique consist identifier (unique at least for the owner) / IEC identification number taken from 1st car in consist
TRDP_LABEL_T	owner	consist owner, e.g. “trenitalia.it”
TRDP_UUID_T	uuid	consist UUID for inauguration purposes
UINT8	orient	0 == opposite, 1 == same orientation related to consist
UINT8	lead	0 == car is not leading
UINT8	leadDir	0 == leading direction 1, 1 == leading direction 2
UINT8	tcnNo	Consist number in train related to TCN reference direction
UINT8	iecNo	Consist number in train related to leading car depending IEC reference direction
UINT8	reachable	0 == consist not reachable, inserted manually
UINT8	ecnCnt	Number of ECN in consist
UINT8	etbCnt	Number of ETB in consist
UINT16	fctCnt	Number of public functions in the consist
TRDP_FCT_INFO_T	*pFct	Pointer to consist public function list. Memory needs to be provided by application before calling tau_getAddCstInfo()
UINT16	carCnt	number of cars in consist
TRDP_CAR_INFO_T	*pCarInfo	Pointer to car info list. Memory needs to be provided by application before calling tau_getAddCstInfo()
UINT16	propVer	Poperties version
UINT16	propLen	Length of consist properties
UINT8	*pProp	Pointer to consist properties. Memory needs to be provided by application before calling tau_getAddCstInfo()

Table 95 Structure `TRDP_CST_INFO_T` – consist configuration information

Type	Name	Description
UINT32	version	Train info structure version
TRDP_LABEL_T	id	Train id, e.g. “ICE75”, “IC346”
TRDP_LABEL_T	operator	Train operator, e.g. “trainitalia.it”
TRDP_INAUG_STATE_T	inaugState	Train inauguration status
UINT32	topoCnt	IEC (i.e. TCN) topography counter
UINT8	iecOrient	0 == IEC reference orientation is opposite to TCN reference direction
UINT16	carCnt	number of cars in the train
UINT16	cstCnt	number of consists in the train
TRDP_CST_INFO_T	*pCstInfo	Pointer to consist info list for application use and convenience.

Table 96 Structure `TRDP_TRAIN_INFO_T` – train configuration information

14.2.2. tau_getEtbState

Name	tau_getEtbState	

Synopsis C	`TRDP_ERR_T tau_getEtbState (`
	`  TRDP_INAUG_STATE_T  *pInaugState,`
	`  UINT32  *pTopoCnt);`	

Synopsis C++	`TRDP_ERR_T tau::getEtbState (`
	`  TRDP_INAUG_STATE_T  *pInaugState,`
	`  UINT32  *pTopoCnt);`	

Abstract	Function to retrieve the inauguration state and the topography counter.	
Parameters	`pInaugState`	Pointer to an inauguration state variable to be returned.
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.2.3. tau_getTrnInfo

Name	tau_getTrnInfo	

Synopsis C	`TRDP_ERR_T tau_getTrnInfo (`
	`  TRDP_TRAIN_INFO_T  *pTrnInfo,`
	`  UINT32  *pTopoCnt);`	

Synopsis C++	`TRDP_ERR_T tau::getTrnInfo (`
	`  TRDP_TRAIN_INFO_T  *pTrnInfo,`
	`  UINT32  *pTopoCnt) ;`	

Abstract	Function to retrieve the consist information of a train's consist.	
Parameters	`pTrnInfo`	Pointer to train information to be returned. Memory needs to be provided by application.
	`pTopoCnt`	Pointer to topo counter. Input: If the topo counter value set at the call is different from 0, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.2.4. tau_getCstBasicInfo

Name	tau_getCstBasicInfo	

Synopsis C	`TRDP_ERR_T tau_getCstBasicInfo (`
	`  TRDP_CST_INFO_T  *pCstInfo,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  cstLabel,`
	`  UINT16  cstCnt);`	

Synopsis C++	`TRDP_ERR_T tau::getCstBasicInfo (`
	`  TRDP_CST_INFO_T  *pCstInfo,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  cstLabel,`
	`  UINT16  cstCnt);`	

Abstract	Function to retrieve the consist basic information of train consist(s). **Note:** Consist detail information like car information, consist properties and consist functions can be only retrieved in a second step providing the memory in the consist info structure by the application.	
Parameters	`pCstInfo`	Pointer to consist information to be returned. Memory needs to be provided by application.
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`cstLabel`	Pointer to a consist label. NULL means own consist.
	`cstCnt`	Number of consists fitting in the provided memory with pCstInfo.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.2.5. tau_getCstDetailInfo

Name	**tau_getCstDetailInfo**	

Synopsis C	`TRDP_ERR_T tau_getCstDetailInfo (`
	`  TRDP_CST_INFO_T  *pCstInfo,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  cstLabel,`
	`  UINT16  cstCnt);`	

Synopsis C++	`TRDP_ERR_T tau::getCstDetailInfo (`
	`  TRDP_CST_INFO_T  *pCstInfo,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  cstLabel,`
	`  UINT16  cstCnt);`	

Abstract	Function to retrieve the consist additional information (consist properties, consist function table, car table) of consist(s). **Note:** the memory needs to be provided by the application by putting valid pointers in the consist info structure(s). Null pointers will be skipped.	
Parameters	`pCstInfo`	Pointer to consist information to be returned. Memory needs to be provided by application.
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`cstLabel`	Pointer to a consist label. If NULL the label from the consist info structure is taken.
	`cstCnt`	Number of consists in the consist info list provided with pCstInfo.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.2.6. tau_getCarDetailInfo

Name	tau_getCarDetailInfo	

Synopsis C	`TRDP_ERR_T tau_getCarDetailInfo (`
	`  TRDP_CAR_INFO_T  *pCarInfo,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  devLabel,`
	`  const TRDP_LABEL_T  carLabel,`
	`  const TRDP_LABEL_T  cstLabel,`
	`  UINT32  carCnt);`	

Synopsis C++	`TRDP_ERR_T tau::getCarDetailInfo (`
	`  TRDP_CAR_INFO_T  *pCarInfo,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  devLabel,`
	`  const TRDP_LABEL_T  carLabel,`
	`  const TRDP_LABEL_T  cstLabel,`
	`  UINT32  carCnt);`	

Abstract	Function to retrieve the car detail information (car properties, function table and device table) of a consist's car. **Note:** the memory needs to be provided by the application by putting valid pointers in the car info structure(s). Null pointers will be skipped.	
Parameters	`pCarInfo`	Pointer to car information. Memory needs to be provided by application.
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`devLabel`	Pointer to a device label.
	`carLabel`	Pointer to a car label. If NULL the label from car info structure is taken.
	`cstLabel`	Pointer to a consist label.
	`carCnt`	Number of cars in the car info list provided with pCarInfo.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.2.7. tau_getDevDetailInfo

Name	**tau_getDevDetailInfo**	

Synopsis C	`TRDP_ERR_T tau_getDevDetailInfo (`
	`  TRDP_DEV_INFO_T  *pDevInfo,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  devLabel,`
	`  const TRDP_LABEL_T  carLabel,`
	`  const TRDP_LABEL_T  cstLabel,`
	`  UINT16  devCnt);`	

Synopsis C++	`TRDP_ERR_T tau::getDevDetailInfo (`
	`  TRDP_DEV_INFO_T  *pDevInfo,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  devLabel,`
	`  const TRDP_LABEL_T  carLabel,`
	`  const TRDP_LABEL_T  cstLabel,`
	`  UINT16  devCnt);`	

Abstract	Function to retrieve the detailed device information like device function table and properties of car's device(s).	
Parameters	`pDevInfo`	Pointer to device information to be returned. Memory needs to be provided by application.
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`devLabel`	Pointer to a device label. NULL means that the label from the dev info structure is taken.
	`carLabel`	Pointer to a car label.
	`cstLabel`	Pointer to a consist label.
	`devCnt`	Number of devices in the device list provided with pDevInfo.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.2.8. tau_insertCstInfo

Name	**tau_insertCstInfo**	

Synopsis C	`TRDP_ERR_T tau_insertCstInfo (`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_CST_INFO_T *pCstInfo);`	

Synopsis C++	`TRDP_ERR_T tau::insertCstInfo (`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_CST_INFO_T *pCstInfo);`	

Abstract	Function to insert the consist information of a train's consist for correction of train inauguration result.	
Parameters	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`pCstInfo`	Pointer to consist information to be inserted. Memory needs to be provided by application and can be freed after the function call.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.2.9. tau_getTrnCarCnt

Name	tau_getTrnCarCnt	

Synopsis C	`TRDP_ERR_T tau_getTrnCarCnt (`
	`  UINT16  *pTrnCarCnt,`
	`  UINT32  *pTopoCnt);`	

Synopsis C++	`TRDP_ERR_T tau::getTrnCarCnt (`
	`  UINT16  *pTrnCarCnt,`
	`  UINT32  *pTopoCnt);`	

Abstract	Function to retrieve the total number of cars in the train.	
Parameters	`pTrnCarCnt`	Pointer to the number of cars to be returned.
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.2.10. tau_getCstCarCnt

Name	tau_getCstCarCnt	

Synopsis C	`TRDP_ERR_T tau_getCstCarCnt (`
	`  UINT16  *pCstCarCnt,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  cstLabel);`	

Synopsis C++	`TRDP_ERR_T tau::getCstCarCnt (`
	`  UINT16  *pCstCarCnt,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  cstLabel);`	

Abstract	Function to retrieve the total number of cars in the given consist.	
Parameters	`pCstCarCnt`	Pointer to the number of cars to be returned.
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`cstLabel`	Pointer to a consist label. NULL means own consist.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.2.11. tau_getCstFctCnt

Name	**tau_getCstFctCnt**	

Synopsis C	`TRDP_ERR_T tau_getCstFctCnt (`
	`  UINT16  *pCstFctCnt,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  cstLabel);`	

Synopsis C++	`TRDP_ERR_T tau::getCstFctCnt (`
	`  UINT16  *pCstFctCnt,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  cstLabel);`	

Abstract	Function to retrieve the total number of public functions in the given consist.	
Parameters	`pCstFctCnt`	Pointer to the number of functions to be returned.
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`cstLabel`	Pointer to a consist label. NULL means own consist.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.2.12. tau_getCarDevCnt

Name	**tau_getCarDevCnt**	

Synopsis C	`TRDP_ERR_T tau_getCarDevCnt (`
	`  UINT16  *pCarDevCnt,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  carLabel,`
	`  const TRDP_LABEL_T  cstLabel);`	

Synopsis C++	`TRDP_ERR_T tau::getCarDevCnt (`
	`  UINT16  *pCarDevCnt,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  carLabel,`
	`  const TRDP_LABEL_T  cstLabel);`	

Abstract	Function to retrieve the total number of devices in a car.	
Parameters	`pCarDevCnt`	Pointer to the number of devices to be returned.
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`carLabel`	Pointer to a car label. NULL means own car if cstLabel == NULL.
	`cstLabel`	Pointer to a consist label. NULL means own consist.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.2.13. tau_getOrient

Name	tau_getOrient	

Synopsis C	`TRDP_ERR_T tau_getOrient (`
	`  UINT8 *pDevOrient,`
	`  UINT8 *pCarOrient,`
	`  UINT8 *pCstOrient,`
	`  UINT8 *pIecOrient,`
	`  UINT32 *pTopoCnt,`
	`  TRDP_LABEL_T devLabel,`
	`  TRDP_LABEL_T carLabel,`
	`  TRDP_LABEL_T cstLabel);`	

Synopsis C++	`TRDP_ERR_T tau::getOrient (`
	`  UINT8 *pDevOrient,`
	`  UINT8 *pCarOrient,`
	`  UINT8 *pCstOrient,`
	`  UINT8 *pIecOrient,`
	`  UINT32 *pTopoCnt,`
	`  TRDP_LABEL_T devLabel,`
	`  TRDP_LABEL_T carLabel,`
	`  TRDP_LABEL_T cstLabel);`	

Abstract	Function to retrieve the orientation of the given consist.	
Parameters	`pDevOrient`	Pointer to device orientation related to car to be returned. 0 == opposite, 1 == same orientation related to car
	`pCarOrient`	Pointer to car orientation related to consist to be returned. 0 == opposite, 1 == same orientation related to consist
	`pCstOrient`	Pointer to consist orientation related to TCN reference direction to be returned. 0 == opposite, 1 == same orientation related to TCN reference direction
	`pIecOrient`	Pointer to IEC orientation related to TCN orientation to be returned. 0 == opposite, 1 == same orientation related to TCN reference direction
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`devLabel`	devLabel == NULL means own device if own car, own consist is selected.
	`carLabel`	carLabel == NULL means own car if own consist is selected.
	`cstLabel`	cstLabel == NULL means own consist
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3. TRDP Address API

TRDP contains some general utilities functions for IP address and URI resolution.

14.3.1. Definitions

No specific definitions.

14.3.2. tau_getOwnIds

Name	tau_getOwnIds	

Synopsis C	`TRDP_ERR_T tau_getOwnIds (`
	`  TRDP_LABEL_T  devId,`
	`  TRDP_LABEL_T  carId,`
	`  TRDP_LABEL_T  cstId);`	

Synopsis C++	`TRDP_ERR_T tau::getOwnIds (`
	`  TRDP_LABEL_T  devId,`
	`  TRDP_LABEL_T  carId,`
	`  TRDP_LABEL_T  cstId);`	

Abstract	Get URI from an IP address and topo counter value. The topo counter value can be checked against the current value, see below.	
Parameters	`devId`	Returns the device label (host name).
	`carId`	Returns the car label.
	`cstId`	Returns the consist label.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3.3. tau_getOwnAddr

Name	tau_getOwnAddr	
Synopsis C	`UINT32 tau_getOwnAddr(void);`	
Synopsis C++	`UINT32 tau::getOwnAddr(void);`	
Abstract	Get own IP address.	
Parameters	-	
Returns C/C++	IP address	

14.3.4. tau_addr2Uri

Name	tau_addr2Uri	

Synopsis C	`TRDP_ERR_T tau_addr2Uri (`
	`  TRDP_URI_HOST_T  uri,`
	`  UINT32  *pTopoCnt,`
	`  TRDP_IP_ADDR  addr);`	

Synopsis C++	`TRDP_ERR_T tau::addr2Uri (`
	`  TRDP_URI_HOST_T  uri,`
	`  UINT32  *pTopoCnt,`
	`  TRDP_IP_ADDR  addr);`	

Abstract	Get URI from an IP address and topo counter value. The topo counter value can be checked against the current value, see below.	
Parameters	`uri`	Pointer to resulting URI.
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`addr`	IP address, 0 == own address.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3.5. tau_uri2Addr

Name	tau_uri2Addr	

Synopsis C	`TRDP_ERR_T tau__uri2Addr (`
	`  TRDP_IP_ADDR  *pAddr,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_URI_T  uri);`	

Synopsis C++	`TRDP_ERR_T tau::uri2Addr(`
	`  TRDP_IP_ADDR  *pAddr,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_URI_T  uri);`	

Abstract	Get IP address from an URI and topo counter value. The topo counter value can be checked against the current value, see below.	
Parameters	`pAddr`	Pointer to resulting IP address
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`uri`	Pointer to URI, NULL==own URI
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3.6. tau_label2CarId

Name	tau_label2CarId	

Synopsis C	`TRDP_ERR_T tau_label2CarId (`
	`  TRDP_LABEL_T  carId,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  carLabel,`
	`  const TRDP_LABEL_T  cstLabel);`	

Synopsis C++	`TRDP_ERR_T tau::label2CarId (`
	`  TRDP_LABEL_T  carId,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  carLabel,`
	`  const TRDP_LABEL_T  cstLabel);`	

Abstract	Function to retrieve the carId of the car with label carLabel in the consist with cstLabel.	
Parameters	`carId`	Pointer to a label string to return the car id
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`carLabel`	Pointer to the car label. NULL means own car if cstLabel == NULL.
	`cstLabel`	Pointer to the consist label. NULL means own consist.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3.7. tau_label2CarNo

Name	tau_label2CarNo	

Synopsis C	`TRDP_ERR_T tau_label2CarNo (`
	`  UINT8  *pCarNo,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  carLabel,`
	`  const TRDP_LABEL_T  cstLabel);`	

Synopsis C++	`TRDP_ERR_T tau::label2CarNo (`
	`  UINT8  *pCarNo,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  carLabel,`
	`  const TRDP_LABEL_T  cstLabel);`	

Abstract	The function delivers the car number to the given label. The first match of the table will be returned in case there is no unique label given.	
Parameters	`pCarNo`	Pointer to the car number to be returned
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`carLabel`	Pointer to the car label. NULL means own car if cstLabel == NULL.
	`cstLabel`	Pointer to the consist label. NULL means own consist.
Returns	0 if OK, !=0 if error, see chapter 12.1	

C/C++	

14.3.8. tau_label2IecCarNo

Name	tau_label2IecCarNo	

Synopsis C	`TRDP_ERR_T tau_label2IecCarNo (`
	`  UINT8 *pIecCarNo,`
	`  UINT32 *pTopoCnt,`
	`  const TRDP_LABEL_T carLabel,`
	`  const TRDP_LABEL_T cstLabel);`	

Synopsis C++	`TRDP_ERR_T tau::label2IecCarNo (`
	`  UINT8 *pIecCarNo,`
	`  UINT32 *pTopoCnt,`
	`  const TRDP_LABEL_T carLabel,`
	`  const TRDP_LABEL_T cstLabel);`	

Abstract	The function delivers the IEC car number to the given label. The first match of the table will be returned in case there is no unique label given.	
Parameters	`pIecCarNo`	Pointer to the IEC car number to be returned
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`carLabel`	Pointer to the car label. NULL means own car if cstLabel == NULL.
	`cstLabel`	Pointer to the consist label. NULL means own consist.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3.9. tau_carNo2Ids

Name	**tau_carNo2Ids**	

Synopsis C	`TRDP_ERR_T tau_carNo2Ids (`
	`  TRDP_LABEL_T  carId,`
	`  TRDP_LABEL_T  cstId,`
	`  UINT32  *pTopoCnt,`
	`  UINT8  carNo,`
	`  UINT8  trnCstNo);`	

Synopsis C++	`TRDP_ERR_T tau::carNo2Ids (`
	`  TRDP_LABEL_T  carId,`
	`  TRDP_LABEL_T  cstId,`
	`  UINT32  *pTopoCnt,`
	`  UINT8  carNo,`
	`  UINT8  trnCstNo);`	

Abstract	Function to retrieve the car and consist id of the car given with carNo and trnCstNo.	
Parameters	`carId`	Pointer to the car id to be returned
	`cstId`	Pointer to the consist id to be returned
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`carNo`	Car number in consist. 0 means own car when trnCstNo == 0.
	`trnCstNo`	Consist sequence number in train. 0 means own consist.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3.10. tau_iecCarNo2Ids

Name	tau_iecCarNo2Ids	

Synopsis C	`TRDP_ERR_T tau_iecCarNo2Ids (`
	`  TRDP_LABEL_T  carId,`
	`  TRDP_LABEL_T  cstId,`
	`  UINT32  *pTopoCnt,`
	`  UINT8  iecCarNo);`	

Synopsis C++	`TRDP_ERR_T tau::iecCarNo2Ids (`
	`  TRDP_LABEL_T  carId,`
	`  TRDP_LABEL_T  cstId,`
	`  UINT32  *pTopoCnt,`
	`  UINT8  iecCarNo);`	

Abstract	Function to retrieve the car and consist id from a given IEC car sequence number.	
Parameters	`carId`	Pointer to the car id to be returned
	`cstId`	Pointer to the consist id to be returned
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`iecCarNo`	IEC car number in train. 0 means own car when trnCstNo == 0.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3.11. tau_addr2CarId

Name	tau_addr2CarId	

Synopsis C	`TRDP_ERR_T tau_addr2CarId (`
	`  TRDP_LABEL_T  carId,`
	`  UINT32  *pTopoCnt,`
	`  TRDP_IP_ADDR  ipAddr);`	

Synopsis C++	`TRDP_ERR_T tau:: addr2CarId (`
	`  TRDP_LABEL_T  carId,`
	`  UINT32  *pTopoCnt,`
	`  TRDP_IP_ADDR  ipAddr);`	

Abstract	Function to retrieve the carId of the car hosting a device with the IPAddress ipAddr.	
Parameters	`carId`	Pointer to the car id to be returned
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`ipAddr`	IP address. 0 means own address, so the own car id is returned.
Returns C/C++	0 if OK, !=0 if error.	

14.3.12. tau_addr2CarNo

Name	tau_addr2CarNo	

Synopsis C	`TRDP_ERR_T tau_addr2CarNo (`
	`  UINT8  *pCarNo,`
	`  UINT32  *pTopoCnt,`
	`  TRDP_IP_ADDR  ipAddr);`	

Synopsis C++	`TRDP_ERR_T tau::addr2CarNo (`
	`  UINT8  *pCarNo,`
	`  UINT32  *pTopoCnt,`
	`  TRDP_IP_ADDR  ipAddr);`	

Abstract	Function to retrieve the car number in consist of the car hosting the device with the IP address ipAddr.	
Parameters	`pCarNo`	Pointer to the car number in consist to be returned
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`ipAddr`	IP address. 0 means own address, so the own car id is returned.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3.13. tau_addr2IecCarNo

Name	tau_addr2IecCarNo	

Synopsis C	`TRDP_ERR_T tau_addr2IecCarNo (`
	`  UINT8  *pIecCarNo,`
	`  UINT32  *pTopoCnt,`
	`  TRDP_IP_ADDR  ipAddr);`	

Synopsis C++	`TRDP_ERR_T tau::addr2IecCarNo (`
	`  UINT8  *pIecCarNo,`
	`  UINT32  *pTopoCnt,`
	`  TRDP_IP_ADDR  ipAddr);`	

Abstract	Function to retrieve the IEC car number of the car hosting the device with the IP address ipAddr.	
Parameters	`pIecCarNo`	Pointer to the IEC car number in train to be returned
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`ipAddr`	IP address. 0 means own address, so the own car id is returned.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3.14. tau_cstNo2CstId

Name	tau_cstNo2CstId	

Synopsis C	`TRDP_ERR_T tau_cstNo2CstId (`
	`  TRDP_LABEL_T  cstId,`
	`  UINT32  *pTopoCnt,`
	`  UINT32  cstNo);`	

Synopsis C++	`TRDP_ERR_T tau::cstNo2CstId (`
	`  TRDP_LABEL_T  cstId,`
	`  UINT32  *pTopoCnt,`
	`  UINT32  cstNo);`	

Abstract	Function to retrieve the consist identifier of the consist with train consist sequence number cstNo.	
Parameters	`cstId`	Pointer to the consist id to be returned
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`cstNo`	Consist sequence number based on IP reference direction. 0 means own consist.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3.15. tau_iecCstNo2CstId

Name	tau_iecCstNo2CstId	

Synopsis C	`TRDP_ERR_T tau_iecCstNo2CstId (`
	`  TRDP_LABEL_T  cstId,`
	`  UINT32  *pTopoCnt,`
	`  UINT32  iecCstNo);`	

Synopsis C++	`TRDP_ERR_T tau::iecCstNo2CstId (`
	`  TRDP_LABEL_T  cstId,`
	`  UINT32  *pTopoCnt,`
	`  UINT32  iecCstNo);`	

Abstract	Function to retrieve the consist identifier of the consist with IEC consist number iecCstNo.	
Parameters	`cstId`	Pointer to the consist id to be returned
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`iecCstNo`	IEC consist number based on the leading car depending IEC reference direction. 0 means own consist.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3.16. tau_label2CstId

Name	**tau_label2CstId**	

Synopsis C	`TRDP_ERR_T tau_label2CstId (`
	`  TRDP_LABEL_T  cstId,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  carLabel,`
	`  const TRDP_LABEL_T  cstLabel);`	

Synopsis C++	`TRDP_ERR_T tau:: label2CstId (`
	`  TRDP_LABEL_T  cstId,`
	`  UINT32  *pTopoCnt,`
	`  const TRDP_LABEL_T  carLabel,`
	`  const TRDP_LABEL_T  cstLabel);`	

Abstract	Function to retrieve the consist identifier of the consist with IEC sequence consist number iecCstNo.	
Parameters	`cstId`	Pointer to the consist id to be returned
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`carLabel`	Pointer to a car label. NULL means any car if cstLabel is != NULL.
	`cstLabel`	Pointer to a consist label. NULL means any consist if carLabel != NULL.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3.17. tau_label2CstNo

Name	tau_label2CstNo	

Synopsis C	`TRDP_ERR_T tau_label2CstNo (`
	`  UINT8 *pCstNo,`
	`  UINT32 *pTopoCnt,`
	`  const TRDP_LABEL_T carLabel);`	

Synopsis C++	`TRDP_ERR_T tau::label2CstNo (`
	`  UINT8 *pCstNo,`
	`  UINT32 *pTopoCnt,`
	`  const TRDP_LABEL_T carLabel);`	

Abstract	Function to retrieve the consist sequence number of the consist hosting a car with label carLabel.	
Parameters	`pCstNo`	Pointer to the consist number to be returned
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`carLabel`	Pointer to a car label. NULL means own car.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3.18. tau_label2IecCstNo

Name	tau_label2iecCstNo	

Synopsis C	`TRDP_ERR_T tau_label2iecCstNo (`
	`  UINT8 *pIecCstNo,`
	`  UINT32 *pTopoCnt,`
	`  const TRDP_LABEL_T carLabel);`	

Synopsis C++	`TRDP_ERR_T tau::label2iecCstNo (`
	`  UINT8 *piecCstNo,`
	`  UINT32 *pTopoCnt,`
	`  const TRDP_LABEL_T carLabel);`	

Abstract	Function to retrieve the leading car depending IEC consist number of the consist hosting a car with label carLabel.	
Parameters	`pIecCstNo`	Pointer to the IEC consist number to be returned
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`carLabel`	Pointer to a car label. NULL means own car.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3.19. tau_addr2CstId

Name	tau_addr2CstId	

Synopsis C	`TRDP_ERR_T tau_addr2CstId (`
	`  TRDP_LABEL_T  cstId,`
	`  UINT32  *pTopoCnt,`
	`  TRDP_IP_ADDR_T  addr);`	

Synopsis C++	`TRDP_ERR_T tau::addr2CstId (`
	`  TRDP_LABEL_T  cstId,`
	`  UINT32  *pTopoCnt,`
	`  TRDP_IP_ADDR_T  addr);`	

Abstract	Function to retrieve the consist identifier of the consist hosting the device with the IP-Address addr.	
Parameters	`cstId`	Pointer to the consist id to be returned.
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`addr`	IP address. 0 means own device, so the own consist id is returned.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3.20. tau_addr2CstNo

Name	tau_addr2CstNo	

Synopsis C	`TRDP_ERR_T tau_addr2CstNo (`
	`  UINT8  pCstNo,`
	`  UINT32  *pTopoCnt,`
	`  TRDP_IP_ADDR_T  addr);`	

Synopsis C++	`TRDP_ERR_T tau::addr2CstNo (`
	`  UINT8  pCstNo,`
	`  UINT32  *pTopoCnt,`
	`  TRDP_IP_ADDR_T  addr);`	

Abstract	Function to retrieve the consist number of the consist hosting the device with the IP-Address addr.	
Parameters	`pCstNo`	Pointer to the consist number to be returned.
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`addr`	IP address. 0 means own device, so the own consist id is returned.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.3.21. tau_addr2IecCstNo

Name	tau_addr2IecCstNo	

Synopsis C	`TRDP_ERR_T tau_addr2IecCstNo (`
	`  UINT8  pCstNo,`
	`  UINT32  *pTopoCnt,`
	`  TRDP_IP_ADDR_T  addr);`	

Synopsis C++	`TRDP_ERR_T tau::addr2IecCstNo (`
	`  UINT8  pCstNo,`
	`  UINT32  *pTopoCnt,`
	`  TRDP_IP_ADDR_T  addr);`	

Abstract	Function to retrieve the leading car depending IEC consist number of the consist hosting the device with the IP-Address addr.	
Parameters	`pCstNo`	Pointer to the IEC consist number to be returned.
	`pTopoCnt`	Pointer to topo counter. Input: If a topo counter value != 0 is given in the call, the value will be checked against the current topo counter value. Output: The current value of the topo counter.
	`addr`	IP address. 0 means own device, so the own consist id is returned.
Returns C/C++	0 if OK, !=0 if error, see chapter 12.1	

14.4. TRDP Marshalling API

14.4.1. Defintions

No specific definitions.

14.4.2. tau_initMarshall

Name	**tau_initMarshall**	

Synopsis C	`TRDP_ERR_T tau_initMarshall(`
	`  void **ppRefCon,`
	`  UINT32 numComId,`
	`  TRDP_COMID_DSID_MAP_T *pComIdDsIdMap,`
	`  UINT32 numDataset,`
	`  TRDP_DATASET_T *pDataset[]);`	

Synopsis C++	`TRDP_ERR_T TAU::initMarshall(`
	`  void **ppRefCon,`
	`  UINT32 numComId,`
	`  TRDP_COMID_DSID_MAP_T *pComIdDsIdMap,`
	`  UINT32 numDataset,`
	`  TRDP_DATASET_T *pDataset[]);`	

Abstract	Initialises marshalling with a given data set list that could be read out of an XML-file. The supplied array must be sorted by ComId's. The array must exist during the use of the marshalling functions (until tlc_terminate()).	
Parameters	`ppRefCon`	Returns a pointer to be used for the reference context of marshalling/unmarshalling
	`numComId`	Number of entries in the ComId DatasetId mapping list.
	`pComIdDsIdMap`	Pointer to the ComId DatasetId mapping list.
	`numDataSet`	Number of datasets in the array referenced with pDataset
	`pDataset`	Pointer to an array of pointers to structures of type TRDP_DATASET_T, see Table 5
Returns C/C++	TRDP_NO_ERR no error TRDP_MEM_ERR provided buffer to small TRDP_PARAM_ERR Parameter error	

14.4.3. tau_marshall

Name	tau_marshall	

Synopsis C	`TRDP_ERR_T tau_marshall(`
	`  void  *pRefCon,`
	`  UINT32  comId,`
	`  const UINT8  *pSrc,`
	`  UINT8  *pDest,`
	`  UINT32  *pDestSize,`
	`  TRDP_DATASET_T **ppDSPointer`
	`);`	

Synopsis C++	`TRDP_ERR_T TAU::marshall(`
	`  void  *pRefCon,`
	`  UINT32  comId,`
	`  const UINT8  *pSrc,`
	`  UINT8  *pDest,`
	`  UINT32  *pDestSize,`
	`  TRDP_DATASET_T **ppDSPointer`
	`);`	

Abstract	Marshalls a data set referenced with comId and packs it into a send buffer.	
Parameters	`pRefCon`	Pointer to user context
	`comId`	ComId for this data
	`pSrc`	Pointer to source data
	`pDest`	OUT: Pointer to destination buffer
	`pDestSize`	IN: size of destination buffer OUT: number of bytes written
	`ppDSPointer`	Pointer to pointer to cached dataset, used to skip dataset search for further calls with the same comId. Set NULL if not used. Set content to NULL if it can't be provided from previous call.
Returns C/C++	TRDP_NO_ERR no error TRDP_MEM_ERR provided buffer to small TRDP_INIT_ERR marshalling not initialized TRDP_COMID_ERR comId not existing	

14.4.4. tau_marshallDs

Name	**tau_marshallDs**	

Synopsis C	`TRDP_ERR_T tau_marshallDs(`
	`  void  *pRefCon,`
	`  UINT32  datasetId,`
	`  const UINT8  *pSrc,`
	`  UINT8  *pDest,`
	`  UINT32  *pDestSize,`
	`  TRDP_DATASET_T **ppDSPointer`
	`);`	

Synopsis C++	`TRDP_ERR_T tau::marshallDs(`
	`  void  *pRefCon,`
	`  UINT32  datasetId,`
	`  const UINT8  *pSrc,`
	`  UINT8  *pDest,`
	`  UINT32  *pDestSize,`
	`  TRDP_DATASET_T **ppDSPointer`
	`);`	

Abstract	Marshalls a data set and packs it into a send buffer.	
Parameters	`pRefCon`	Pointer to user context
	`datasetId`	Dataset Id for this data
	`pSrc`	Pointer to source data
	`pDest`	OUT: Pointer to destination buffer
	`pDestSize`	IN: size of destination buffer OUT: number of bytes written
	`ppDSPointer`	Pointer to pointer to cached dataset, used to skip dataset search for further calls with the same comId. Set NULL if not used. Set content to NULL if it can't be provided from previous call.
Returns C/C++	TRDP_NO_ERR no error TRDP_MEM_ERR provided buffer to small TRDP_INIT_ERR marshalling not initialized TRDP_PARAM_ERR Parameter error	

14.4.5. tau_unmarshall

Name	tau_unmarshall	

Synopsis C	`TRDP_ERR_T tau_unmarshall(`
	`  void  *pRefCon,`
	`  UINT32  comId,`
	`  UINT8  *pSrc,`
	`  UINT8  *pDest,`
	`  UINT32  *pDestSize,`
	`  TRDP_DATASET_T **ppDSPointer`
	`);`	

Synopsis C++	`TRDP_ERR_T TAU::unmarshall(`
	`  void  *pRefCon,`
	`  UINT32  comId,`
	`  UINT8  *pSrc,`
	`  UINT8  *pDest,`
	`  UINT32  *pDestSize,`
	`  TRDP_DATASET_T **ppDSPointer`
	`);`	

Abstract	Unmarshalls the data set referenced with comId and unpacks it into a receiving buffer.	
Parameters	`pRefCon`	Pointer to user context
	`comId`	ComId for this data
	`pSrc`	Pointer to source data
	`pDest`	OUT: Pointer to destination buffer
	`pDestSize`	IN: size of destination buffer OUT: number of bytes written
	`ppDSPointer`	Pointer to pointer to cached dataset, used to skip dataset search for further calls with the same comId. Set NULL if not used. Set content to NULL if it can't be provided from previous call.
Returns C/C++	TRDP_NO_ERR no error TRDP_MEM_ERR provided buffer to small TRDP_INIT_ERR marshalling not initialized TRDP_COMID_ERR comId not existing	

14.4.6. tau_unmarshallDs

Name	**tau_unmarshallDs**	

Synopsis C	`TRDP_ERR_T tau_unmarshallDs(`
	`  void  *pRefCon,`
	`  UINT32  datasetId,`
	`  UINT8  *pSrc,`
	`  UINT8  *pDest,`
	`  UINT32  *pDestSize,`
	`  TRDP_DATASET_T **ppDSPointer`
	`);`	

Synopsis C++	`TRDP_ERR_T tau::unmarshallDs(`
	`  void  *pRefCon,`
	`  UINT32  datasetId,`
	`  UINT8  *pSrc,`
	`  UINT8  *pDest,`
	`  UINT32  *pDestSize,`
	`  TRDP_DATASET_T **ppDSPointer`
	`);`	

Abstract	Unmarshalls a data set and unpacks it into a receiving buffer.	
Parameters	`pRefCon`	Pointer to user context
	`datasetId`	Dataset Id for this data
	`pSrc`	Pointer to source data
	`pDest`	OUT: Pointer to destination buffer
	`pDestSize`	IN: size of destination buffer OUT: number of bytes written
	`ppDSPointer`	Pointer to pointer to cached dataset, used to skip dataset search for further calls with the same comId. Set NULL if not used. Set content to NULL if it can't be provided from previous call.
Returns C/C++	TRDP_NO_ERR no error TRDP_MEM_ERR provided buffer to small TRDP_INIT_ERR marshalling not initialized TRDP_PARAM_ERR Parameter error	

14.4.7. tau_calcDatasetSize

Name	tau_calcDatasetSize	
Synopsis C	`TRDP_ERR_T tau_calcDatasetSize ( void *pRefCon, UINT32 dsId, UINT8 *pSrc, UINT32 *pDestSize, TRDP_DATASET_T **ppDSPointer );`	
Synopsis C++	`TRDP_ERR_T tau::calcDatasetSize ( void *pRefCon, UINT32 dsId, UINT8 *pSrc, UINT32 *pDestSize, TRDP_DATASET_T **ppDSPointer );`	
Abstract	Calculate size of a dataset. For fixed size datasets the size is based on calculated value stored in the TRDP database. For variable size datasets the size is calculated based on current data. **Note:** The source data is expected to be marshalled data, i.e. data from wire.	
Parameters	`pRefCon`	Pointer to user context
	`dsId`	Dataset identifier
	`pSrc`	Pointer to source data
	`pDestSize`	Pointer to the size of the dataset
	`ppDSPointer`	Pointer to pointer to cached dataset, used to skip dataset search for further calls with the same comId. Set NULL if not used. Set content to NULL if it can't be provided from previous call.
Returns C/C++	TRDP_NO_ERR no error TRDP_INIT_ERR marshalling not initialized TRDP_PARAM_ERR Parameter error	

14.4.8. tau_calcDatasetSizeByComId

Name	**tau_calcDatasetSizeByComId**	

Synopsis C	`TRDP_ERR_T  tau_calcDatasetSizeByComId (`
	`  void  *pRefCon,`
	`  UINT32  comId,`
	`  UINT8  *pSrc,`
	`  UINT32  *pDestSize,`
	`  TRDP_DATASET_T **ppDSPointer`
	`);`	

Synopsis C++	`TRDP_ERR_T tau::calcDatasetSizeByComId (`
	`  void  *pRefCon,`
	`  UINT32  comId,`
	`  UINT8  *pSrc,`
	`  UINT32  *pDestSize,`
	`  TRDP_DATASET_T **ppDSPointer`
	`);`	

Abstract	Calculate size of a dataset of the telegram given with comId. For fixed size datasets the size is based on calculated value stored in the TRDP database. For variable size datasets the size is calculated based on current data. **Note:** The source data is expected to be marshalled data, i.e. data from wire.	
Parameters	`pRefCon`	Pointer to user context
	`comId`	ComId of the telegram to calculate the size for
	`pSrc`	Pointer to source data
	`pDestSize`	Pointer to the size of the dataset in the telegram
	`ppDSPointer`	Pointer to pointer to cached dataset, used to skip dataset search for further calls with the same comId. Set NULL if not used. Set content to NULL if it can't be provided from previous call.
Returns C/C++	TRDP_NO_ERR no error TRDP_INIT_ERR marshalling not initialized TRDP_PARAM_ERR Parameter error	

15. Statistics and Diagnostics

TRDP is generating information, warning and error messages for debugging purposes additionally during execution TRDP automatically collects information that can be used for statistic or diagnostic analysis.

Example:

- Device information (uptime, IP address etc.)
- Number of sent and received PD and MD telegrams
- Number of messages with problems (FCS, no listeners, retransmissions etc.)
- Usage of common resources (memory, queues)

This information can be retrieved in two ways:

- Via an API from an application
- Via PD or MD

15.1. Debug Support

To debug TRDP there are debug/trace outputs printed on a terminal and/or to a debug file. The outputs can be implemented and controlled as described below.

The outputs are given to the call back routine that was given as parameter of tlc_init() (see chapter 7.4.4 and 7.4.6). If a NULL pointer was given, there is no output at all. The call back routine itself gets all information back (timestamp, source file name and line number, category and debug message) and can, based on the parameters given in the XML file (see chapter 10.6), filter the output and redirect it to a file.

15.2. Statistic Data

This chapter describes all statistic variables that can be accessed in a TRDP device. All variables are read-only if not specified otherwise.

15.2.1. tlc_getVersion

Name	tlc_getVersion	
Synopsis C	`const TRDP_VERSION_T * tlc_getVersion (void);`	
Synopsis C++	`const TRDP_VERSION_T * tlc::getVersion (void);`	
Abstract	Read TRDP version as TRDP_VERSION_T. Typically version is incremented only after incompatible changes, release after compatible enhancements, update after bug fixes, evolution in development process.	
Parameters	–	-
Returns C/C++	`TRDP_VERSION_T *`	

Type	Name	Description
UINT8	ver	Version - incremented for incompatible changes
UINT8	rel	Release - incremented for compatible changes
UINT8	upd	Update - incremented for bug fixes
UINT8	evo	Evolution - incremented for build

Table 97 Structure `TRDP_VERSION_T – version structure`

15.2.2. tlc_getVersionString

Name	tlc_getVersionString	
Synopsis C	`const char* tlc_getVersionString (void);`	
Synopsis C++	`const char* tlc::getVersionString (void);`	
Abstract	Read TRDP version in human readable format 'vv.rr.uu.ee' with vv-2 digits decimal version, rr-2 digits decimal release, uu-2 digits decimal update, ee-2 digits decimal evolution. Typically version is incremented only after incompatible changes, release after compatible enhancements, update after bug fixes, evolution in development process.	
Parameters	–	-
Returns C/C++	TRDP version in format 'vv.rr.uu.ee'	

15.2.3. tlc_resetStatistics

Name	tlc_resetStatistics	
Synopsis C	`TRDP_ERR_T tlc_resetStatistics (` `  TRDP_APP_T  appHandle);`	
Synopsis C++	`TRDP_ERR_T tlc::resetStatistics (void);`	
Abstract	Reset TRDP statistics.	
Parameters	`appHandle`	(C) handle returned by tlc_openSession()
Returns C	0 if OK, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_NOINIT_ERR application handle invalid	
Returns C++	“TRDPException” according to chapter 12.1.	

15.2.4. tlc_getStatistics

Name	tlc_getStatistics	

Synopsis C	`TRDP_ERR_T tlc_getStatistics (`
	`  TRDP_APP_T  appHandle,`
	`  TRDP_STATISTICS_T *pStatistics);`	

Synopsis C++	`TRDP_ERR_T tlc::getStatistics (` `  TRDP_STATISTICS_T *pStatistics);`	
Abstract	Read TRDP internal statistics. Memory for statistics information will be reserved by tlc layer and needs to be freed by the user.	
Parameters	`appHandle`	(C) handle returned by tlc_openSession()
	`pStatistics`	Pointer to the status and statistics information.
Returns C	0 if OK, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_NOINIT_ERR application handle invalid	

	TRDP_PARAM_ERR invalid parameter
Returns C++	“TRDPException” according to chapter 12.1.

Type	Name	Description
UINT32	Total	Total memory size
UINT32	Free	Free memory
UINT32	minFree	Minimal free memory in statistics interval
UINT32	allocBlocks	Number of allocated blocks
UINT32	allocErr	Number of allocation errors
UINT32	freeErr	Number of freeing errors
UINT32	blockSize[TRDP_MEM_BLK_T]	Memory block sizes
UINT32	usedBlockSize[TRDP_MEM_BLK_T]	Used memory blocks per block size

Table 98 Structure `TRDP_MEM_STATISTICS_T` – memory statistics and configuration information

Type	Name	Description
UINT32	defQos	default QoS for PD
UINT32	defTtl	default TTL for PD
UINT32	defTimeout	Default timeout for PD
UINT32	numSubs	Number of subscribed ComId’s, returned value can be used to calculate the array size for tlc_getSubsStatistics()
UINT32	numPub	Number of published ComId’s, returned value can be used to calculate the array size for tlc_getPubStatistics()
UINT32	numRcv	number of received PD packets
UINT32	numCrcErr	number of received PD packets with CRC error
UINT32	numProtErr	number of received PD packets with protocol error
UINT32	numTopoErr	number of received PD packets with wrong topo count
UINT32	numNoSubs	number of received PD push packets without subscription
UINT32	numNoPub	Number of received PD pull packets without publisher
UINT32	numTimeout	number of PD timeouts
UINT32	numSend	number of sent PD packets

Table 99 Structure `TRDP_PD_STATISTICS_T` – PD statistics and configuration information

Type	Name	Description
UINT32	defQos	default QoS for MD
UINT32	defTtl	default TTL for MD
UINT32	defReplyTimeout	Default reply timeout for MD
UINT32	defConfirmTimeout	Default confirm timeout for MD
UINT32	numList	Number of Listeners
UINT32	numRcv	number of received MD packets
UINT32	numCrcErr	number of received MD packets with CRC error
UINT32	numProtErr	number of received MD packets with protocol error
UINT32	numTopoErr	number of received MD packets with wrong topo count
UINT32	numNoListener	number of received MD packets without listener, returned value can be used to calculate the array size for tlc_getUdp/TcpListStatistics
UINT32	numReplyTimeout	number of MD reply timeouts
UINT32	numConfirmTimeout	number of MD confirm timeouts
UINT32	numSend	number of sent MD packets

Table 100 Structure `TRDP_MD_STATISTICS_T` – MD statistics and configuration information

Type	Name	Description
UINT32	version	TRDP version
TRDP_TIME_T	timestamp	actual time stamp
UINT32	uptime	time in seconds since last initialization
UINT32	statisticTime	time in seconds since last reset of statistics
TRDP_LABEL_T	hostname	Own host name
TRDP_LABEL_T	leaderName	leader host name
TRDP_IP_ADDR_T	ownIpAddr	own IP address
TRDP_IP_ADDR_T	leaderIpAddr	leader IP address
UINT32	processPrio	priority of TRDP process
UINT32	processCycle	cycle time of TRDP process in microseconds
UINT32	numJoin	Number of joined multicast IP addresses, returned value can be used to calculate the array size for tlc_getJoinStatistics()
UINT32	numRed	Number of redundancy groups, returned value can be used to calculate the array size for tlc_getRedStatistics()
TRDP_MEM_STATISTICS_T	mem	Memory statistics
TRDP_PD_STATISTICS_T	Pd	PD statistics
TRDP_MD_STATISTICS_T	udpMd	UDP MD statistics
TRDP_MD_STATISTICS_T	tcpMd	TCP MD statistics

Table 101 Structure `TRDP_STATISTICS_T` – global statistics and configuration information

15.2.5. tlc_getSubsStatistics

Name	tlc_getSubsStatistics	

Synopsis C	`TRDP_ERR_T tlc_getSubsStatistics (`
	`  TRDP_APP_T  appHandle,`
	`  UINT16  *pNumSubs,`
	`  TRDP_SUBS_STATISTICS_T  *pStatistics);`	

Synopsis C++	`TRDP_ERR_T tlc::getSubsStatistics (`
	`  UINT16  *pNumSubs,`
	`  TRDP_SUBS_STATISTICS_T  *pStatistics);`	

Abstract	Read TRDP internal PD subscribe statistics. Memory for statistics information must be provided by the user. The reserved length is given via pNumSub implicitly.	
Parameters	`AppHandle`	(C) handle returned by tlc_openSession()
	`PNumSubs`	Pointer to the number of subscriptions
	`pStatistics`	Pointer to a list with the subscription status and statistics information.
Returns C	0 if OK, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_NOINIT_ERR handle invalid TRDP_PARAM_ERR invalid parameter TRDP_MEM_ERR there are more data than requested	
Returns C++	“TRDPException” according to chapter 12.1.	

Type	Name	Description
UINT32	comId	Subscribed ComId
TRDP_IP_ADDR_T	joinedAddr	Joined IP address
TRDP_IP_ADDR_T	filterAddr	Filter IP address, i.e. IP address of the sender for this subscription, 0.0.0.0 in case all senders.
UINT32	callBack	Reference for call back function if used
UINT32	timeout	Time-out value in us. 0 = No time-out supervision
TRDP_ERR_T	status	Receive status information TRDP_NO_ERR, TRDP_TIMEOUT_ERR
TRDP_TO_BEHAVIOR_T	toBehav	Behaviour at time-out. Set data to zero / keep last value
UINT32	numRecv	Number of packets received for this subscription

Table 102 Structure `TRDP_SUBS_STATISTICS_T` – PD subscription statistics information

15.2.6. tlc_getPubStatistics

Name	**tlc_getPubStatistics**	

Synopsis C	`TRDP_ERR_T tlc_getPubStatistics (`
	`  TRDP_APP_T  appHandle,`
	`  UINT16  *pNumPub,`
	`  TRDP_PUB_STATISTICS_T  *pStatistics);`	

Synopsis C++	`TRDP_ERR_T tlc::getPubStatistics (`
	`  UINT16  *pNumPub,`
	`  TRDP_PUB_STATISTICS_T  *pStatistics);`	

Abstract	Read TRDP internal PD publish statistics. Memory for statistics information must be provided by the user. The reserved length is given via pNumPub implicitly.	
Parameters	`appHandle`	(C) handle returned by tlc_openSession()
	`pNumPub`	Pointer to the number of publishers
	`pStatistics`	Pointer to a list with the publish status and statistics information
Returns C	0 if OK, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_NOINIT_ERR handle invalid TRDP_PARAM_ERR invalid parameter TRDP_MEM_ERR there are more data than requested	
Returns C++	"TRDPException" according to chapter 12.1.	

Type	**Name**	**Description**
UINT32	comId	Subscribed ComId
TRDP_IP_ADDR_T	destAddr	IP address of destination for this publishing
UINT32	cycle	Publishing cycle in µs
UINT32	redId	Redundancy group id
UINT32	redState	Redundant state.!0=Leader or 0=Follower
UINT32	numPut	Number of packet updates
UINT32	numSend	Number of packets sent for this publish

Table 103 Structure `TRDP_PUB_STATISTICS_T` – PD publish statistics information

15.2.7. tlc_getUdpListStatistics

Name	**tlc_getUdpListStatistics**	

Synopsis C	`TRDP_ERR_T tlc_getUdpListStatistics (`
	`  TRDP_APP_T  appHandle,`
	`  UINT16  *pNumList,`
	`  TRDP_LIST_STATISTICS_T  *pStatistics);`	

Synopsis C++	`TRDP_ERR_T tlc::getUdpListStatistics (`
	`  UINT16  *pNumList,`
	`  TRDP_LIST_STATISTICS_T  *pStatistics);`	

Abstract	Read TRDP internal UDP MD listener statistics. Memory for statistics information must be provided by the user. The reserved length is given via pNumLis implicitly.	
Parameters	`appHandle`	(C) handle returned by tlc_openSession()
	`pNumList`	Pointer to the number of listeners
	`pStatistics`	Pointer to a list with the listener status and statistics information
Returns C	0 if OK, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_NOINIT_ERR handle invalid TRDP_PARAM_ERR invalid parameter	

	TRDP_MEM_ERR	there are more data than requested
Returns C++	"TRDPException" according to chapter 12.1.	

Type	Name	Description
UINT32	comId	Subscribed ComId
TRDP_URI_USER_T	uri	URI to listen to (user part)
TRDP_IP_ADDR_T	joinedAddr	Joined IP address
UINT32	callback	Call back function reference if used
UINT32	queue	Queue reference if used
UINT32	userRef	User reference if used
UINT32	numRecv	Number of received packets

Table 104 Structure `TRDP_LIST_STATISTICS_T` – MD listener statistics information

15.2.8. tlc_getTcpListStatistics

Name	tlc_getTcpListStatistics	

Synopsis C	`TRDP_ERR_T tlc_getTcpListStatistics (`
	`  TRDP_APP_T  appHandle,`
	`  UINT16  *pNumList,`
	`  TRDP_LIST_STATISTICS_T  *pStatistics);`	

Synopsis C++	`TRDP_ERR_T tlc::getTcpListStatistics (`
	`  UINT16  *pNumList,`
	`  TRDP_LIST_STATISTICS_T  *pStatistics);`	

Abstract	Read TRDP internal TCP MD listener statistics. Memory for statistics information must be provided by the user. The reserved length is given via pNumLis implicitly.	
Parameters	`appHandle`	(C) handle returned by tlc_openSession()
	`pNumList`	Pointer to the number of listeners
	`pStatistics`	Pointer to a list with the listener status and statistics information
Returns C	0 if OK, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_NOINIT_ERR handle invalid TRDP_PARAM_ERR invalid parameter TRDP_MEM_ERR there are more data than requested	
Returns C++	"TRDPException" according to chapter 12.1.	

15.2.9. tlc_getRedStatistics

Name	tlc_getRedStatistics	
Synopsis C	TRDP_ERR_T tlc_getRedStatistics (TRDP_APP_T appHandle, UINT16 *pNumRed, TRDP_RED_STATISTICS_T *pStatistics);	
Synopsis C++	TRDP_ERR_T tlc::getRedStatistics (UINT16 *pNumRed, TRDP_RED_STATISTICS_T *pStatistics);	
Abstract	Read TRDP internal redundancy group statistics. Memory for statistics information must be provided by the user. The reserved length is given via pNumRed implicitly.	
Parameters	appHandle	(C) handle returned by tlc_openSession()
	pNumRed	Pointer to the number of redundancy groups
	pStatistics	Pointer to a list with the id and status information of the redundancy group
Returns C	0 if OK, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_NOINIT_ERR handle invalid TRDP_PARAM_ERR invalid parameter TRDP_MEM_ERR there are more data than requested	
Returns C++	“TRDPException” according to chapter 12.1.	

Type	Name	Description
UINT32	Id	Redundancy id
TRDP_RED_STATE_T	state	Leader/Follower

Table 105 Structure `TRDP_RED_STATISTICS_T` – redundancy statistics information

15.2.10. tlc_getJoinStatistics

Name	tlc_getJoinStatistics	
Synopsis C	TRDP_ERR_T tlc_getJoinStatistics (TRDP_APP_T appHandle, UINT16 *pNumJoin, UINT32 *pIpAddr);	
Synopsis C++	TRDP_ERR_T tlc::getRedStatistics (UINT16 *pNumJoin, UINT32 *pIpAddr);	
Abstract	Read TRDP internal multicast join statistics. Memory for statistics information must be provided by the user. The reserved length is given via pNumJoin implicitly.	
Parameters	appHandle	(C) handle returned by tlc_openSession()
	pNumJoin	Pointer to the number of joined multicast IP adresses
	pStatistics	Pointer to a list with the joined multicast IP adresses
Returns C	0 if OK, !=0 if error, see chapter 12.1. TRDP_NO_ERR no error TRDP_NOINIT_ERR handle invalid TRDP_PARAM_ERR invalid parameter TRDP_MEM_ERR there are more data than requested	
Returns C++	“TRDPException” according to chapter 12.1.	

16. Installation & Integration

16.1. Targets

TRDP is shipped as source code and supports the following set of primary targets.

Target	OS	CPU + HW	Description
WIN32		x86	WindowsXP 32 bit on x86
		x86	Windows732 bit on x86
POSIX	__APPLE__		Mac OS X on x86
	__linux__		Linux
	__QNXNTO__	Neutrino	QNX on Neutrino
VXWORKS		PowerPC	VxWorks on PowerPC

Table 106 TRDP primary targets

16.2. TRDP Deliverables

The tables below show the directories for all supported targets. The files are described in the release notes.

16.2.1. Target Independing Files

Directory	Use
..\src\api	TRDP API header files
..\src\common	TRDP source code files
..\src\example	TRDP example files
..\src\vos\api	VOS API header files
..\src\vos\common	VOS target independing source code files

Table 107 TRDP output file formats

16.2.2. Target Specific Files

Directory	Use
..\src\vos\posix	VOS POSIX depending source code files
..\src\vos\windows	VOS WIN32 depending source code files
..\src\vos\vxworks	VOS VXWORKS depending source code files

Table 108 TRDP target specific files

16.2.3. Spy Files

Directory	Use
..\spy\src	Wireshark plugin source code files
..\spy\doc	Wireshark plugin documentation
..\spy\linux32	Linux32 wireshark plugin (SO)
..\spy\win32	Win32 wireshark plugin (DLL))

Table 109 TRDP spy files

16.2.4. Configuration

Directory	Use
..\config	XML scheme and bulid settings

Table 110 TRDP configuration files

16.2.5. Resources

Directory	Use
..\resources\windows\getopt	getopt implementation for Windows. Used in TRDP test programs.
..\resources\windows\iconv-1.9.2	iconv implementation for Windows. Used for Wireshark plugin and tau_xml.
..\resources\windows\libxml	libxml implementation for Windows. Used for Wireshark plugin and tau_xml.
..\resources\windows\pthread	pthread implementation for Windows. Used in VOS functionality.
..\resources\windows\wireshark	Wireshark 1.8.3 for Windows.

Table 111 TRDP target specific files

16.2.6. Build environment

Directory	Use
..\	Makefile – to be configured by make config
..\VisualC	VisualC 2010 configuration for TRDP library and related test examples
..\XCode	Xcode configuration for TRDP library

Table 112 TRDP target specific files

16.2.7. Tests

Directory	Use
..\test\diverse	Test diverse functions of the library
..\test\laddermdtest	Test of the TRDP ladder MD functionality
..\test\ladderpdtest	Test of the TRDP ladder PD functionality
..\test\lint	PCLint profile for Windows
..\test\marshalling	Test of the TRDP marshalling functionality

Directory	Use
..\test\mdpatterns	Test of the TRDP MD patterns
..\test\pdpatterns	Test of the TRDP PD patterns
..\test\udpmdcom	Test of the UDP MD communication
..\test\xml	Test of the TRDP XML configuration

Table 113 TRDP tests

16.2.8. Examples

Directory	Use
..\examples	Examples for the use of TRDP

Table 114 TRDP examples

16.3. 64 bit Data Types

Most compilers align data up to 4 bytes with natural alignment, i.e. data is aligned depending on its size. A 4 byte data is aligned to an address that is a multiple of 4.

Alignment of data with sizes larger than 4 bytes, e.g. 64 bit integers or real, is handled in different ways depending on operating systems. For an 8 byte data Windows by default aligns data within a structure to 8 byte alignment but the structure itself is aligned to 4 byte address. Linux for x86 aligns 8 byte data to 4 byte alignment everywhere.

Note: TRDP requires that the generating environment is set up to align 8 byte data to 4 byte aligned addresses.

16.4. MS Windows Patches

Note: Windows XP users must add a registry entry in order for the QoS settings to be set on the IP-packets, see http://support.microsoft.com/kb/248611.

Note: In some cases Windows XP does not correctly respond to IGMP V3 messages regarding multicast. In that case, please see http://support.microsoft.com/kb/815752 and follow the instructions to add the registry key “IGMPVersion” with the value 3, which will force Windows XP to use IGMP V2.

17. TRDP Configuration

17.1. TRDP Configuration Rules

- The in the standard defined ports must be used if interoperability is required
- The TRDP provided memory should be calculated well to fit to the use case regarding size and blocks. For receiving PD and MD the following resources excl. transferred data (n) are used:
    - 12208 bytes per TRDP session,
      the required size can be reduced by using less than the 80 sockets foreseen (116 bytes per socket). For MD_SUPPORT==0 only 4 sockets are used . So only 1712 bytes are needed per TRDP session.
    - 1608 bytes per PD subscriber
    - 1656 bytes per PD publisher
    - 284+n bytes per MD caller session (tlm_request(), tlm_notify())
    - 112+n bytes per MD reply (tlm_reply())
    - 72 bytes per MD listener (md_addListener())
    - 172+n bytes per MD replier session (receiving a request or a notification)
- Different processes on a device must use different TRDP sessions with different IP addresses.
- For TRDP error reporting and debug support the application should provide a print function in tlc_init().

```
void dbgOut (
    void        *pRefCon,
    TRDP_LOG_T  category,
    const CHAR8 *pTime,
    const CHAR8 *pFile,
    UINT16      LineNumber,
    const CHAR8 *pMsgStr)
{
    const char *catStr[] = {"ERR :", "WARN:", "INFO:", "DGB :"};

    /* excluded categories *!
    if (category != VOS_LOG_DBG)
    {
        if (pLogFile != NULL)
        {
            fprintf(pLogfile,
               "%s %s %s %s:%d ",
               pTime,
               catStr[category],
               pMsgStr,
               pFile,
               LineNumber,
               );
            fflush(plogfile);
        }
    }
}
```

- Marshalling/unmarshalling should use the provided stubs in tlc_openSession() to prevent unnecessary copying of the data (received data are only copied once in tlp_get(), data to send are only copied once in tlp_publish(), tlp_put(), tlp_request(), tlm_notify(), tlm_request(), tlm_reply(), tlm_replyQuery(), tlm_confirm()). Only these functions are using the provided stubs for marshalling,

unmarshalling. Within the callback routines for received PD and MD the wire data are provided and the application is responsible for further unmarshalling.

- Safe data transmission support for PD should use the provided stub in tlc_openSession() to prevent unnecessary copying of the data (received data are only copied once in tlp_get(), data to send are only copied once in tlp_publish(), tlp_put(), tlp_request()). Only these functions are using the provided stubs for safe data transmission support. Within the callback routines for received PD the wire data are provided and the application is responsible for safe data transmission support.

17.2. TRDP Compiler Switches

In this chapter all mandatory and optional defines to adapt TRDP are described.

The compiler switches in the following table shall be set to define the target.

Define	Default	Use
WIN32	undefined	Windows 32 bit target
POSIX	undefined	Posix target
VXWORKS	undefined	VxWorks target
__linux__	undefined	To be set additionally to POSIX to handle the correct target OS.
__APPLE__	undefined	To be set additionally to POSIX to handle the correct target OS.
__QNXNTO__	undefined	To be set additionally to POSIX to handle the correct target OS.

Table 115 TRDP target compiler switches

The defines in the following table can be set to modify the behaviour of the TRDP

Define	Default	Use
MD_SUPPORT	undefined	Compiler switch, if not defined or 0, message data communication is not supported.
TRDP_PD_UDP_PORT	20548	PD UDP receive port
TRDP_MD_UDP_PORT	20550	Default MD UDP receive port
TRDP_MD_TCP_PORT	20550	Default MD TCP receive port
VOS_MAX_NUM_IF	4	Number of supported Ethernet interfaces
VOS_MAX_IF_NAME_SIZE	16	Ethernet interface name length
VOS_MAX_NUM_UNICAST	10	Number of supported host IP addresses
VOS_MAX_SOCKET_CNT	80 (4)	Maximum number of concurrent useable sockets per TRDP session. Without MD_SUPPORT already limited to 4.
VOS_MAX_MULTICAST_CNT	20 (10)	Maximum number of multicast groups a socket can join. Without MD_SUPPORT already limited to 10.
VOS_DEFAULT_IFACE	“eth0”	Interface to retrieve the MAC address from to calculate the UUID.
VOS_SOCKBUF_SIZE	64k (8k)	Defines the minimal size for socket send and receive buffers. Without MD_SUPPORT already limited to 8k.

Table 116 TRDP behaviour compiler switches and defines

The defines in the following table can be set to modify the default values used by tau_xml.c for SDT configuration.

Define	Default	Use
TRDP_SDT_DEFAULT_SMI2	0	Default SDT safe message identifier
TRDP_SDT_DEFAULT_NRXSAFE	3	TRDP_SDT_DEFAULT_NRXSAFE
TRDP_SDT_DEFAULT_NGUARD	100	Default SDT initial timeout cycles
TRDP_SDT_DEFAULT_CMTHR	10	Default SDT channel monitoring threshold

Table 117 SDT behaviour compiler defines

17.3. TRDP Code Size

Due to its modularity TRDP can be downsized from full functionality with ~9500 LOC to ~5000 LOC for only PD functionality (without MD, TAUMarshall, TAUXML). The following table gives a small overview (LOC taken from TRDP 1.0.0.0).

File	Blank	Comment	Code
trdp_mdcom.c	370	423	2056
trdp_pdcom.c	109	230	636
trdp_if.c	270	650	1659
trdp_utils.c	119	273	642
trdp_stats.c	62	150	265
trdp_dllmain.c	6	26	21
tau_xml.c	201	379	984
tau_marshall.c	166	307	920
vos_mem.c	101	230	450
vos_utils.c	30	72	207
vos_tread.c (POSIX)	156	304	709
vos_sock.c (POSIX)	154	359	904
vos_shared_mem.c (POSIX)	19	67	84
vos_tread.c (WIN32)	154	302	714
vos_sock.c (WIN32)	178	362	870
vos_shared_mem.c (WIN32)	19	67	83

Table 118 TRDP code size

17.4. TRDP Configuration Example

Examples of source code using the PD and MD API are provided in the TRDP SDK release.

18. TRDP Test Environment

Intentionally left blank.

19. TRDP Adaptation

19.1. Build Environment Adaptation

Within the SDK the build environment for XCode, VisualC2010 and a makefile is provided.

The following check list shall be used to adapt TRDP build environment (makefile) to Your system:

1. Edit 'buildsettings_%TARGET%_TEMPLATE' in 'config' folder and change its file name (e.g. to 'buildsettings_%TARGET%'), so that it is not overwritten on svn update. Adapt the specified paths to your build system environment.
2. Apply build settings in build shell using 'source config/buildsettings_%TARGET%'
3. Make configuration settings for target. These are stored in the files ending with '_config' in the 'config' folder (e.g. POSIX_X86_config or VXWORKS_PPC_config). Use 'make POSIX_X86_config' which copies the settings for the specified target to 'config/config.mk', which is then included automatically every time make is called.
4. Steps 1-3 need to be done once. Only Step 2 must be repeated every time the shell is changed / closed
5. Use 'make help' to view available parameters and further make information. As the target has already been defined by the config settings in Step 3, no more information needs to be passed to make.

19.2. Adaptation for Further Targets

TRDP can be adapted easily to further targets due to its modular structure. All target depending functions and defines are in the vos_ header and source files.

The following check list shall be used for the adaptation to other targets:

1. Define a new compiler switch “XXXX” for a complete new OS or “__XXXX__” for an OS using the Posix interface
2. Extend the includes and type defines in vos_types.h for the new target where necessary
3. Extend the includes in vos_sock.h, vos_thread.h and vos_shared_mem.h for the new target where necessary
4. If the new target supports POSIX interface extend the vos-files for Posix where necessary.
5. If a complete new OS shall be supported create a new directory below “src/vos” and provide the source and header files vos_shared_mem.c, vos_sock.c, vos_thread.c, vos_private.h for the new OS.
6. Adapt the makefile.

20. Performance – to be updated

Tests have been performed to find out how fast TRDP can send PD data on different targets with smaller and larger datasets. The table below gives an estimate of the maximum number of datasets that can be sent per time period, i.e. cycle time of PD process. Note that the measured maximum will give a heavy CPU load and that the possible sending of PD data is a lot lower as there will also be a need for other task to be executed.

The measurements have been done on target computers with no other activity than the test programs. The numbers will of course decrease if the computer is heavily loaded.

One observation was that much of the CPU load for sending was generated from the Ethernet stack.

Target	**VxWorks on VCU-C**		**Linux on VCU-C**		**Windows (x86)**	
Cycle time	**10 ms**	**1 s**	**10 ms**	**1 s**	**10 ms**	**1 s**
10 bytes	41	4100	58	5800	383	38300
1000 bytes	34	3400	39	3900	75	7500

Table 119 TRDP performance (Only sending PD with a very small application)

From the measurements it can be seen that the frequency of sending is more important than the size of each message. From this some guidelines can be deduced:

Avoid many small datasets which each are sent often.

It is better to join small datasets into larger ones.

Receiving messages will also generate CPU load even for message without any MD listener or PD subscriber. Also messages sent as multicast will be received in the own device if the corresponding multicast IP address is joined. Use the statistics to check if there are many unwanted messages received.

Avoid a system sending a lot of unnecessary traffic, e.g. by using “grpall” when the messages not are necessary for all devices.

It is better to use unicast or other groups, than “grpall”, only joined by devices interested in the send messages.

21. Contact Addresses

If you want to come into contact with us please write an email to

info@tcnopen.org

or visit our Web site at

www. tcnopen.org

If you need any information about the product please contact the Hotline

Country:

Phone:

E-mail: