TCNOpen	**Open Source Developement**

TRDP

Train Real Time Data Protocol

System Architecture & Design Specification

Document reference no: TCN-TRDP2-D-BOM-O19-04

Author:	Bernd Löhr, Armin-Hagen Weiss
Organisation:	Bombardier, NewTec
Document date:	06 August 2013
Revision:	4
Status:	draft

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

Participants		
Name and Surname	Organisation	Role
Bernd Löhr	NewTec GmbH / Bombardier Transportation	Designer, Implementer, Author
Mikel Korta	CAF	
Simone Pachera	Saira Electronics	

History			
V1	31 May 12	Bernd Löhr	Initial version
V2	10 July 13	Bernd Löhr	Added more VOS, Statistics, Marshalling functional descriptions and diagrams. Added build settings for make, Lint, Visual C and Xcode
V3	01 Aug 13	Armin-H. Weiss	Added MD part
V4	09 Aug 13	Armin-H. Weiss	Editorial Changes

Table of Contents

Index of Figures

Index of Tables

1. Introduction

1.1. Purpose

This document describes the basic design of the TRDP protocol stack and may aid in extending and including further protocol features apart from process data handling.

1.2. Intended Audience

The intended audience is the programmers, who want to add additional features and/or want to support more target systems.

1.3. References/Related Documents

Reference	Number	Title
[1]	IEC51375-2-3	TRDP Protocol (Annex A)
[2]	TCN-TRDP2-D-BOM-033-xx	TRDP Reference Manual (generated from source code by Doxygen)
[3]	RFC4122	UUID (http://www.ietf.org/rfc/rfc4122.txt)

Table 1: References

1.4. Abbreviations and Definitions

Abbreviation	Definition
API	Application Programming Interface
ComID	Communication Identifier – determines message content (PD and MD)
IDE	Integrated Development Environment
MD	Message Data
PD	Process Data
POSIX	Portable Operating System Interface for uniX
OS	Operating System
QoS	Quality of Service (= Type of Service, TOS)
TAU	TRDP Application Utility (Function calling prefix)
TCP	Transmission Control Protocol
TLC	TRDP Light Common (Function calling prefix)
TLP	TRDP Light Process data (Function calling prefix)
TLM	TRDP Light Message data (Function calling prefix)
TOS	Type of Service, Ethernet header value
TTL	Time To Live
UDP	User Datagram Protocol
UUID	Universally Unique IDentifier
VOS	Virtual Operating System

Table 2: Abbreviations and Definitions

2. System Description

2.1. Overview

The TRDP prototype stack consists of the following functional groups:

- Application/Session layer handling
- Process Data handling (publish/subscribe)
- Optional Message Data handling
- Statistics handling
- Virtual Operating System
- Utilities

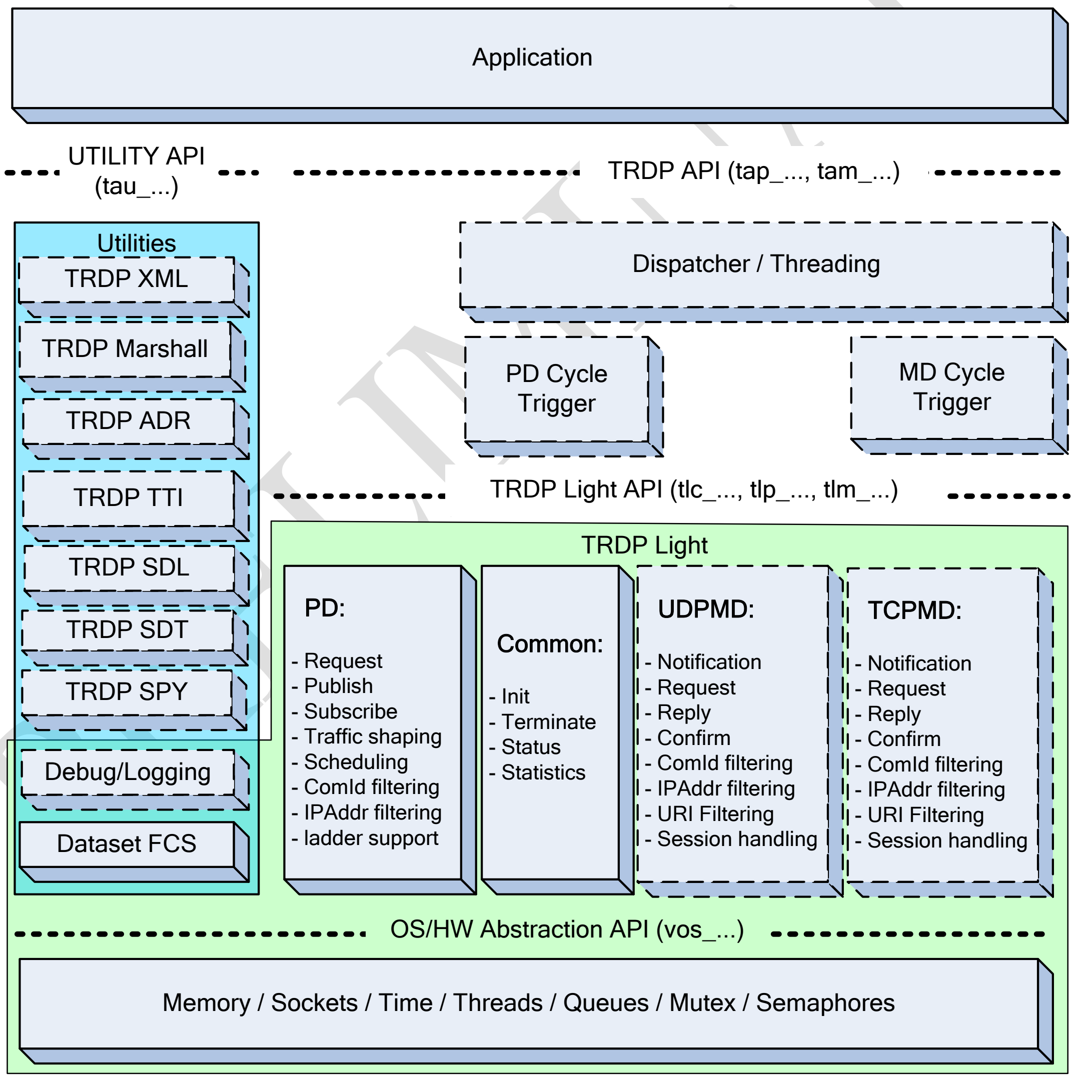

Figure 1: TRDP Functional Layers

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 or converting IP/URI addresses. So TRDP is providing scalability for low end devices using only TRDP Light and 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.

2.2. Application Session Layer Handling

The application session layer functions handle the basic resource management – they register a session of a client application and control the overall behaviour and options for that session. The same application can have several sessions open at the same time.

Note: The term ‘session’ can either apply to the application’s session with the TRDP stack, or to a message data communication session (2.4.3). Within this chapter 2.2 the term ‘session’ will refer to the application’s session.

Each session is internally represented by an instance of the data object `TRDP_SESSION_T`. Sessions are maintained by a singly linked list and are protected by mutexes.

The first function an application must call before any other call to the protocol stack can succeed is `tlc_init()`.
`tlc_init()` will initialize the VOS memory subsystem using the supplied memory configuration parameters and will create a recursive mutex to protect concurrent access to the session queue. The selected memory scheme will be valid for all subsequently opened application sessions until `tlc_terminate()` is called.
`tlc_terminate()` will close all open sessions, release the allocated memory area and delete the mutex.
`tlc_openSession()` creates a new session element using the given process configuration paramaters such as reusing addresses/ports, blocking/non blocking mode and required traffic shaping, initializes it with the supplied default parameters for PD, MD and marshalling and creates a recursive session mutex to protect concurrent access to the session data.

On successful initialization a session handle will be returned and will be used by the caller to select the right session when calling all session related functions (`tlp_`..., `tlm_`..., `tlc_setTopoCount`). If the application quits or doesn't need the session anymore, it should close it by calling `tlc_closeSession()` then `tlc_terminate()`.
The session handle is actually the opaque pointer to the memory area reserved for the session's internal management data (`TRDP_SESSION_T`) and must be checked for validity before dereferencing (using `isValidSession()`).

To allow different packet options mainly for PD, like different TTLs and QoS values, and opening sockets is quite expensive, the number of concurrent sockets should be limited. In `tlc_openSession()` a pool of socket descriptors is created and maintained. If a new publish call is done, the pool of already open sockets is searched for the same send parameters and source address. Either a new or an already open socket will be returned and a reference counter incremented. Handling of the socket pool is done through the functions `trdp_initSockets()`, `trdp_requestSocket()` and `trdp_releaseSocket()` in `trdp_utils.c`. A maximum of 80 sockets per session may be handled by default. The actual usable maximum number of sockets depends on the selected target system (VOS) and might probably be less. The VOS defined `VOS_MAX_SOCKET_CNT` determines the maximum socket count.

Each application session will handle network traffic on one selected interface, determined by its IP address, only. This allows for multi-session, multi-homing usage (but it might not be supported by every target). If a target does not support a certain option at runtime, a referring error message will be generated.

Figure 2 shows the basic data layout of a two-session application. The first session already has two PD publishers, two PD subscribers and two MD Listeners allocated. As well there are two open caller sessions (MD send queue) and two open replier sessions (MD receive queue) as well as a just received MD telegram. The second application session has none (Actually, every application session initially has one publisher for statistical data (ComID 35) and one subscription to ComID 31. See 2.6)

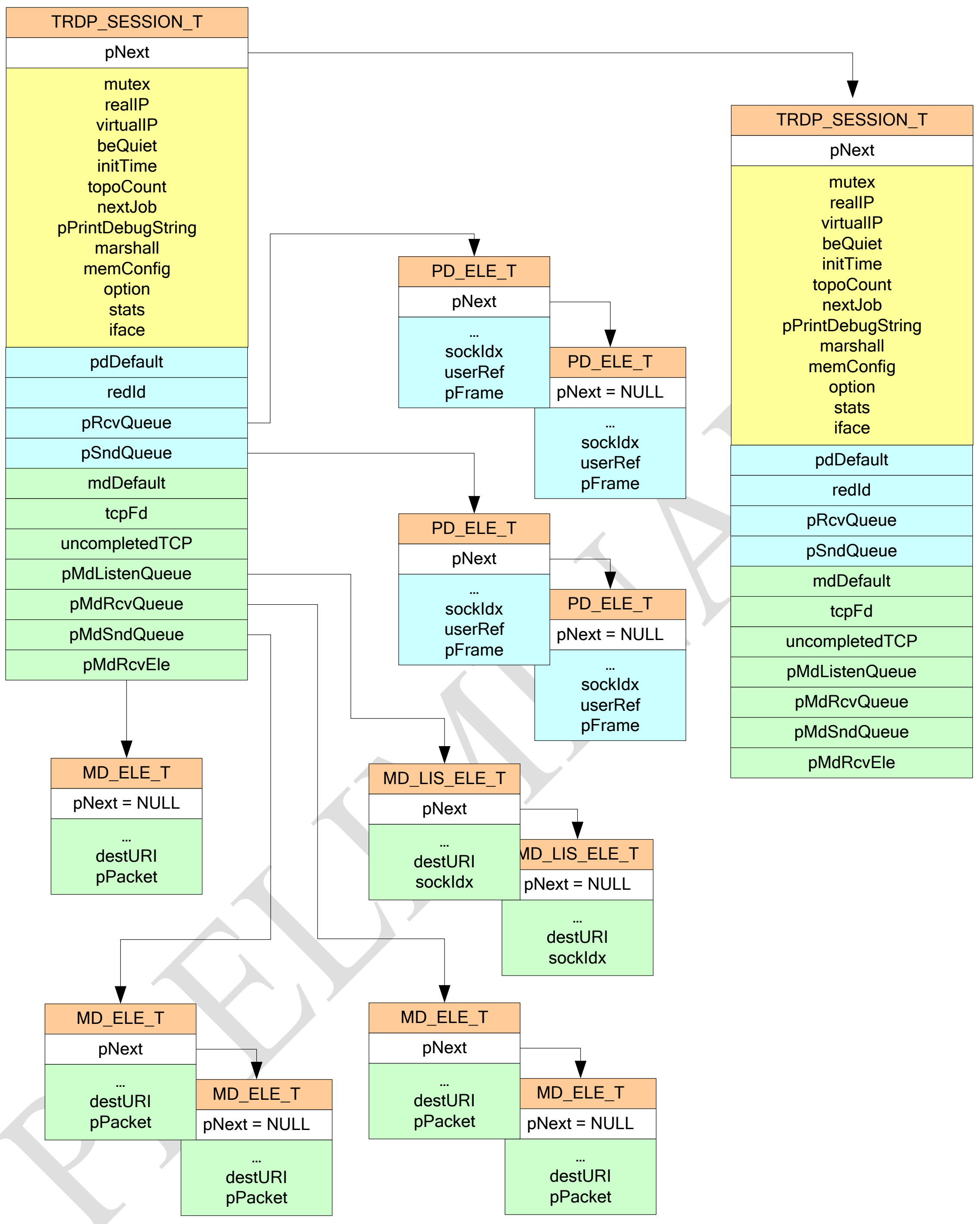

Figure 2: Main Data Structures

There are two additional, session related functions to mention:
`tlc_reinitSession` should be called by the application when it detects a link-up event. It will re-join any multicast groups, which might have been dropped in between.
`tlc_setTopoCount` will set a new topology counter to be used to verify incoming traffic.

2.3. Processing

`tlc_process()` is the main work function. It is the only function where sending and receiving takes place. Depending on the selected mode, whether blocking/non-blocking, using the `select()` function or polling, receiving process and message data is done.
`tlc_process()` has to be called at least at the lowest time interval that has been defined when publishing process data. The send queue is checked for the next pending packet – if it is due, `trdp_pdSend()` or `trdp_mdSend()` will be called.
Next job is to look for timeouts of subscribed packets. If a late packet is detected, the `PD_ELE` entry is marked as overdue and, if callbacks are enabled, the supplied user function will be called.
If blocking mode is on (`select()` is used), the receive queue is scanned for the ready socket descriptor and `trdp_pdReceive()` is called, getting new packet data. Again, the user-supplied callback may be called.
After process data has been handled, optionally message data will be processed depending on the state of the transaction kept in the relevant `mdQueue` element.

Note: Callbacks will only be called from within `tlc_process()`. To avoid recursion, `tlc_process()` must not be called from within a callback.

To support the application in determining the right time to call `tlc_process()` and to minimize CPU usage through polling, the function `tlc_getInterval()` checks all queues for pending packets to send or receive and computes and returns the next time when `tlc_process()` must be called before any timing constraints may hit. The application should always call `tlc_getInterval()`, wait for the amount of the reported time and then call `tlc_process()` in its main loop.

The sample calling sequences in shows a basic functional flow for a publisher only. Before entering the main loop, the application has to open a session and publishes PD. The process data will not be sent immediately but a `PD_ELE_T` will be queued into the send queue. The send queue will be searched for pending packets in `tlc_getInterval()` to set the due time. The returned time value should be used by the application to wait before calling `tlc_process()`. If `MD_SUPPORT==1`, message data session timeouts must be considered as well.

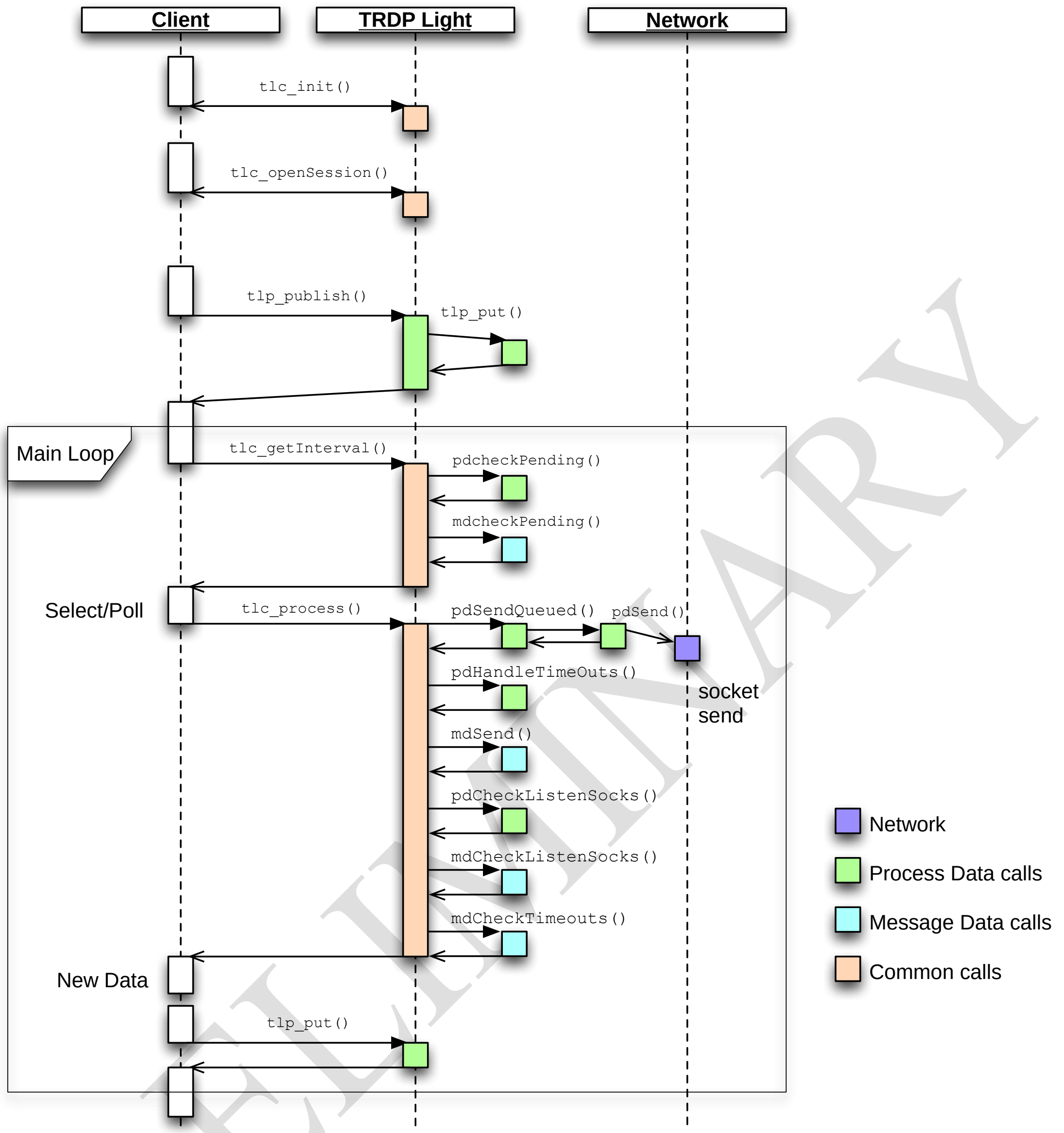

Figure 3: Send and Receive Message and Process Data

When receiving data (subscribe), data can be fetched either by using callbacks (out of `tlc_process()`) or at any time by polling using `tlp_get()`.
 shows a basic sending and receiving sequence of process data:

- `pdCheckPending()` and `mdCheckPending()` return in a file descriptor set which sockets are used and the minimum time when the next packets are due to be sent.
- `pdSendQueued()` walks through the list of published PDs and calls the socket send function (through `trdp_pdSend()` and `vos_sockSendUDP`)
- `pdHandleTimeOuts()` detects missing PDs and might call back to indicate a time-out error
- `pdCheckListenSocks()` checks the supplied file descriptor set for read-ready sockets and reads the packet from the socket (via `pdReceive()` and `vos_sockReceiveUDP()`). In case of receive errors like FCS mismatch, I/O error

etc. the user supplied callback function will be called. Internally the error will be saved in the `PD_ELE_T` anyway.

- The `md`-prefixed functions will basically do the same for message data, except that there will be a listener queue, a receive queue and a send queue with special handling for TCP and UDP based communication sessions.

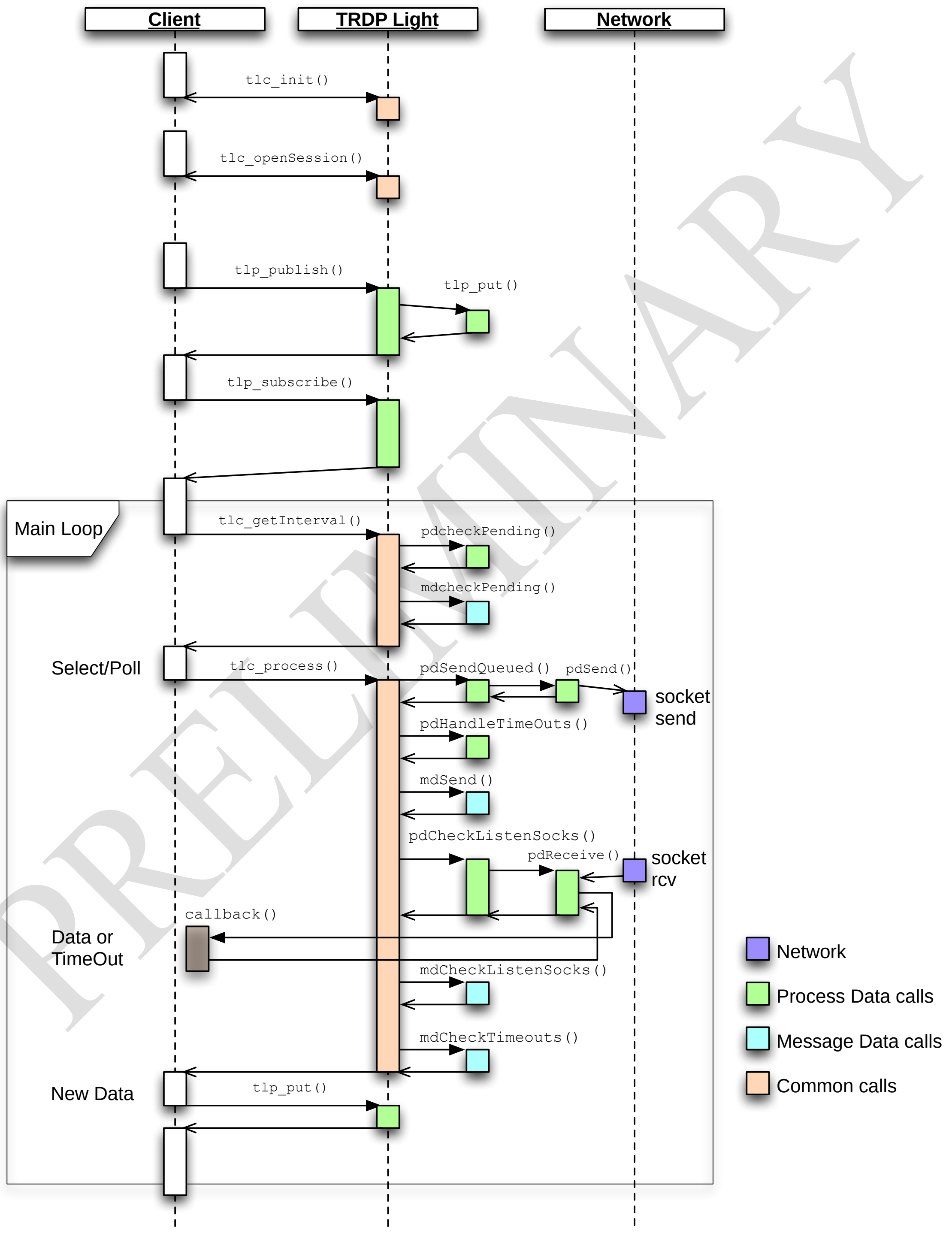

Figure 4: Sample Call Sequence for Subscription

If a callback is not provided by the user application (and non-blocking mode is set), it may collect new data by polling the `tlp_get()` function. `tlp_get()` will call `pdReceive()` repeatedly as long as data can be read from the PD socket. After testing for possible timeout of data, `tlp_get()` transfers the data to the user supplied buffer. Optionally, as with `tlp_put()`, a user supplied de-marshalling function will be used to copy the data while aligning and correcting the endianess to maintain the host system's data representation (see 2.7). If the timeout behaviour for this PD is set to `TRDP_TO_SET_TO_ZERO`, the user-supplied buffer is zeroed out (`memset()`).

Detailed description of the PD communication-related functions can be found in [2].

2.4. Process Data

For process data handling, two queues are maintained – a send and a receive queue (see also Figure 2). The send queue is filled or diminished by `tlp_publish()` resp. `tlp_unpublish()`, likewise the receive queue is filled or diminished by `tlp_subscribe()` and `tlp_unsubscribe()`.

2.4.1. Publish

On publishing, the caller supplies all necessary data to construct the PD header:

- comId
- current topology counter
- source and destination IP address
- interval
- send parameters
- pointer to initial data

`tlp_publish()` will request an adequate socket and will create and insert a new queue element into the send queue of the used session.
The needed data buffer is allocated through `vos_memAlloc()` and will be freed only when the PD is un-published.
Data is initially copied into the element data buffer by calling `tlp_put()`. Data can be updated at any time using `tlp_put()`. It will be sent only when `tlc_process()` is called, though! `tlp_put()` will copy the data into the internally managed data buffer by either doing a 1:1 memory copy or calling a supplied marshalling function to translate the local data representation into network format (see 2.7).
Any published packet can be sent using an interval (timeToGo, PUSH pattern) or on request (PULL pattern) or both. To prevent having discontinuous sequence counters in the case pushing of PD's is interrupted by PULL requests, two separate sequence counters must be maintained with each `PD_ELE_T`.

2.4.2. Subscribe

On subscribing, the functional flow is very similar to publishing; a new element is inserted into the receive queue instead and an eventual multicast group may be joined.
On `tlp_unsubscribe()` the subscription element will be released from the queue and the allocated memory (and the frame buffer) will be freed.

The data buffer for receiving PDs will always be allocated with the maximum data size (1436 + 36). This is different to message data handling, where the header will be peeked at first to find the announced data length, then a buffer of the correct size is allocated and the packet is read.

2.4.3. Traffic shaping

If traffic shaping is enabled, the starting time values of all PD packets to be sent must be evenly distributed over the largest used interval. Re-scheduling of the packet send times must take place each time new a PD is published or un-published.

The duration of PD packets on a 100MBit/s network ranges from 3µs to 150µs max. Because a cyclic thread scheduling below 5ms would put a too heavy load on the system, and PD packets cannot become larger than 1472 (+ UDP header) bytes, we need not account for differences in packet sizes.
Another factor is the differences in intervals for different packets: We should only change the starting times of the packets within half the interval time. Otherwise a late addition of packets could lead to timeouts of already queued packets.
The input requirements for traffic shaping are contradictory: The outgoing traffic should be evenly distributed over the time but the intervals of the packets to send may be of any value. For this reason only the starting time of the packets may be adjusted. Packet agglomeration at repeating times cannot be avoided and depends on the ratio and range of the interval times of all pending PDs!
Scheduling will be computed based on the smallest interval time in `tlc_getInterval()`.

2.5. Message Data

Message data handling is a compile time option, which is controlled by `#define MD_SUPPORT=1`.
All message data handling and most MD relatated data definitions are guarded by `#if MD_SUPPORT` and the corresponding `#endif`.

2.5.1. Overview

The diagram below shows the MD Communication Architecture with the most significant components.

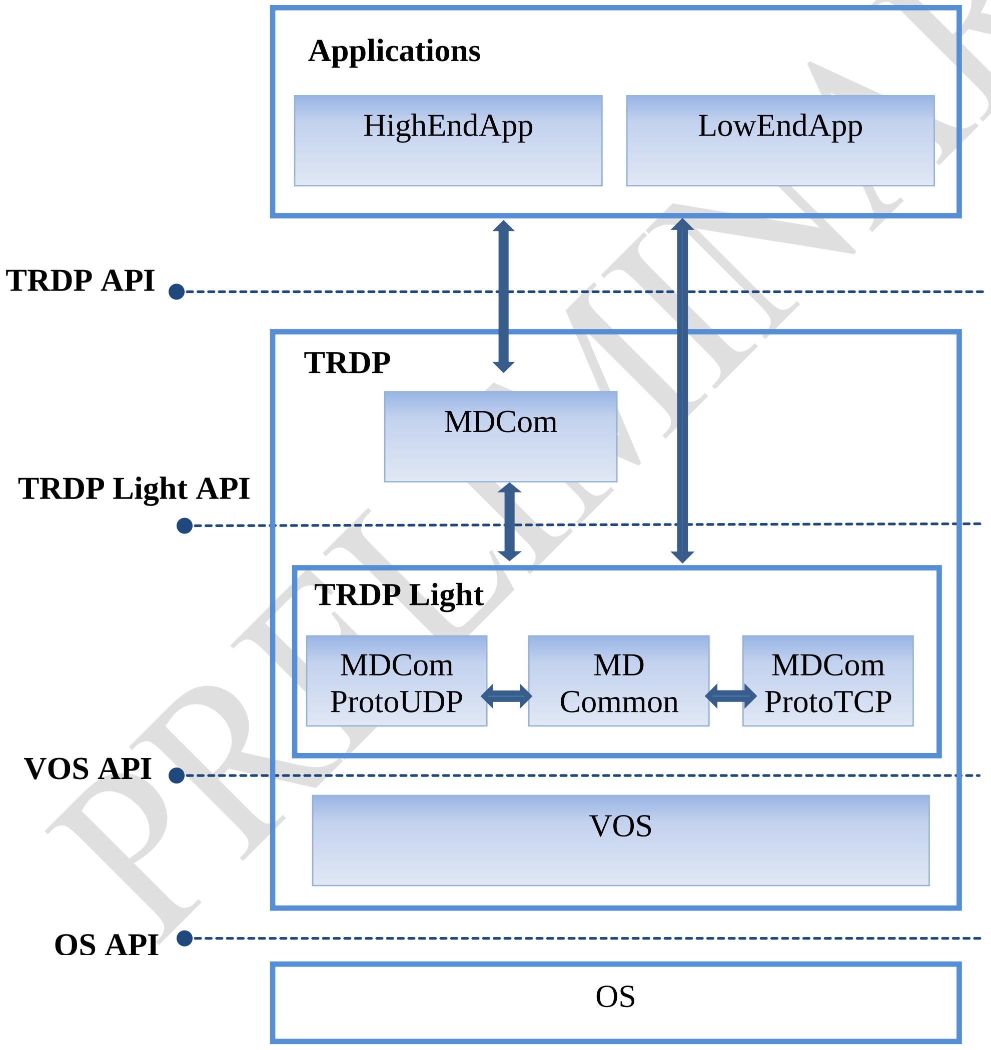

Figure 5: MD communication architecture

There are two main application types

- High End Application, use TRDP API to access MD advanced functions (queues, threading, etc.) provided by MDCom component
- Low End Application, use TRDP Light API to access basic MD function provided by TRDP Light through UDPMDCom and TCPMDCom components

MDCom behaves like Low End Application using TRDP Light API.

Regarding the TRDP Light API, three different components can be identified on it; UDP layer, TCP layer and the Common layer. The Message Data is based in the Common layer, where common functionalities and resources are provided equally to Udp and Tcp components.

All of them have access to the VOS (the OS and hardware abstraction layer) which provides a standard interface for the OS functions 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 Operative Systems are completely hided in the VOS.

MD component is accessible by application and MDCom using TRDP Light API, and uses VOS API to access to UDP/TCP specific functions.

Support MD over both UDP and TCP requires Figure 4

- MD handling layer independent from UDP/TCP
- VOS functions to manage UDP/TCP

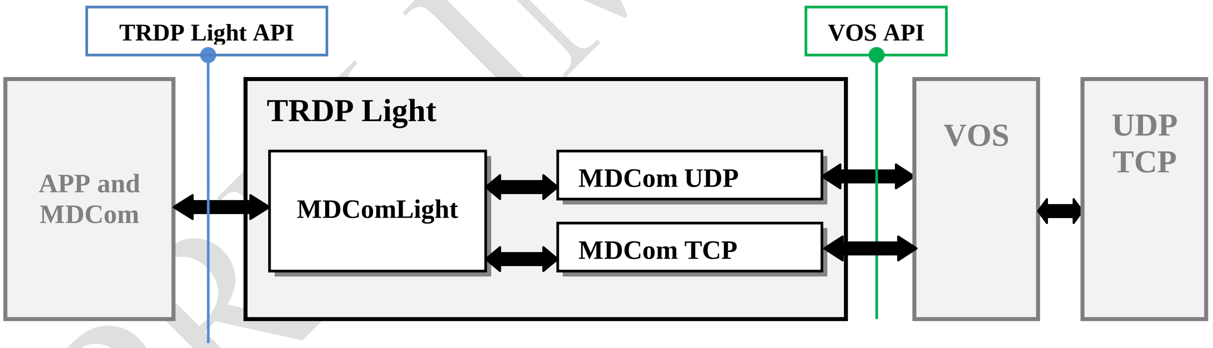

Figure 6: MD Light UDP/MD integration

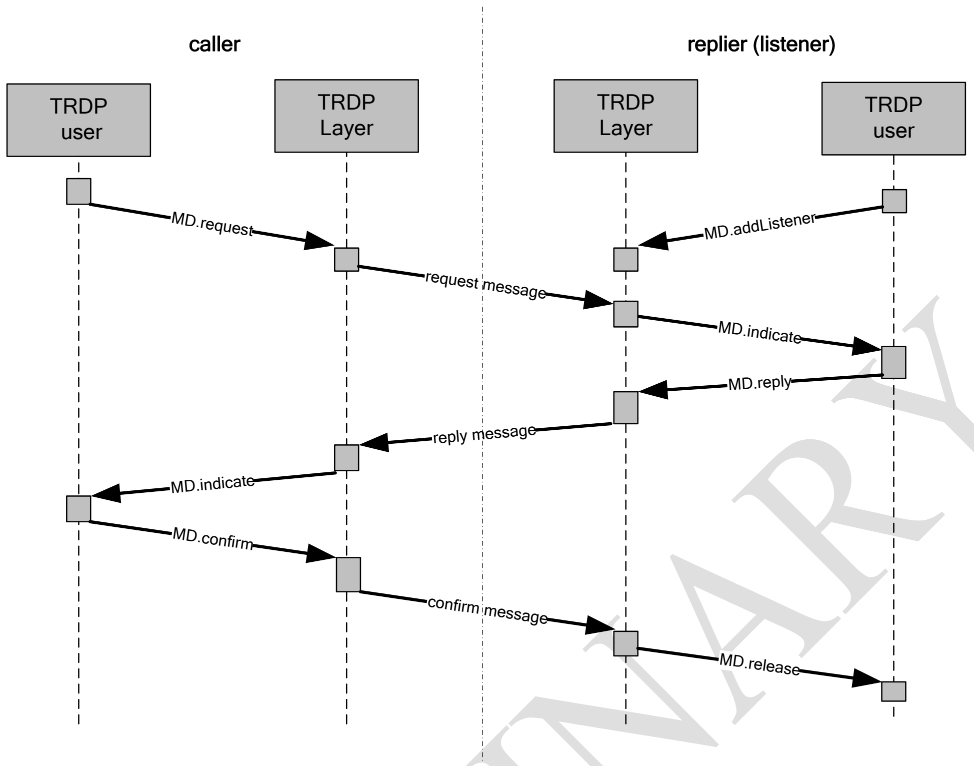

Figure 7: Application and protocol interactions

A generic replier application adds an MD listener for each MD it need to receive.
When an MD that match with registered parameters is received, TRDP stack will send it to replier application (indicate) that can (or not) reply; the reply can also contains an MD confirmation from the caller, that will be managed in the same way.

A generic caller application can send MD Notify and MD Request messages to a single (singlecast) or multiple (multicast) repliers. In case of reply one or more responses are expected, but all are singlecast. If the replier sends a reply with confirmation request, caller will send an MD Confirmation message, always singlecast.

Each TRDP MD event is notified to application layer executing a callback function, register by application itself during the initialization phase.

2.5.1.1. Communication Model

The objective of this diagram below is to explain the general idea of the MD communication. It is used one queue to store the messages that will be sent, and the receive function is called to get the messages. The *trdp_process()* is called periodically to manage these functionalities. The result of send/receive operations will be given via callback functionality to the application.

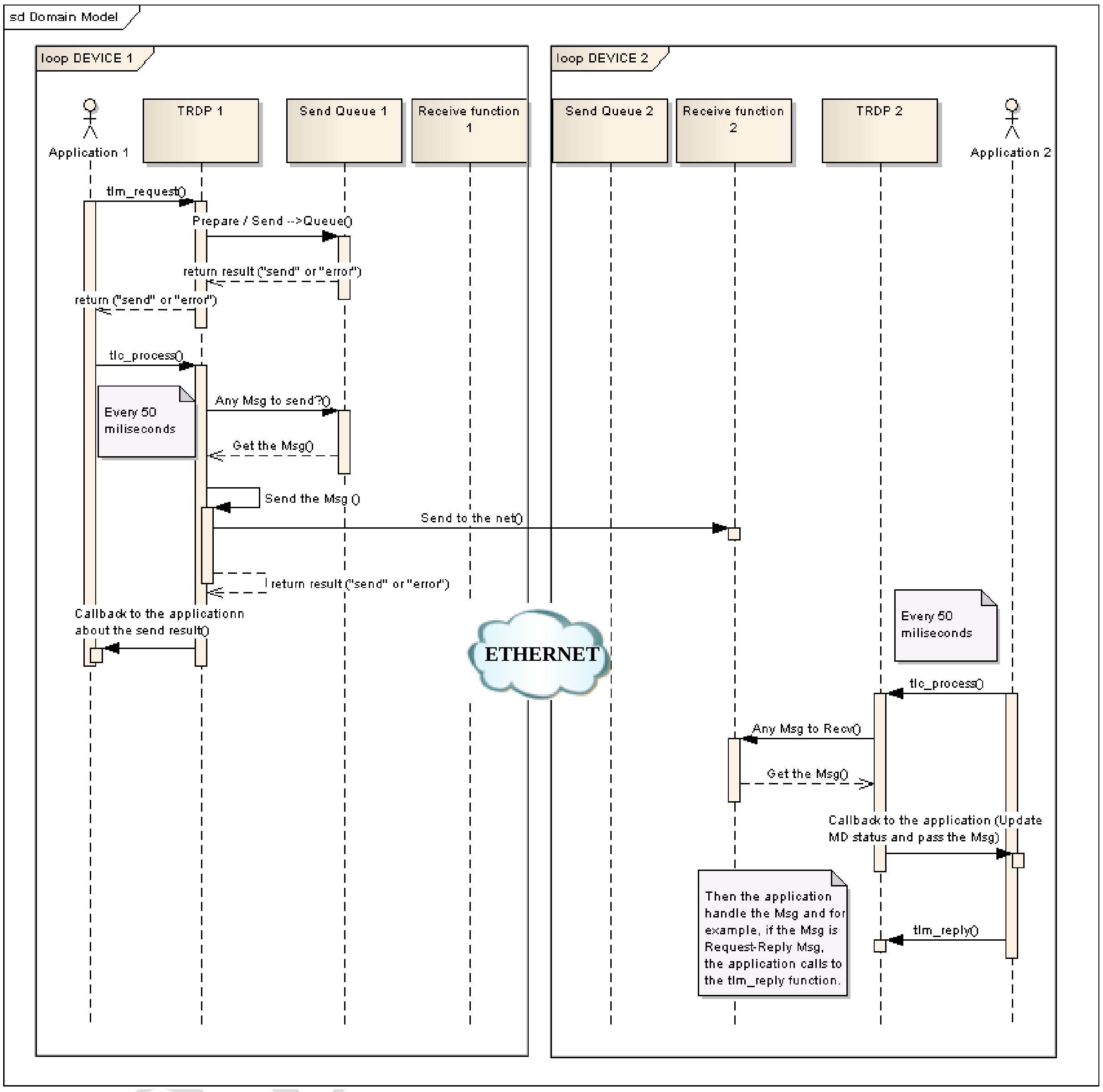
sd Domain Model
loop DEVICE 1
loop DEVICE 2
Application 1
TRDP 1
Send Queue 1
Receive function 1
Send Queue 2
Receive function 2
TRDP 2
Application 2
tlm_request()
Prepare / Send -->Queue()
return result ("send" or "error")
return ("send" or "error")
tlc_process()
Every 50 miliseconds
Any Msg to send?()
Get the Msg()
Send the Msg ()
Send to the net()
return result ("send" or "error")
Callback to the applicationn about the send result()
ETHERNET
Every 50 miliseconds
tlc_process()
Any Msg to Recv()
Get the Msg()
Callback to the application (Update MD status and pass the Msg)
Then the application handle the Msg and for example, if the Msg is Request-Reply Msg, the application calls to the tlm_reply function.
tlm_reply()

Figure 8: MD Communication example

In the above diagram is shown the message exchange between two devices. Initially, the Device 1 prepares the message and it stores the message in the Send Queue. After that, the tlc_process() is called periodically and this will send the messages that are in the SendQueue.

Regarding to the Device 2, when it calls the tlc_process(), it will check if there is any message received. In this case, the message will be checked and notified to the application. (The message reception functionality is inside the TRDP although in the diagram is shown separately).

The diagram shows the relation between the TRDP and the Application layer. The interaction between the application and TRDP functions is sampled graphically (e.g. the tlc_process() function calls to trdp_mdSend(), trdp_mdCheckListenSockets() and trdp_mdCheckTimeouts() in the TCP layer).

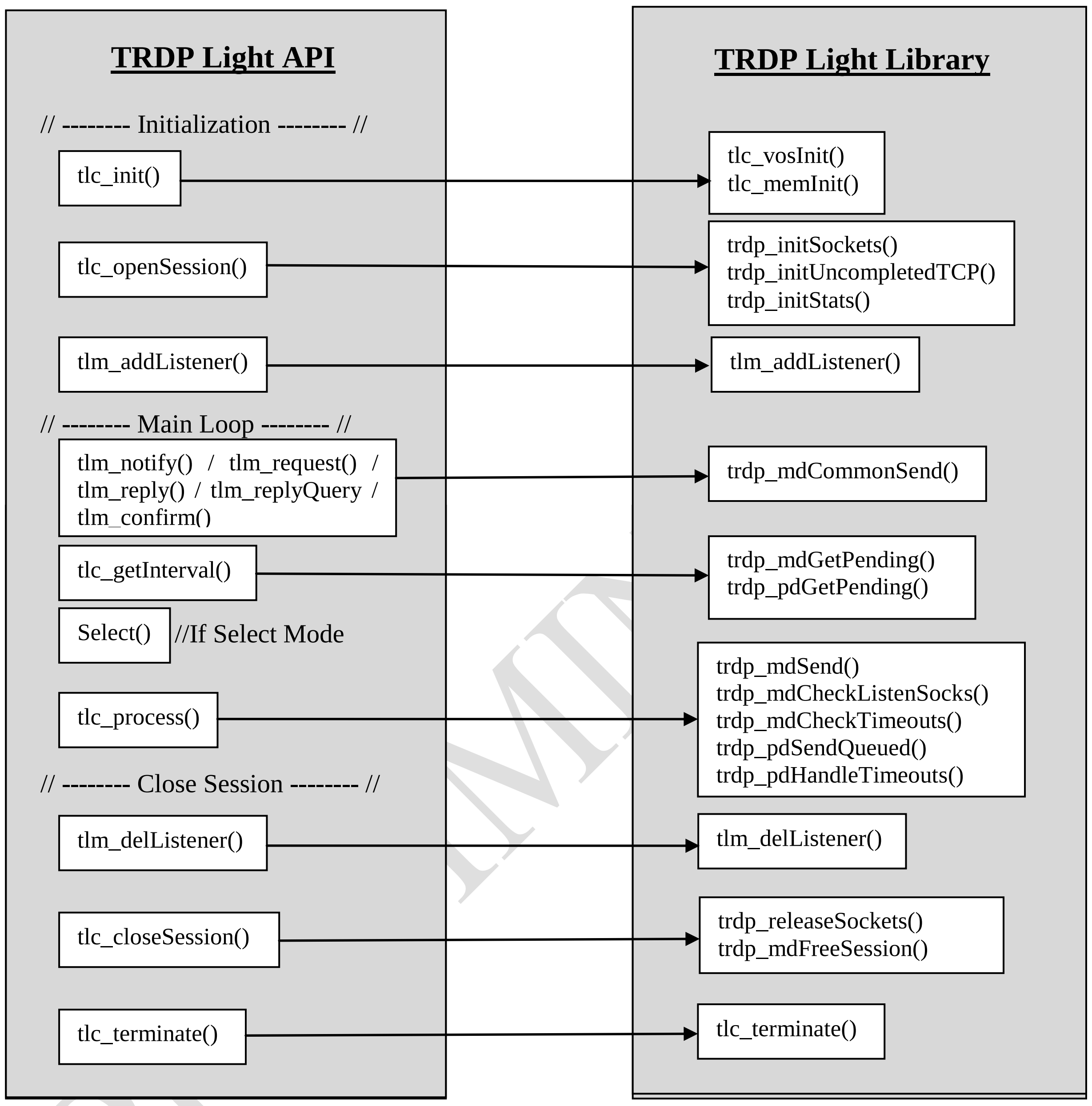

Figure 9: TRDP Light MD Call Hierarchy

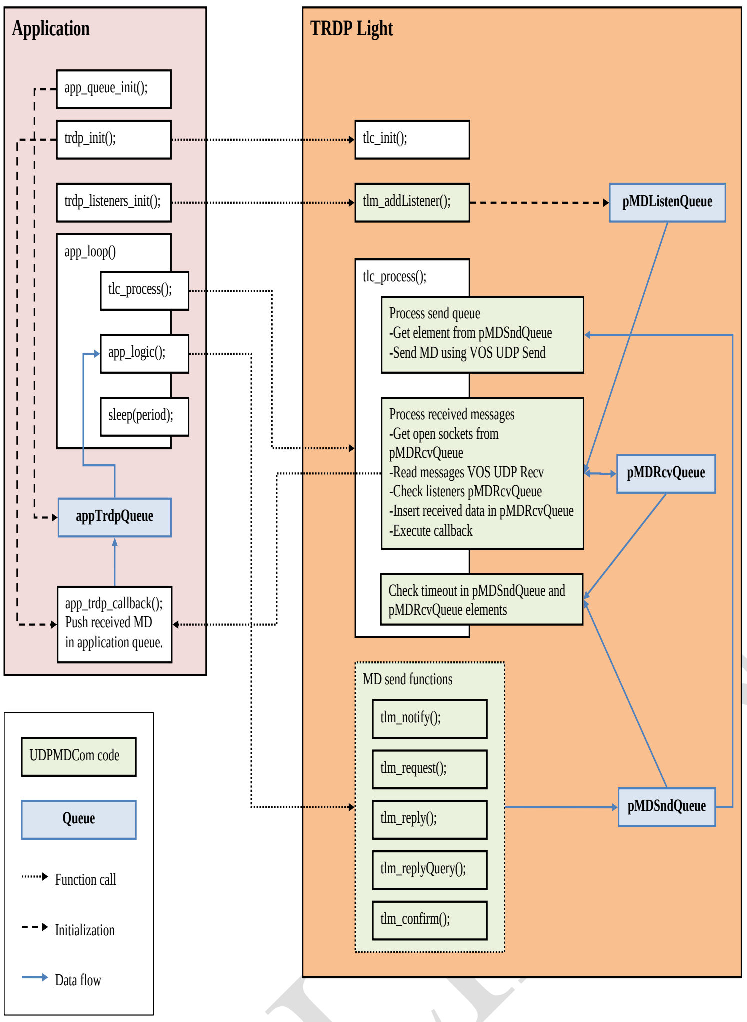
Application
app_queue_init();
trdp_init();
trdp_listeners_init();
app_loop()
tlc_process();
app_logic();
sleep(period);
appTrdpQueue
app_trdp_callback(); Push received MD in application queue.
TRDP Light
tlc_init();
tlm_addListener();
pMDListenQueue
tlc_process();
Process send queue -Get element from pMDSndQueue -Send MD using VOS UDP Send
Process received messages -Get open sockets from pMDRcvQueue -Read messages VOS UDP Recv -Check listeners pMDRcvQueue -Insert received data in pMDRcvQueue -Execute callback
pMDRcvQueue
Check timeout in pMDSndQueue and pMDRcvQueue elements
MD send functions
tlm_notify();
tlm_request();
tlm_reply();
tlm_replyQuery();
tlm_confirm();
pMDSndQueue
UDPMDCom code
Queue
Function call
Initialization
Data flow

Figure 10: TRDP Light MD Application Interaction

2.5.1.2. Message Data Queues

Three queues are used for Message Data (see also Figure 2):

- MdListenQueue: The queue contains all listeners of the TRDP session. All the data about the listeners added in the tlm_addListener() is stored here.
- MdSndQueue: The queue contains the caller sessions with the MD items that are waiting to be sent. The messages are added in the tlm_common_send() function, and are sent within the tlc_process(). After the item has been sent, the session is kept in the queue if the message requires one or more response message (updating its next state). Otherwise the item is deleted from the queue. The session will be closed after having received all requested replies and sending al necessary confirms. Per default the caller session will be closed after the reply timeout.

- MdRcvQueue: The queue contains the replier sessions. When a message is received, it is checked if a related listener is existing comparing the message comId/URI. If a related listener is found in the MdListenQueue and the maximum number of open sessions is not yet reached a replier session is opened adding a new replier session with the received MD element in the MDRcvQueue. Otherwise the message is rejected. The replier session will be removed from the queue if the reply is sent or - in case of a reply query – if a confirmation has been received. Per default the sessin will be removed after a given reply or confirm timeout.

2.5.1.3. Socket Handling

All the open sockets information is saved in the TRDP_SESSION_T iface array. Apart from the UDP MD and TCP MD sockets, the iface structure is also used by the UDP PD sockets. Each of the socket protocol type is specified in the iface.

2.5.1.4. Callback Mechanism

While for receiving PD callback and polling is possible, TRDP Light provides for receiving MD only the callback mechanism.

Even it is possible to handle received MD directly in callback function, this is not a best practice because the callback is blocking the further TRDP processing.

The preferred pattern, as shown in Figure 10, requires callback queues where all received MD iare put into an application queue, named *appTrdpQueue*. The message will be processed afterwards by application logic implemented in *app_logic()* function. This pattern allows to make TRDP more performant than immediately respond into callback, and it is ready to be implemented with multitask or multithread approach.

The TRDP functionalities used in the same way for both protocols (UDP/TCP) are described in the following chapters.
A generic application needs to create a message queue to receive TRDP MD via callback, needs to initialize TRDP stack itself with required parameters and needs finally to execute core packets processing calling in a periodic loop *tcl_process()* function and queue process function (see also chapter 1.1).

2.5.1.5. Parametrisation

TRDP_MD_CONFIG_T (tlc_openSession())

- MD callback function
- User context pointer for callback
- TRDP_SEND_PARAM_T - Default MD send parameter (QOS, TTL)
- TRDP_FLAGS_T - Default MD flags
- Default MD timeouts
    - replyTimeout: the reply timeout in µs value used when a request message has been sent. Default value 5s (TRDP_MD_DEFAULT_REPLY_TIMEOUT).

  - confirmTimeout: the confirm timeout in µs value used when a reply message has been sent. Default value 1s (TRDP_MD_DEFAULT_CONFIRM_TIMEOUT).

  - sendingTimeout: it's the period of time in which the message has to be sent. The timeout starts after the socket connection request is done to the destination device. Default value 5s (TRDP_MD_DEFAULT_SENDING_TIMEOUT).

  - connectTimeout: it defines the socket life. This means that if the socket is not used by any session the connectTimeout period, it will be closed. Default value 5s (TRDP_MD_DEFAULT_CONNECTION_TIMEOUT).

- MD UDP port
- MD TCP port
- Maximum number of open sessions to prevent DoS

TRDP_PROCESS_CONFIG_T (tlc_openSession()):

- TRDP_OPTION_T
  - TRDP_OPTION_NONE for non-blocking calls.
  - TRDP_OPTION_BLOCK for blocking calls

TRDP_SEND_PARAM_T (tlm_addListener(), tlm_request(), tlm_reply(),tlm_replyQuery(), tlm_confirm()):

- TTL and QoS can be set telegram specific

TRDP_FLAGS_T (tlm_addListener(), tlm_request(), tlm_reply(),tlm_replyQuery(), tlm_confirm()):

- TRDP_FLAGS_TCP - For TCP use in tlm_request() and tlm_addListener(). For reply and confirm the protocol defined in the request will be used.
- TRDP_FLAGS_CALLBACK - In TRDP Light for MD only callback can be used
- TRDP_FLAGS_MARSHALL – For receiving MD using callback mechanism there is no integrated marshalling possible

2.5.2. *Message Data Support of Common Functions*

2.5.2.1. tlc_init()

The application initializes and configures the TRDP stack calling the tlc_init() function. For that purpose, the function requires the memory configuration parameters (TRDP_MEM_CONFIG_T) and it returns a unique handle to be used in further calls to the stack. See chapter 2.2.

2.5.2.2. tlc_openSession()

A new session with the TRDP stack is opened and configured. Regarding MD the following parameters can be configured:

- callback function
- timeouts
- UDP/TCP ports

Giving a valid TRDP_MD_CONFIG_T structure were the message data base parameters are defined, the MD will be initialized.
The callback function needs to be configured to handle received MD. TRDP uses the callback function to notify the application about all the MD events; send/receive messages, timeouts, errors…. See chapter 2.2.

2.5.2.3. tlc_closeSession()

This function closes an existing TRDP session freeing all used sockets and removing all existing TRDP MD sessions calling *trdp_mdFreeSession()*.

2.5.2.4. tlc_getInterval()

To calculate time to the next call of *tlc_process()* this function is calling for MD handling *trdp_mdCheckPending()*.

2.5.2.5. tlc_process()

To handle MD telegrams, the application needs to call periodically the *tlc_process()* function that performs all TRDP stack related activities and callback execution (see also chapter 1.1).

Regarding MD, *tlc_process()* performs the following main tasks

- MD send handling calling *trdp_mdSend()*
- MD receive handling calling *trdp_mdCheckListenSocks()*
- MD timeout handling calling *trdp_mdCheckTimeouts()*

See chapter 1.1.

2.5.2.6. tlc_setTopoCount()

The execution of *tlc_setTopoCount()* sets the topo counter value inside TRDP stack, used to process incoming packets. See chapter 2.2.

2.5.3. Message Data API Functions

2.5.3.1. tlm_addListener()

Append the subscribe message information (comId's...) to the MD Receive Queue. In order to receive messages it is necessary to be subscribed to its comId/URI before, otherwise the message will be discarded.

Function *tlm_addListener()* mainly performs following steps

1. Create a new *MD_LIS_ELE_T* instance for the subscribed MD, containing all information needed (ComID, destination URI, flags, etc.)
2. Obtain a proper UDP (*trdp_requestSocket()*) or TCP socket (*trdp_mdGetTCPSocket()*) with the required parameters (QoS, TTL)
3. Eventually configure multicast related variables
4. Add the element to *pMDListenQueue* queue (*MD_LIS_ELE_T* pointer)

Each listener can be uniquely identify passing a unique value *pUserRef* to *tlm_addListener()*.

The list *pMDListenQueue* is used by receive process to filter MD that shall be received and ignore the others.

2.5.3.2. tlm_delListener()

This Function delets a peviously added listener and is freeing the related socket as well as the memory for the listener handling.

2.5.3.3. tlm_notify()

This function is sending a notification and is mapped directly to *trdp_mdCommonSend()*.

2.5.3.4. tlm_request()

This function is sending a request and is mapped directly to *trdp_mdCommonSend()*.

2.5.3.5. tlm_reply()

This function is sending a reply and is mapped directly to *trdp_mdCommonSend()*.

2.5.3.6. tlm_replyQuery()

This function is sending a reply query and is mapped directly to *trdp_mdCommonSend()*.

2.5.3.7. tlm_confirm()

This function is sending a confirmation and is mapped directly to *trdp_mdCommonSend()*.

2.5.3.8. tlm_error()

This function is sending an error message and is mapped directly to *trdp_mdCommonSend()*.

2.5.4. Message Data Support Functions

2.5.4.1. trdp_mdCommonSend()

This function is called to send any kind of message data. It gets a socket and prepares the MD message. Then, the message is put to the send queue.

The different MD send functions are implemented calling *trdp_mdCommonSend()* with proper parameters.

The reason is that MD send related functions have many common parts and it is more effective to implement them with a single function called with different parameters.

trdp_mdCommonSend() has the following tasks:

1. check if the MD to be sent has a sessionID related to an existing one in the receiving, *pMDRcvQueue*, or sent, *pMDSndQueue*, queues, and if yes take the necessary actions updating relater parameters
2. If there is not yet a TCP connection opened to the destination device a new socket is got and is saved in the iface array. Otherwise the resources that have been initialized before are used to send the message.
3. create a new *MD_ELE_T* for new MD
4. configure timeouts properly (depending on MD type)
5. request a socket with required parameters (QoS, TTL)
6. create a new sessionID, if new MD, or use the existing one found in *pMDRcvQueue* or *pMDSndQueue*
7. add element to *pMDSndQueue* queue (*MD_ELE_T*)

All *pMDSndQueue* elements are not sent immediately but they are stored into a TRDP internal send queue in order to be processed during *tlc_process()* function execution and sent using VOS function. This pattern is used to left to TRDP the full control of TCP/UDP telegrams transmission and reception hiding protocol complexity to user.

2.5.4.2. trdp_mdSend()

Function is called from *tlc_process()* and handles the following tasks:

- scan all *pMDSndQueue* elements, update message data states
- manage telegram count, state machine and timeouts, to execute callback in case of relevant events
- handle TCP send connections
- send the telegrams using *trdp_mdSendPacket()*
- mark processed elements that will be removed from *pMDSndQueue*
- eventually check *pMDRcvQueue* for pending replies

2.5.4.3. trdp_mdSendPacket()

Function is called from *trdp_mdSends()* and handles the following tasks:

- send all elements, using TCP or UDP depending on element flags

2.5.4.4. trdp_mdCheckListenSocks()

Function is called from *tlc_process()* and handles the following tasks:

- checks the sockets for received MD packets
- loops through the socket list and checks readiness
- processes a ready socket with *trdp_mdRecv*() function

2.5.4.5. trdp_mdRecv()

Function is called from *trdp_mdCheckListenSocks()* and handles the following tasks:

- handle TCP and UDP protocols, using *MD_ELE_T flags* parameter
- get telegrams from the socket using *trdp_mdRecvPacket()* function
- handle MD telegram and session finite state machine
- execute registered callback function to report events to application layer

2.5.4.6. trdp_mdRecvPacket()

Function is called from *trdp_mdRecv()* and handles the following tasks:

- get packets from operating system network queue using *vos_sockReceiveUDP()/vos_sockReceiveTCP()* function depending whether *pktFlags* parameter of MD_ELE_T element descriptor bit TRDP_FLAGS_TCP set or not
- calculate and allocate memory for the complete telegram
- put together packets received to a complete telegram
- check received telegram using trdp*_mdCheck()*
- update statistics

2.5.4.7. trdp_mdCheck()

Function is called from *trdp_mdRecvPacket()* and handles the following tasks:

- check header and data CRC
- check protocol version and type
- check telegram length
- check topo counter

2.5.4.8. trdp_mdCheckTimeouts()

Function is called from *tlc_process()* and handles the following tasks:

- checks *pMDRcvQueue* element for reply and confirmation timeouts
- manages request retries if one reply is expected but not received
- executes registered callback function to report events to application layer

2.5.4.9. trdp_mdCheckPending()

Function is called from *tlc_getInterval()* and looks for pending MD packets.

2.5.4.10. trdp_mdFreeSession()

This function frees the memory of a MD session.

2.5.4.11. trdp_mdCloseSessions()

This function checks for MD sessions to be closed, frees the related sockets and the memory of a MD session.

2.5.4.12. trdp_mdGetTCPSocket()

This function opens a listening socket for TCP.

2.6. Statistics

All functions effectively taking part in communication will update the counters kept in their session variable `TRDP_STATISTICS_T stats`. Functions referring to statistics are collected in the file `trdp_stats.c`.

All receiving and sending functions have access to the session element of their context and can increment/decrement the corresponding statistics values while operating. Nevertheless some statistics values will be computed only on demand – either by the application calling `tlc_getStatistics()` or a remote client using a ComID 31 control packet to “PULL” ComID 35 statistics data. This applies to the time fields (`timeStamp, upTime, statisticTime`), the current memory usage and the actual number of publishers and subriptions.
In `trdp_pdReceive()`, when receiving a message of type ‘`Pr`’ with ComID 31, the statistics packet is updated and sent directly to the requester by setting the private element flag `TRDP_REQ_2B_SENT` and calling `pdSendQueued()`.

2.7. Marshalling

The marshalling/un-marshalling functions are optional utilities independent from the rest of the TRDP stack, but are designed to be easily used as an integrated extension. When opening an application session, pointers to marshalling/un-marshalling functions can be provided to the TRDP stack, which in turn will use the provided functions to effectively copy data between the internal network data buffer and the applications data buffer on request only. This will work automatically when using `tlp_pdGet()` and `tlp_pdPut()`, but must be inserted separately when data is read using callbacks.

All marshalling-handling functions are defined in `tau_marshall.c` and are declared in the header file `tau_marshall.h`.

The marshalling functions need to know:

- the host compiler's alignment for all defined data types (VOS defined macro `ALIGNOF()`)
- the host compiler's size of all defined data types (operator `sizeof()`)
- the host computer's endianess – provided by a compile-time flag (`__BIG_ENDIAN__`)
- the structural definition of the packet to process (comID) – provided by the application

The `tau_initMarshall()` function must be called by the application before any other call to the marshalling subsystem. It will accept and maintain two application-supplied tables:

- a fixed number of `TRDP_COMID_DSID_MAP_T` elements provides the mapping between comIDs and datasetIDs
- a fixed number of `TRDP_DATASET_T` elements providing the structure of datasets.

Optionally an entry to allow cached access to datasets using a pointer to a dataset element structure will be maintained for every encountered dataset.
All of these tables will be accessed by the standard `qsort()/bsearch()` functions of the C-library, they must be mutable and must not be altered by the application after `tau_initMarshall()` processed them. Using the cached dataset pointer will save some calls to `bsearch()` on large datasets. Once a ComID /dataset match was found, the pointer to the dataset element definition is saved and allows direct access to the alignment information for both marshalling and de-marshalling.

Figure 11 shows an overview of the principle table structure:

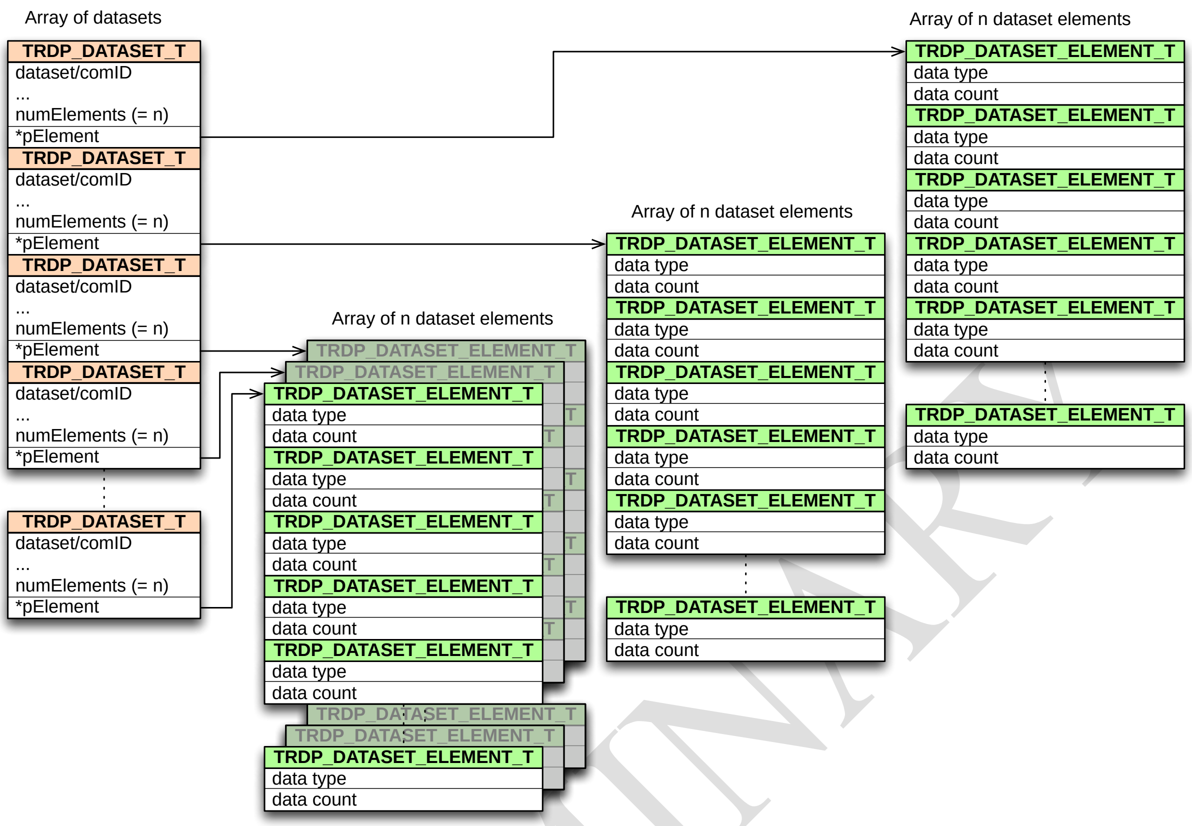

Figure 11: Structure of Dataset Tables

The relationship between ComID and datasetIDs is injective (with the exception of data-less ComIDs); datasets (structures) can include other datasets. Datasets contain one or more data-set elements, which in turn define one or more instances of one data type.

Data types (`TRDP_DATA_TYPE_T`) defined are:

```
- TRDP_BOOLEAN        = 1,  UINT8, 1 bit relevant
- TRDP_CHAR8          = 2,  char, can be used also as UTF8
- TRDP_UTF16          = 3,  Unicode UTF-16 character
- TRDP_INT8           = 4,  Signed integer, 8 bit
- TRDP_INT16          = 5,  Signed integer, 16 bit
- TRDP_INT32          = 6,  Signed integer, 32 bit
- TRDP_INT64          = 7,  Signed integer, 64 bit
- TRDP_UINT8          = 8,  Unsigned integer, 8 bit
- TRDP_UINT16         = 9,  Unsigned integer, 16 bit
- TRDP_UINT32         = 10, Unsigned integer, 32 bit
- TRDP_UINT64         = 11, Unsigned integer, 64 bit
- TRDP_REAL32         = 12, Floating point real, 32 bit
- TRDP_REAL64         = 13, Floating point real, 64 bit
- TRDP_TIMEDATE32     = 14, 32 bit UNIX time
- TRDP_TIMEDATE48     = 15, 48 bit TCN time (32 bit UNIX time
                       and 16 bit ticks)
- TRDP_TIMEDATE64     = 16, 32 bit UNIX time + 32 bit micro-seconds (==
                       struct timeval)
- TRDP_TYPE_MAX       = 30, Values greater are considered nested data-
                       sets
```

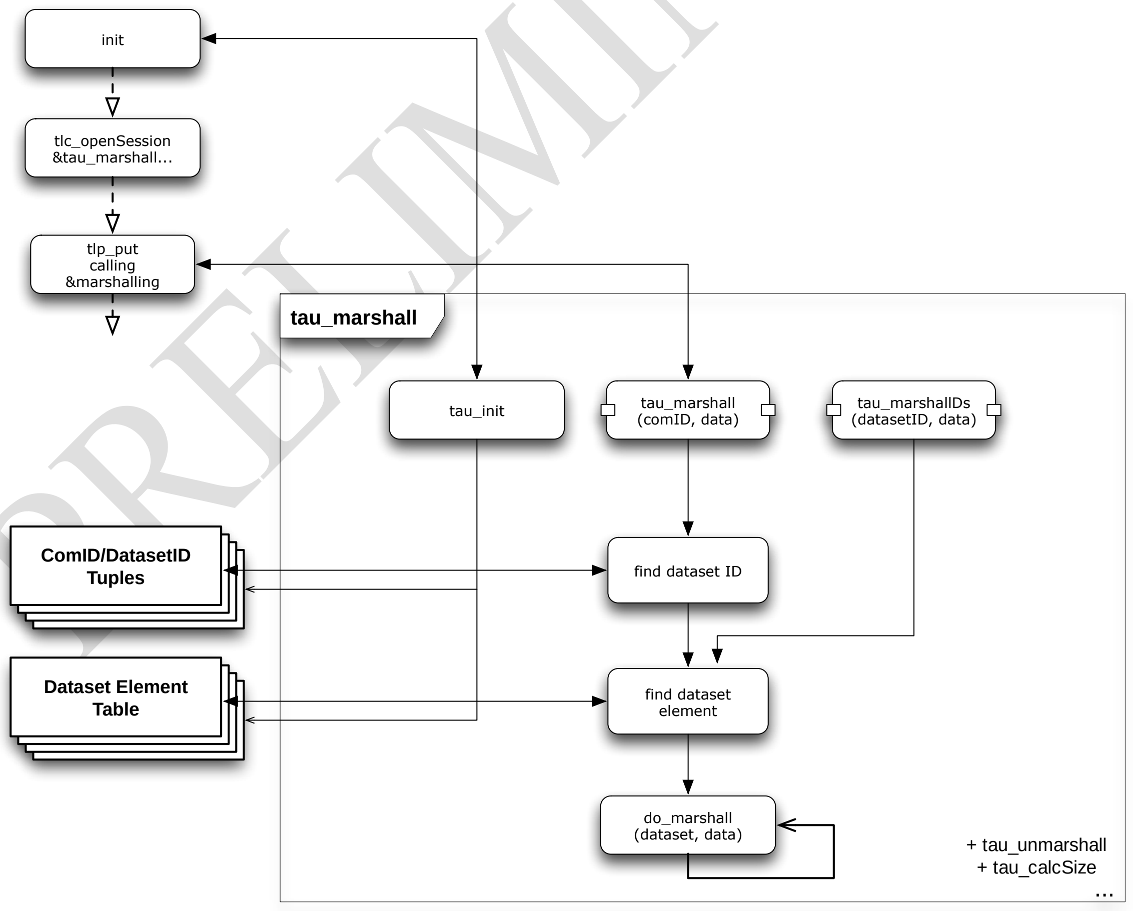
init
tlc_openSession &tau_marshall...
tlp_put calling &marshalling
tau_marshall
tau_init
tau_marshall (comID, data)
tau_marshallDs (datasetID, data)
ComID/DatasetID Tuples
find dataset ID
Dataset Element Table
find dataset element
do_marshall (dataset, data)
+ tau_unmarshall
+ tau_calcSize
...

Figure 12: Marshalling Overview

When receiving packets over the network, ComIDs define the structure of the complete payload and they are the first and only identifier to find the right description for decoding. If other structures are nested inside, they must be identified through dataset IDs. Functions called by TRDP communication functions, like `tlp_get()` and `tlp_put()`, will supply comIDs to the marshalling functions, while functions called from with the marshalling system can provide dataset IDs only. This results is a two-step approach, where the first entry needs to find the matching dataset ID for a given ComID, while in the second step, also using `bsearch()`, the corresponding dataset element description is found. See Figure 12 for an overview of the functions involved and the principal interrelationship with each other.
Un-marshalling and computing the marshalled size will work in the same way.

2.7.1. Marshalling function

`tau_marshall()` converts data structures in native representation, i.e. natural alignment of data types with host endianess, into a packed representation using network (big endian) byte order. It uses `bsearch()` to find the dataset corresponding to the supplied ComID and calls the actual marshalling function (`do_marshall()`), which can call itself recursively for nested datasets. The recursion depth is limited to 5. Figure 13 shows the simplified workflow:

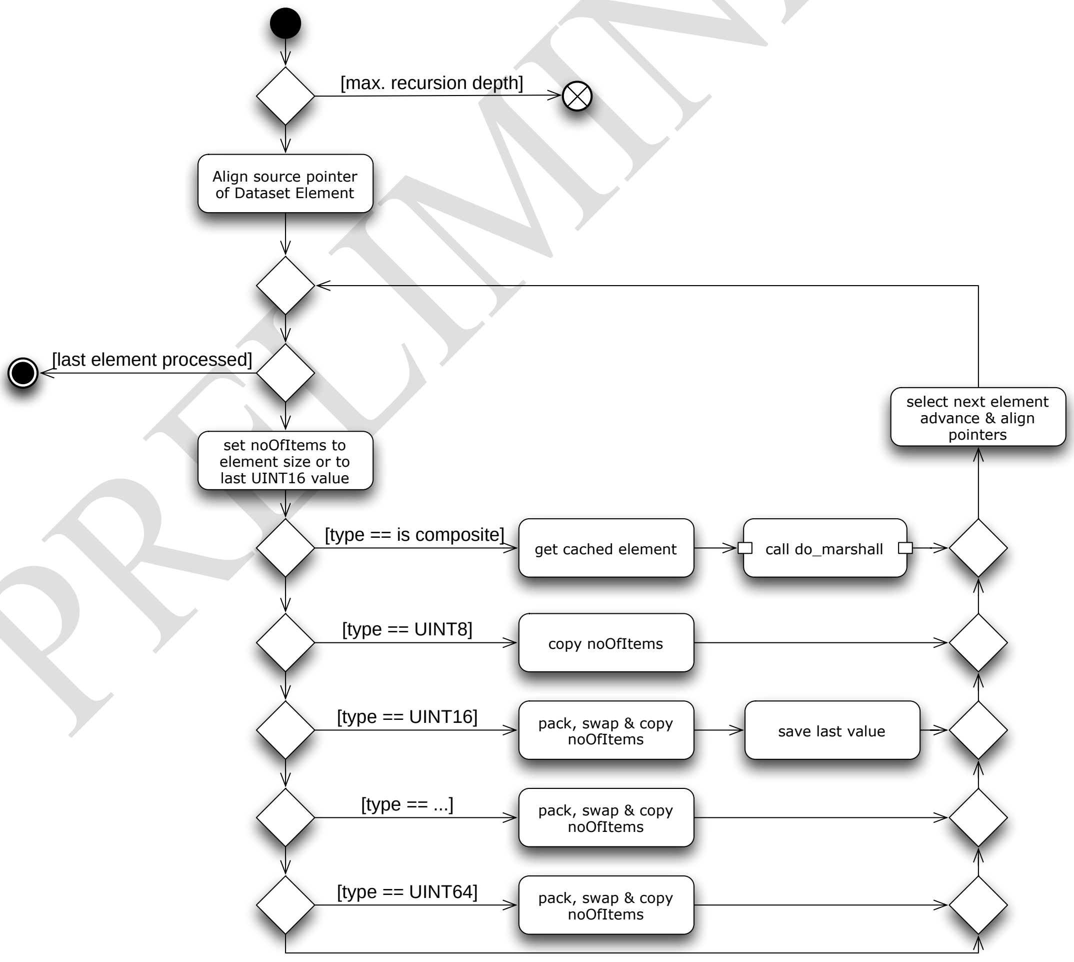
[max. recursion depth]
Align source pointer of Dataset Element
[last element processed]
select next element advance & align pointers
set noOfItems to element size or to last UINT16 value
[type == is composite]
get cached element
call do_marshall
[type == UINT8]
copy noOfItems
[type == UINT16]
pack, swap & copy noOfItems
save last value
[type == ...]
pack, swap & copy noOfItems
[type == UINT64]
pack, swap & copy noOfItems

Figure 13: Activity Diagram of Marshalling Function

After checking the recursion depth, we first align the source pointer to the natural alignment of the host's structure data type. Looping over the supplied dataset elements, for each entry we branch depending on the value of the type field. If it is a composite type (i.e. a dataset), we call ourselves recursively. For basic data types we do a packed copy and swapping according to the host compiler's settings.
There are two exceptions to this:

- Byte values (`TRDP_BOOLEAN`, `TRDP_CHAR8`, `TRDP_INT8`, `TRDP_UINT8`) need no special treatment, just copying
- The last `TRDP_UINT16` value is always kept and used as size parameter in case of a dynamic array definition.

2.7.2. Un-marshalling function

`tau_unmarshall()` converts data structures in packed representation in network (big endian) byte order into the native representation, i.e. natural alignment of data types with host endianess of the host system. It also uses `bsearch()` to find the dataset corresponding to the supplied ComID and calls the actual un-marshalling function (`do_unmarshall()`), which can call itself recursively for nested datasets. The recursion depth is limited to 5. The flow of control is the same as for marshalling, except that instead of accessing naturally aligned source data, the target pointer must be aligned and the data will be un-packed.

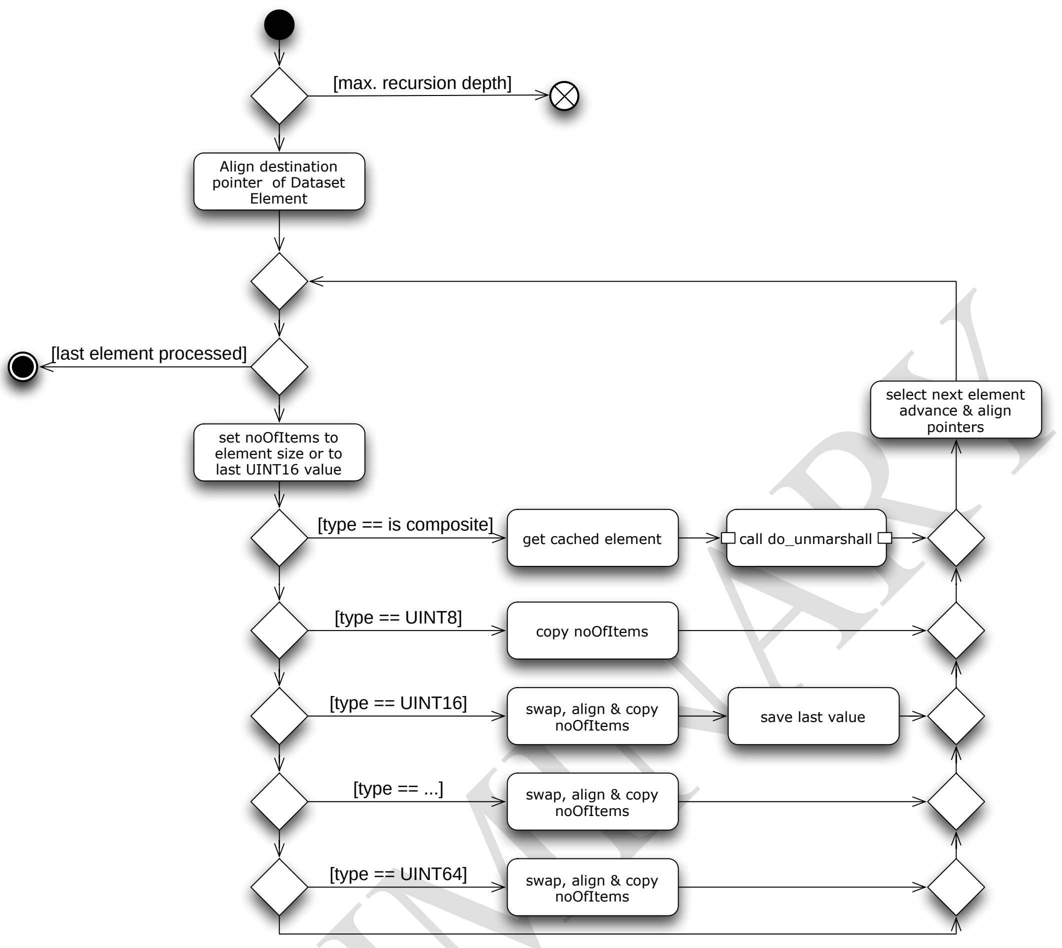
[max. recursion depth]
Align destination pointer of Dataset Element
[last element processed]
set noOfItems to element size or to last UINT16 value
select next element advance & align pointers
[type == is composite]
get cached element
call do_unmarshall
[type == UINT8]
copy noOfItems
[type == UINT16]
swap, align & copy noOfItems
save last value
[type == ...]
swap, align & copy noOfItems
[type == UINT64]
swap, align & copy noOfItems

Figure 14: Activity Diagram of Un-Marshalling Function

2.7.3. Compute Data Size

Calculating the needed marshalled data size is supported by `tau_calcDatasetSize()` and `tau_calcDatasetSizeByComId()`. The effort is nearly the same as real marshalling; just the copy phase is left out of the program flow. The same flow diagram applies.

2.7.4. Alignment

Critical for the correct (un-) marshalling function is the pointer/object alignment. There are some compilers which support the operator `alignof()`, but unfortunately the most common compiler family, gcc, has an incorrect implementation since V 3.2. The VOS supported `ALIGNOF()` macro circumvents this behaviour by being defined compiler dependent and using the `offsetof` operator for gcc. To verify correct implementation for different targets/compilers an alignment test function (`vos_initRuntimeConsts()`) is called during VOS initialization and will return an error (and logging output) if the `ALIGNOF` macro does not work as expected.

2.8. VOS – Virtual Operating System

All calls to system dependent services like memory, network, CPU (threading) are mapped to an abstraction layer. Target dependent implementations are located in directories named after their targets. For version 1.0 of the TRDP protocol stack, two major targets are supported:

- A Windows 32 Bit target using winsock2
- A generic POSIX target with branches for Linux, Darwin and QNX

Creating and selecting a specific target involves making changes to the `Makefile` and/or IDE project settings, see 2.10.1 and 2.10.3.

The VOS subsystem consists of:

- Type definitions
- Memory/Queue functions
- Utilities: crc32, debugging output, alignment
- Networking/Socket functions
- Threading/Time/Mutex/Semaphore functions
- Shared Memory functions

and can principally be used without the TRDP stack.

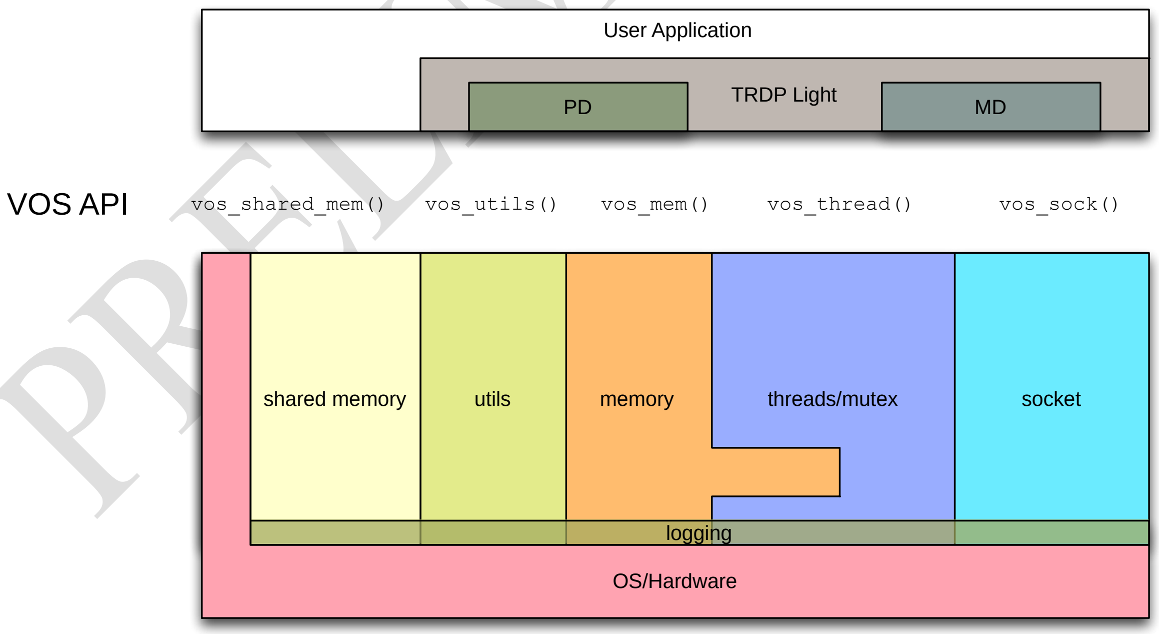

Figure 15: VOS Functional Layers

The TRDP Light stack does not use the shared memory subsystem itself – it is provided to support the implementation of traffic stores (e.g. the ladder system). Also semaphores and thread calls are not used, but all TRDP API functions are prepared for multi-threading by always taking a mutex before accessing non-local variables.

The indentation of the memory and the threads/mutex sections in Figure 15 shall indicate VOS memory functions using VOS mutexes and VOS mutexes using the VOS memory functions.

Applications, which want to use the VOS functions, will include the appropriate header files located in `vos/api`. The first call to the VOS system must be `vos_init()`, which will be called by `tlc_init()` if the TRDP stack is used.

The exact calling parameters and function prototypes are defined in the corresponding header files and in the Doxygen-generated reference manual [2].

2.8.1. Types

The VOS subsystem defines system independent data types, which are used throughout the API and inside the TRDP stack. Also some error codes are defined, which match the ones used in TRDP. Data types inside the particular implementations will use the target system's definitions. These data types are declared in the file `vos_types.h`.

2.8.2. Memory

One of the major target independent VOS features is the memory subsystem. It provides the management of fixed, pre-set block sizes of memory blocks from a user supplied area of memory.

This avoids wasting memory on small systems, where (because of MISRA rules) the use of the target system's dynamic memory allocation scheme is not allowed and fixed sized tables would be used instead.

The VOS memory subsystem circumvents this by implementing a pseudo-dynamic scheme, where the application (via `tlc_init()`) must pre-set:

- a memory area of defined size to be used
- a list of pre-allocated blocks to minimize fragmentation (optional)

Via `#defines`, the list of block sizes to manage is adjusted to TRDP's needs, i.e. to fit closely the size of the memory structures of PD publishers, subscribers, MD listeners and maximum packet size. This can be changed during compile time.

For regular systems (with Message Data support), the following block sizes are defined and can be allocated:

48, 72, 128, 180, 256, 512, 1024, 1480, 2048, 4096, 11520,
16384, 32768, 65536, 131072 Bytes

Default pre-allocations are 10 x 1480, 2 x 4096, 4 x 32768 Bytes

Small systems (`MD_SUPPORT` == 0) will provide:

32, 48, 128, 180, 256, 512, 1024, 1480, 2048, 4096, 11520,
16384, 32768, 65536, 131072 Bytes

Default pre-allocations are 8 x 1480, 1 x 11520 Bytes

All of these definitions and function prototypes are declared in the file `vos_mem.h`.

2.8.2.1. Memory Init

`vos_memInit()` initializes its global variables and creates a memory mutex, which controls access to the memory subsystem and allows the use of the memory subsystem with multithreading. The VOS-based mutex implementation might use itself the memory subsystem – a possible lock must be avoided by using static memory for the memory mutex. Three options can be selected by calling the function with different parameter values:

- if the supplied memory area pointer is NULL and the area size is zero, the subsystem is disabled and the `vos_memAlloc()` and `vos_memFree()` calls will be redirected to `malloc()` resp. `free()` of the host system. This mode is only recommended for use in debugging/testing environments. No pre-allocation will take place.
- If the supplied memory area pointer is NULL, but the area size is not zero, the memory area is allocated once using `malloc()` and then managed internally.
- Else the supplied memory area and size will be used.

Next the list of blocks to pre-allocate is traversed and pre-allocated if they would occupy less than half of the supplied memory area. Otherwise pre-allocation must not take place.

2.8.2.2. Memory Block Allocation

`vos_memAlloc()` will reserve a memory block for use by the caller until `vos_memFree()` is called for the same block.

Allocation is done by following these steps:

- Round up the demanded memory size to multiples of 32Bit.
- Find the closest size equal or greater than the requested size.
- Take the memory mutex.
- Try to allocate any returned memory blocks with the closest size equal or greater than the requested size.
- 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 the requested size.
- Try to allocate any returned memory blocks with a greater size.
- Release the mutex.
- Still no suitable block found? Return error.

The content of the returned memory block is set to all 0s (using `memset()`).

2.8.2.3. Memory Block De-Allocation

If the stack or user application does not need the memory area any more it releases it by calling `vos_memFree()`. After getting the memory mutex it will add this memory block to the list of available memory blocks, which can then be reused later.

If system memory allocation was used (`vos_memInit()` called with NULL-pointer, but parameter `size` > 0), the allocated memory will be free'd.

2.8.2.4. Statistics

To be able to track memory usage during development and deployment, the `vos_memCount()` function returns the number of used memory blocks, free memory blocks, total memory used, number of failed memory allocations and the list of memory block sizes.

2.8.2.5. Sort/Search

`vos_qsort()` and `vos_bsearch()` are wrappers for the standard library's qsort and bsearch functions and use the same parameters. If available, they can be mapped directly to the target supplied function.

2.8.2.6. String functions

`vos_strnicmp()` implements a case insensitive comparison and expects UTF8 encoded, zero terminated character strings. `vos_strncpy()` is the same as the standard `strncpy()` function and will usually be mapped to that function.

2.8.2.7. Queue/Mailbox

The functions `vos_queueCreate()`, `vos_queueSend()`, `vos_queueReceive()`, `vos_queueDestroy()` implement a mutex and semaphore protected queue, stack or priority queue, which can be used for inter-task communication by the application. The basic TRDP light stack does not use VOS supported queuing.

2.8.2.8. Utilities

The utility functions include the `vos_init()` function, which initializes the VOS subsystem and supplies a pointer to the user defined logging output function.

It will be the first call of an application after being instantiated. `trdp_init()`, as the main user of VOS will call `vos_init()` before calling `vos_memInit()` and passes the logging function pointer along. All error, debug and logging output will be routed via this function.

Four levels of logging output are defined:

Name	Value	Comment
VOS_LOG_ERROR	0	This is a critical error
VOS_LOG_WARNING	1	This is a warning
VOS_LOG_INFO	2	This is an informal message
VOS_LOG_DBG	3	This is a debug info

Every type of logging output is always generated – the application can decide, whether to print or ignore a specific level of message.

`vos_init()` will also call `vos_initRuntimeConsts()`, which will compute the alignment of various data types and compare these results with the result of the `ALIGNOF` macro. On any discrepancy, initialization will fail. This will aid in porting to new platforms and compilers. This code maybe omitted for small systems by defining `MD_SUPPORT=0`.

In addition, the thread and network subsystem's initialization functions will be called (`vos_threadInit()` and `vos_sockInit()`).

`vos_terminate()` will release all resources of the VOS subsystem by calling the corresponding …`Term()` functions and will release the allocated memory area (`vos_memDelete(NULL)`).

The crc32 function `vos_crc32()` implements the computation of the frame check sum according to IEEE802.3 and uses the pre-computed table from [1] .

2.8.3. Network

Socket and network related functions are declared in `vos_sock.h`.

2.8.3.1. Network Utilities

There are the common byte swapping functions `ntoh` (network to host) and `hton` (host to network) for 16 and 32 Bit unsigned values:

`vos_ntohs(),vos_ntohl(,)vos_htons(),vos_htonl()`

These functions must not swap endianess if the host data format is Big Endian (= network byte order).

The utility function `vos_dottedIP()` converts an IP address in the form of 10.0.0.1 into its number equivalent in host endianess (e.g. 0x0A000001). If available, the POSIX function `inet_addr()` may be used by the target implementation. `vos_ipDotted()` is the counter part function.

`vos_isMulticast()` returns `TRUE` if the supplied IP address is in the range 224.0.0.0 and 239.255.255.255.

`vos_getMacAddress()` returns the MAC address for a specified network interface; if no interface name is provided (pointer is `NULL`), the MAC address of the default interface will be returned. The realization of this function is highly target dependent: Even on the POSIX target, there are different implementations for Linux and BSD (QNX, Darwin).

On Linux, a socket must be opened and the MAC address can be retrieved with `ioctl`. On BSD, the list returned by `getifaddr()` must be traversed. On Windows, the NetBIOS's enumeration function is used.

`vos_getInterfaces()` is similar to `vos_getMacAddress()` in terms of target dependency – in fact it also returns the MAC addresses of all interfaces. Note that the application will provide the array and `vos_getInterfaces()` fills the supplied array up to value of `pAddrCnt`. On return, the number of interfaces found is returned in `pAddrCnt`.

`vos_select()` is a wrapper for the standard `select()` function. For small systems, this could be defined as no-op (only polling mode supported)

2.8.3.2. Socket Functions

Before calling any of the socket functions, the application will call `vos_sockInit()`. Any initialization of the socket subsystem can be done here. For POSIX, nothing special needs to done; the Windows implementation will prepare the WinSock2 library.

`vos_sockTerm()` will release any resources of the socket subsystem.

`vos_sockOpenUDP/TCP()` creates and returns a new socket for the particular protocol (UDP `SOCK_DGRAM` and TCP `SOCK_STREAM`) and sets the supplied socket options through `vos_sockSetOptions()`. To be able to send UDP messages of 64kB en bloc, the send and receive buffers might need to be enlarged. `vos_sockClose()` closes the corresponding socket.

`vos_sockSetOptions()` sets the socket options according to the supplied values. Not all options may be settable for a particular implementation – `reuseAddrPort` for instance might be mapped to either re-use port or re-use address. Blocking mode might not be available and will force the application to poll the sockets; QoS and multicast TTL might prohibit reliable train wide communication, if not settable. To allow destination address filtering, `IP_PKTINFO` or `IP_RECDSTADDR` must be enabled for `recv_message`. This might not be necessary for systems with only one interface.

`vos_sockSetMulticastIf()` allows to determine sending multicast through a particular interface. This overrides the system's default interface.

`vos_sockJoinMC()` and `vos_sockLeaveMC()` will subscribe/unsubscribe to a particular multicast group on a particular interface. Usually, each socket has a maximum of multicast groups it can join. The constant definition `VOS_MAX_MULTICAST_CNT` limits the TRDP stack's joins per socket. If more groups are needed, the TRDP stack will open another socket with the same options (if `reuseAddrPort` is set and possible).

`vos_sockSend()` for UDP and TCP will try to send the supplied data by taking interruptions by signals (on multi-tasking systems) and blocking into account. The TCP version must also check for a lost connection and return the appropriate error (`VOS_NOCONN_ERR`).

`vos_sockReceiveUDP()` uses the `recvmsg()` call with setting up the scatter/gather list. This is necessary (under POSIX and Windows) to get the destination IP address of the packet when multiple interfaces (or multi-homing) are present. Under the Windows XP operating system, the address of the `WSARecvMsg` function must be obtained using a `WSAIoctl` call first, because this function is not accessible through the `WinSock2` interface.

The `vos_sockReceiveUDP()` function must provide a way of peeking at the data to allow the caller to determine the necessary (big) buffer size from the header. The MD part of the TRDP stack uses this to avoid allocation of a buffer of the maximum packet size (64k).

`vos_sockBind()` binds the provided socket to a particular address and port. `vos_sockListen()` is a wrapper for the standard socket listening function to wait for remote clients to connect. `vos_sockAccept()` handles incoming connection requests for TCP. Intermittent signals and aborting connections must be caught and handled.

`vos_sockConnect()` tries to set up a TCP connection to the remote site. It will return no error if the connection was already established.

`vos_sockReceiveTCP()` reads data from the socket into the supplied buffer. It must return `VOS_BLOCK_ERR` if no data could be read in non-blocking mode and `VOS_NODATA_ERR` if the connection was reset while reading the socket.

See [2] for exact call parameters and valid return codes of all socket related functions.

2.8.4. CPU Resources

`vos_thread.h` declares (and partially defines) timer, threading and mutex/semaphore related functions.

2.8.4.1. Time

`vos_getTime()` returns the system's current time in the `VOS_TIME_T` format, which is equal to `struct timeval` on *ix target systems. It should return a monotonic clock value to prevent jitter or unsolicited transmissions or timeouts of PD data in case the clock setting changes during operation. `vos_getTimeStamp()` is used for logging output and should output the system's real time (not monotonic) to have a traceable listing of logging information. The string format is

`yyyymmdd-hhmmss.mmm` (**y**ear, **m**onth, **d**ay, **h**ours, **m**inutes, **s**econds, **m**illiseconds)

Several helper functions are declared to handle the computations with the `VOS_TIME_T` format and are used through out the TRDP stack. They all take pointers to `VOS_TIME_T` as parameters:

`vos_clearTime()` sets the `timeval` to `{0,0}`, `vos_addTime()` adds and `vos_subTime()` substracts the second to/from the first value; `vos_mulTime()` multiplies and `vos_divTime()` divides the first value by the second. They all save the result in the first value.

`vos_cmpTime()` returns `-1`, `0`, `+1` if the second value is smaller, equal resp. greater than the first value.

`vos_getUuid()` will return a 16 Byte long universal unique identifier, which is suitable to be used as session identifier. If the target system does not provide a system generated UUID, a substitute with at least the following features must be provided:

- Unique device ID (usually the MAC address of the default interface)
- Unique time stamp (using local monotonic time stamp)
- Incrementing usage count

The standardized format of the UUID is given in [3].

2.8.4.2. Threads

The thread handling subsystem is not used by the TRDP light stack, but is provided as a utility for higher layers (i.e. the application). All TRDP API session functions are mutex-protected and can therefor work in multithreaded environments. `VOS_MAX_THREAD_CNT` limits the maximum number of concurrent threads to 100.

In the current VOS implementations (POSIX & Windows XP), the `pthread` library is used to implement the thread-related functions. For simple single-threaded targets, these functions along with the mutex and semaphore functions could be defined as no-ops. They must not return error codes other than `VOS_NO_ERR`, though.

`vos_threadInit()` initializes the thread subsystem and should be called by the application before calling any other thread related function.

`vos_threadTerm()` releases all resources taken by the thread subsystem. It will not terminate any threads, though.

`vos_threadCreate()` is the major thread function and creates a new thread using the supplied parameters like

- policy (`VOS_THREAD_POLICY_FIFO`, `VOS_THREAD_POLICY_RR`, `VOS_THREAD_POLICY_OTHER`)
- priority (1…255 (highest, default 0)
- interval (for cyclic threads)
- stack size (default is 16kB)

and the run function plus argument pointer. Optionally (for debugging purpose) a name can be provided.

Note: Policies other than `VOS_THREAD_POLICY_OTHER` will probably require '`root`' permissions of the calling application on POSIX systems.

On successful invocation of the new thread, a handle will be returned to the caller. This is needed to identify the thread in further calls to `vos_threadTerminate()` or `vos_threadIsActive()`.

In the current POSIX and Windows implementations, the created threads are detached, meaning: There is no need to join the thread with the main thread before exiting the application.

`vos_threadDelay()` does not depend on threads as such, but will exempt the calling thread or task from execution for the provided time value. If the delay parameter is zero, the function should yield process time to other tasks, if any. On non-multitasking systems, the delay must be implemented using busy-waiting.

2.8.4.3. Mutex

All TRDP API functions, which access session or global variables, are protected by recursive mutexes. The memory space used will be allocated using `vos_memAlloc()` and a pointer to this area is used as a handle for all calls (un/locking, deleting). An exception to this is the creation of the local mutex used to protect the memory subsystem itself, where the caller (`vos_memInit()`) must allocate the memory for the handle. A valid handle should be identified by a magic number to avoid passing a false pointer to any of the mutex related functions.

`vos_mutexCreate()` creates a recursive mutex, which is unlocked on success. The memory for the mutex is allocated internally.

`vos_mutexLocalCreate()` does the same, but uses a pointer to an already allocated memory area. This function is not exposed to TRDP or the application.

`vos_mutexDelete()` destroys a mutex created by `vos_mutexCreate()`and releases the used memory.

`vos_mutexLocalDelete()` destroys a local memory mutex created using `vos_mutexLocalCreate()`. The caller must release the used memory.

`vos_mutexLock()` and `vos_mutexTryLock()` will lock the mutex . The first version will block until the lock is acquired. The second version returns `VOS_MUTEX_ERR` if the lock couldn't be acquired. They both return `VOS_NO_ERR` if successful.

`vos_mutexUnlock()` will unlock the mutex.

The POSIX and the Windows XP implementation use the `pthread`-supplied mutexes.

2.8.4.4. Semaphore

To support thread-safe mail boxing and message queues, four basic semaphore-handling routines are supported:

`vos_semaCreate()` creates a semaphore and returns a handle to `VOS_SEMA_T` as reference. The semaphore will optionally be taken, depending on the provided parameter.

`vos_semaDelete()` destroys the semaphore and releases associated memory.

`vos_semaTake()` tries to take the semaphore; depending on the value of the timeout parameter, if timeout is

- `0` (zero): do not wait, return immediately with error `VOS_SEMA_ERR` if semaphore could not be taken
- `0xFFFFFFFF`: wait forever for the semaphore, return `VOS_NO_ERR`
- `1...4294967294`: wait for the semaphore up to timeout value, return with error `VOS_SEMA_ERR` on timeout, on success `VOS_NO_ERR`

`vos_semaGive()` releases the semaphore.

The POSIX and the Windows XP implementation use the `sem_` prefixed semaphore functions (POSIX Realtime Extension).

2.9. Source Files

File	Content
tau_addr.h	TRDP address resolution declarations
tau_marshall.h	TRDP marshalling declarations
tau_tti.h	TRDP train topology info interface declarations
tau_xml.h	TRDP configuration interface declarations
trdp_if.c	TRDP Light interface functions
trdp_if_light.h	TRDP Light interface declarations (API)
trdp_mdcom.c	Functions for MD communication
trdp_mdcom.h	Functions for MD communication
trdp_pdcom.c	Functions for PD communication
trdp_pdcom.h	Functions for PD communication
trdp_private.h	Typedefs for TRDP communication
trdp_types.h	Typedefs for TRDP communication
trdp_utils.c	Common utilities for TRDP communication
trdp_utils.h	Common utilities for TRDP communication
api/vos_mem.h	Memory and queue functions declarations
api/vos_shared_mem.h	Shared memory functions declarations
api/vos_sock.h	Typedefs for OS abstraction
api/vos_thread.h	Multitasking functions declarations
api/vos_types.h	Declaration of VOS common data types
api/vos_utils.h	Logging, alignment functions declarations
common/vos_mem.c	Memory functions definitions
common/vos_utils.c	Logging, alignment functions
<target>/vos_private.h	Target dependend declarations and defines
<target>/vos_shared_mem.c	Target impl. of shared memory functions
<target>/vos_sock.c	Target implementation of network function
<target>/vos_thread.c	Target implementation of multitasking functions

2.10. Build System

2.10.1. Predefines

Several compile time options can be set to define a specific behaviour or feature set for the build targets. The following `#defines` are used and checked throughout the TRDP and VOS sources:

Name	Meaning	Defined/set in	Evaluated in

`MD_SUPPORT`	Defines inclusion of message data handling functions in the TRDP library.
	if `1` or not defined -> MD support is enabled
	if `0` and defined -> MD support is disabled, reduced resource usage
		command line option for `make`,
		`Makefile`,
		IDE build settings,
		`trdp_if_light.h`
			`trdp_utils.c`
			`trdp_if.c`
			`trdp_private.h`
			`vos_sock.h`
			`vos_mem.h`

`DEBUG`	Defines generation of symbolics (supressing STRIP command) and compiler optimizations	Environment variable / command line option for `make`	`Makefile`

`POSIX`	Selects includes and source path for POSIX compatible targets	`Makefile`,
		IDE build settings
			`Makefile`
			`vos_types.h`
			`vos_mem.c`
			`vos_sock.c`
			`vos_thread.c`
			`vos_shared_mem.c`

`WIN32`	Selects includes and source path for Windows XP compatible targets	IDE build settings	`trdp_proto.h`
			`trdp_types.h`
			`trdp_private.h`
			`vos_types.h`
			`vos_utils.h`
			`vos_mem.c`
			`vos_sock.h`
			`vos_sock.c`
			`vos_thread.c`
			`vos_shared_mem.c`

`VXWORKS`	Selects includes and source path for vxWorks compatible targets	`Makefile`	`Makefile`
			`vos_types.h`
			`vos_mem.c`
			`vos_sock.h`
			`vos_sock.c`
			`vos_thread.c`
			`vos_shared_mem.c`

Name	Meaning	Defined/set in	Evaluated in

`__linux`
`__APPLE__`
`__QNXNTO__`
	Sub-target defines for the POSIX platform. Differences in network and thread handling are taken into account.	Compiler/system includes	`Makefile`
			`vos_types.h`
			`vos_private.h`
			`vos_sock.c`
			`vos_thread.c`

`__BIG_ENDIAN__`
`__ARMEB__`
`__AARCH64EB__`
`__MIPSEB__`
	Defines the host endianess; if any of these is 1, the host is considered to be big endian	Compiler/system includes	`vos_utils.c`
			`tau_marshall.c`

2.10.2. Command Line

On the top level of the project, a `Makefile` allows the generation of the TRDP library and several test and example utilities. To support multiple targets, a configuration folder with target & compiler specific includes exists. To select a certain target or build setting, a configuration file `config.mk` inside the configuration folder will be created/overwritten by the make command

```
Make <TARGET>_config
```

where `<TARGET>` needs to be replaced by a real target name. Currently at least a native Linux X86 target exists (`POSIX_X86_config`).

With the command

```
Make help
```

the possible targets and options are listed. To add more targets and tool-chains, a file in the `config` directory must be created, where several path and configuration variables for each target can be set.

To generate the documentation from the source files, 'Doxygen' version 1.5 or higher must be installed on the host system. Depending on your installation it might be necessary to adjust a path in the file `Doxyfile` on the same level. To be able to generate the PDF version of the reference manual, "`Ghostscript`" (for the `eps2pdf` utility) is also needed.

For special configurations (e.g. Ladder) different make files can be used and should exist in the project's root directory.

2.10.3. IDE

2.10.3.1. Xcode

The folder `Xcode` contains the project file bundle `trdp.xcodeproj` for the Mac OS X Xcode IDE version 3.2 with several predefined targets and demo projects.

2.10.3.2. Visual C

`Win32TRDP_VS2010.sln` in the folder `VisualC` is a project file for Microsoft's Visual C IDE 2010. Several predefined targets support building of a TRDP static and dynamic library and several test and demo applications.

2.10.4. Lint

Under `test/lint,` MS-DOS batch files for use with Gimpel Software's PC-lint are hosted, `Lint_posix.bat` and `Lint_win.bat`. Before invoking these files, the path to `lint-nt.exe` has to be set according to the installed location of PC-lint.

To be able to 'lint' the POSIX version of the VOS as well, a set of Linux header files is included in the compressed file `posix.zip`, which must be unpacked before usage. The standard headers originate from an Ubuntu 8.04 system (`i486-linux-gnu` gcc V 4.2.4).

The files `std_win.lnt` resp. `std_posix.lnt` contain source path definitions, exemptions and rules.

Output from the lint runs is saved into file `_LINT_WIN.TMP` resp. `_LINT_POS.TMP`