Transport Protocol ISO15765-2

Technical Reference

Single/Multiple Connection

Version 3.14.00

Authors	Oliver Garnatz, Andreas Pick, Peter Herrmann, Thomas Dedler
Status	Released

Document Information

History

Author	Date	Version	Remarks
Rein	1999-06-22	1.0	File created
Baeuerle	1999-11-02	1.42	Description of connection specific timing parameters added
Ebner	2000-07-17	1.51	Single connection version removed; documents only contains multiple connection extensions
Garnatz	2000-09-19	2.03	Adaptation to new MultiConnection TP
Garnatz	2001-02-09	2.07	Added new functionality
Garnatz	2001-05-11	2.10	Update new Generation Tool versions
Garnatz	2001-09-14	2.17	General improvement; Update to version 2.17 of tpmc.c module
Garnatz	2002-01.24	2.27	SingleConnection version is added; Protocol-Overview is added
Garnatz	2002-06-18	2.33	Added restrictions for data consistency
Pick / Garnatz	2002-10-16	2.36	Update: CAN Driver in polling mode Added: Fast transmission of ConsecutiveFrames Update: Usage of TransmitCF parameter
Garnatz	2002-11-29	2.37	General rework
Garnatz	2003-01-16	2.39	Update: TpTransmit/CopyToCan/ApplTpCheckTA
Garnatz	2004-01-13	2.44	Update: ApplTpCopyToCAN
Pick	2004-03-01	2.52	Update: Mixed 29-bit ID addressing TpRxGetCanBuffer TpRxSetBufferOverrun TpRxGetAddressExtension TpTxSetAddressExtension
Pick	2004-05-14	2.60	Multiple ECUs example Restriction on TpTxStateTask/TpRxStateTask Tx/Rx message buffer

			consistency clarification Return value of ApplTpPreCopyCheck Mixed 11-bit ID addressing TpTransmit() return values Added TpCanChannelInit() Added TpRxSetTransmitID() Changed TpRxSetBufferOverrun Changed ApplTpTxCopyToCAN Changes in chapter ‘How to serve Different Connections (only dynamic channels)’.
Pick	2004-12-01	2.68	Added description for GENy configuration tool (ESCAN00008734). Update of API description (ESCAN00008314). Feature list added (ESCAN00008315). Prototype parameter corrected (ESCAN00009965)
Pick	2005-04-07	2.72.00	Added description for multiple addressing systems. C++ access to TPMC.
Pick	2005-07-14	2.73.00	Added description for GENy configuration
Herrmann	2005-07-19	2.73.00	Added new API functions: TpRxSetWaitCorrectSN, TpTxSetStrictFlowControlCheck
Herrmann	2005-08-11	2.73.00	Added new API functions: TpRxSetTimeoutConfirmation, TpTxSetTimeoutConfirmation, TpRxSetTimeoutCF, TpTxSetTimeoutCF
Garnatz	2006-01-13	2.80.00	Added deviation to ISO 15765-2
Herrmann	2006-02-08	2.82.00	ISO 15765-2 deviations elaborated
Herrmann	2006-03-03	2.86.00	Cleanup (ESCAN15514)
Herrmann	2006-03-23	2.86.00	ISO 15765-2 deviations elaborated
Herrmann	2006-04-11	2.87.00	General rework after review
Herrmann	2006-07-03	2.89.00	Added WaitFrame handling.

Herrmann	2007-02-01	2.90.00	Added OEM feature TP_ENABLE_STRICT_DL_CHECK
Herrmann	2007-02-23	2.91.00	Added feature TP_DISABLE_MF_RECEPTION
Herrmann	2007-03-14	2.92.00	Added ApplFuncTpPrecopy callback description and reduced TpRxResetChannel API usage to indication point in time or after.
Herrmann	2007-09-20	2.93.00	Completed Multiple ECU description (see chapter 7.3.1). Added TpRxGet-AddressingFormat / AssignedDestination description.
VERSION 3.xx			
Herrmann	2007-10-15	3.00.00	Added description for new TpClass "Dispatched<AddressingType>"
Herrmann	2007-11-20	3.01.00	Cosmetics / Syntax
Herrmann	2008-01-14	3.02.00	New API: TpTxGetTargetAddress
Herrmann	2008-02-12	3.03.00	Minor corrections within API descriptions (ApplTpTxErrorIndication, TpRxGetCanBuffer)
Herrmann	2008-04-17, 2008-07-17	3.04.00	Added description for TP_ENBLE_DYN_CHANNEL_TIMING. Added description for the usage of extended identifiers for normal addressing as well at configuration time as also dynamically at runtime (TP_USE_EXT_IDS_FOR_NORMAL).
Herrmann	2008-12-10	3.05.00	Added description for GenMsgDelay attribute in chapter 3.4.1
Herrmann	2009-01-25	3.07.00	Adapted version number to ALM package number (3.06.00 skipped)
Herrmann	2009-11-25	3.08.00	Added description for reception and transmission without flow control frames for dyn. (TpRxWithoutFC, TpTxWithoutFC) and static

			(TpTxFlowControl, TpRxFlowControl) Tp classes.
Herrmann	2010-01-12	3.09.00	Enhanced description for DLC checks on the Rx side (see 2.4.2.5). Added API functions for 29-Bit ext. Id dynamic handling.
Heil	2010-11-08	3.10.00	Added more flexibility for DLC checks on the Rx side (see 2.4.2.5)
Herrmann	2011-01-19	3.11.00	Moved TP_MEMORY_MODEL_DATA from user config file to GENy
Herrmann	2011-04-05	3.12	ESCAN00051019: Added new (customer specific) pre-compile switches: TP_ENABLE_IGNORE_FC_RES_STMIN, TP_ENABLE_IGNORE_FC_OVFL (see 3.2.3).
Herrmann Dedler	2011-07-11 2011-09-21	3.13	ESCAN00051019: Added support for the dynamic setting of 29-bit CAN-IDs (see 4.2.2.31, 4.2.2.32, 4.2.3.29, 4.2.3.30). Added new pre-compile switch: TP_USE_UNEXPECTED_FC_CANCELATION (see 3.2.3).
Dedler	2012-04-10	3.13.01	Description of TpRxGetCanBuffer modified according to ESCAN00057225
Dedler	2013-04-30	3.14.00	Description for non-standard flow control handling updated (3.2.3)

Reference Documents

No.	Title
[1]	/ISO/TF2/: ISO FDIS 15765-2; Road vehicles — Diagnostics on CAN — Part 2: Network layer services; Date 2004-07-16
[2]	/OSEK-COM/: OSEK/VDX Communication Version 2.1, revision 1 17th June 1998
[3]	/CANDrv/: Manual for CAN Driver in used version
[4]	ISO15765-2: ISO TC 22/SC 3; ISO 15765-2:2003(E); Road vehicles — Diagnostics on controller area network (CAN) — Part 2: Part 2: Network layer services

Caution

We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector´s release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

Contents

Illustrations

Tables

1 Introduction

1.1 Relation between general component and shipped version capability

We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector's release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

This implementation and this user manual are based on the documents, listed above.

It is important to know the documents above-mentioned for a better understanding and the use of this manual.

/OSEK-COM/ defines different kinds of transmissions. One of them is the USDT (Unacknowledged Unsegmented Data Transfer). It is standardized together with ISO/TC22/SC3 „Diagnostics on CAN“. The result of this standardization is the ISO Spezification15765-2.

The presented Vector-Implementation is based on the harmonized specification between OSEK-COM and ISO. The implementation is suitable for diagnostic purposes (KWP2000) as well as for „long“ messages in „normal“ use.

Task of the transport layer is to transmit messages, which might be longer than a CAN-message. If messages do not fit into a CAN-message, they will be segmented by the transport protocol to be transmitted.

Today the ISO/TF2-transport protocol is mainly used for diagnosis applications in motor vehicle. Most of all KWP2000 is used as a diagnosis protocol.

The introduction is followed by a brief overview of the architecture in the third chapter. On one side the most important points of the specification can be seen there (see also /ISO/TF2 and /OSEK-COM/) and on the other side this explains the main ideas of this implementation.

The fourth chapter presents how to set up the transport protocol in the “Generation Tool”.

The fifth chapter contains a description of user interfaces of implementation.

Transmission attributes and callback functions are presented in a table in chapter 5.

Rules to integrate CANbedded modules in customer projects are content of chapter 5, 6.

Chapter 7 is introducing a more advanced usage of the TP.

The last chapter contains an example for the user.

1.2 Name Conventions

The prefix of a function name determines the module to which it belongs.

Prefix	Remark
ApplTp...	These functions must be defined within the customer's application and were called by the Transport Layer module. The modules, which use functions of the Transport Layer, are always called application in this manual.
ApplTpRx …	Hook-Functions which belong to the "reception part" of the TP.
ApplTpTx …	Hook-Functions which belong to the "transmission part" of the TP.
Can...	Functions belong to the CAN-Driver.
TpRx...	Functions belong to the "reception part" of the Transport Layer.
TpTx…	Functions belong to the "transmission part" of the Transport Layer.

Table 1-1 Name Conventions

1.3 Abbreviations

List of abbreviations in use:

AE	Address Extension
AI	Address Information
AK	Acknowledge
AR	Acknowledge Request
ASDT	Acknowledged Segmented Data Transfer
BS	Block Size
CF	Consecutive Frame
CTS	Clear To Send
DL	Data Length
FC	Flow Control
FF	First Frame
FS	Flow Status Control
ID	Identifier
SF	Single Frame
SN	Sequence Number
ST	Separation Time
TA	Target Address
TP	Transport Protocol
TPCI	Transport Protocol Control Information
USDT	Unacknowledged Segmented Data Transfer
UUDT	Unacknowledged Unsegmented Data Transfer
WT	Wait
XDL	extended Data Length

Table 1-2 Abbreviations

1.4 Channel vs. Connection

A (transport) **channel** is the physical part of the communication link, containing the reception-/transmission mechanism. It can be understood as an instance of TPMC in an object oriented meaning. Each channel can handle one connection at one point in time.

A **connection** describes a logical communication link between two ECU's. In the communication matrix it is a fixed assignment between these ECU's to interchange data (e.g. the diagnostic request and response message between the Tester and an ECU).
A connection includes all necessary communication parameters for the used addressing mode (e.g. CAN-channel, CAN-IDs, Source-and Target Addresses, Base-Addresses, etc).

1.5 TP classes

1.5.1 SingleTP classes

In a Single TP class only one connection is possible, which is using the only available TpChannel.

1.5.2 Static MultiTP classes

While using Static TP classes every connection is fixed assigned to a TpChannel.

1.5.3 Dynamic MultiTP classes

The idea of dynamic TP classes is to use the circumstances that not all connections are used at the same time. Therefore a connection is necessary allocating a TpChannel at run-time.

1.5.4 Dispatched MultiTP classes

The “Dispatched” MultiTP class was introduced to disburden the application from the dispatching job.

Using the “Dynamic MultiTP” classes, which support only one single set of callback functions for all connections together, the dispatching of the actual destination has to be performed by the application.

Using the “Dispatched MultiTP” classes all of the dispatching work is done within the TPMC.

“Dispatched MultiTP” is located between static and dynamic TP classes.

Transmission

The new allocated TpChannel has included blank communication parameters only, except for the connection-handle (`tpChannel = TpTxGetFreeChannel(connection)`). To establish the connection it is necessary to assign the connection parameters to the TpChannel. The TpChannel is always used to refer to the connection (like a handle). Every callback- or API-function has the tpChannel as a parameter.

Reception

If a Single- or FirstFrames is received the Transport Protocol is searching internally for a free TpChannel. If a free TpChannel is found a data buffer will be requested by calling `ApplTpRxGetBuffer()` from the application. Within this function the application has also to decide to which connection the received TP frame belongs.

1.6 SingleConnection vs. MultipleConnection

The TPMC component has two different operation modes: a SingleConnection and a MultipleConnection mode. The MultipleConnection mode has the capability to handle different transmissions and receptions at the same time like ECU 1 in figure 1. If SingleConnection mode is used only one transmission and one reception (one full-duplex connection) can be performed at the same time (ECU 3 and ECU 4). A typical usage for the SingleConnection mode is a diagnostic connection.

The SingleConnection mode needs lower resources (ROM and runtime), than the MultipleConnection mode.

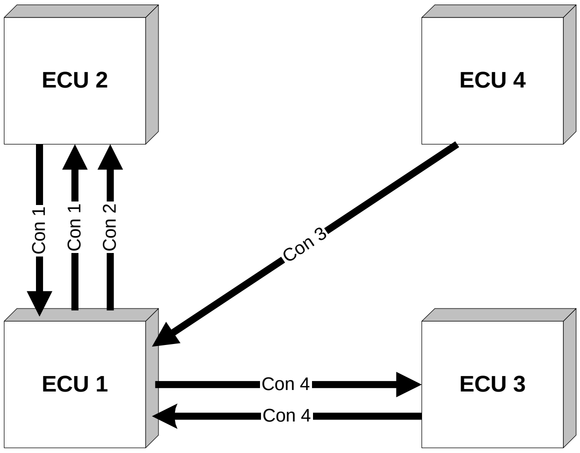


Figure 1-1 SingleConnection vs. MultipleConnection

1.7 Features

The main focus while the development of the Transport Layer is an easy to handle and flexible application interface.

- Therefore the buffer handling should be done by the application itself. This is more flexible than a static buffer handling internally by the Transport Layer.
- Each accepted order to the TP will be acknowledged only once – positive or negative.
- Full-duplex capability - every reception is independent from every transmission and the other way round.
- The static MultipleConnection TP supports connection-specific callback functions.
- SingleConnection mode with lower resource demands.
- Full ISO compliance
- Non-ISO extensions like ‘zero-padding’; ‘connections without FlowControls’
- Multiple addressing mode support (Normal- and Extended Addressing at the same time in the same ECU)

1.7.1 Feature List

Not any version of TPMC offers any mentioned feature

Feature	Availability	Short Description	Default (on / off)
General Features			
Normal Addressing	Liz	11bit CAN ID Addressing, CAN ID identifies TP message	-
Extended Addressing	Liz	11bit CAN ID Addressing, Source Address in CAN ID and Target Address in first data byte	-
Normal Fixed Addressing	Liz	29bit CAN ID Addressing, Source and Target Address in CAN ID	-
Mixed 11bit CAN ID Addressing	Liz	11bit CAN ID Addressing, CAN ID identifies TP message, first data byte used for AddressExtension → Gateway	-
Mixed 29bit CAN ID Addressing	Liz	29bit CAN ID Addressing, Source and Target Address in CAN ID, first data byte used for AddressExtension → Gateway	-
Multiple Addressing	Liz	Combination of former mentioned addressing types	-
Static channel assignment		Assignment between channel and connection is fixed at compile time. Advantage in opposite to dynamic assignment is better efficiency (code + runtime)	
Dynamic channel assignment		Flexible pool of channels, which can be assigned to connections at runtime. If no channel is free the request is rejected. Nr of channels can be <= connections. (Time division multiplexing)	
C++ access to TPMC		C++ applications can access TPMC. Header declared as extern C.	
Additional OBD reception capability		Additional receive path to handle OBD requests at any time, independent to allocated channel resources.	
Receiving Features			
Extended API STmin		Enables functions to set and get the STmin value for a TpChannel.	Off
Extended API BlockSize		Enables functions to set and get the BS value for a TpChannel.	Off
Precopy check / Check TA function		Forwards CAN Driver Precopy callback from TPMC to application. Used for special purposes.	Off
Check Target Address former called: Application Precopy	Mixed29, Normal Fixed	Forwards CAN Driver Precopy callback from TPMC to application. Parameter TargetAddress is evaluated by application. Return value 0xFF rejects reception.	Off
Channel specific timing	Static TPMC	Assigns individual timing values to each channel.	Off

Custom Rx Memcopy		TP calls ApplTpRxCopyFromCAN callback function to enable the application copying the CAN frame data itself.	Off
Rx Channel without FC	Multi TPMC	No FC used in transport protocol communication.	Off
Fast Precopy	Extended , Mixed29, Normal Fixed	Target Address is not evaluated when receiving a TP frame.	Off
Transmission of FC in ISR		The FC is sending in CAN RX IRQ forced from FF and last CF out of a block.	On
Fix Rx DLC Check		Check compares actual DLC with expected frame length (CAN: 8).	Off
Variable Rx DLC Check		Check compares actual DLC with minimum expected frame length. Check is TPMC frame type depending.	On
Functional FC Wait		Non ISO feature: A functional FC with flow status wait is supported to reload with functional addressing the timeout timer awaiting physical FC.	Off
Strict length check		If variable Rx Dlc is enabled then the minimum byte count is checked. If more bytes than announced in the PCI byte (SF and last CF) are received then the frame is accepted nevertheless. When the strict length check feature is enabled (#define TP_ENABLE_STRICT_DL_CHECK) then all frames which do not exactly match the PCI-DL value are ignored.	Off
Suppress Multi - frame reception		For some applications, which use only Single Frames on the Rx side, the reception of Multi Frames can be disabled by setting the TP_DISABLE_MF_RECEPTION switch via a user configuration file. The benefit is the smaller resource consumption. The remaining Single Frame reception is unaffected.	Off
Transmission Features			
Use STmin of FC		The STmin value is used from the FC. See also TxTransmitCF.	Off
Analyze first FC only		Only first FC values are analyzed to set STmin and BlockSize.	Off
Custom TX Memcopy		TP calls ApplTpTxCopyToCAN callback function to enable the application copying the TX data to the CAN frame.	Off
TX Channel without FC	Multi TPMC	Transmission without waiting for a FC. In dynamic TP classes this feature can be activated for each channel.	Off
Fast TX Transmission		Enables the application to send TP frames in cycle time faster than TpTxTask() cycle time.	Off
Transmission of FC in ISR		Directly response with FC in IRQ context of received FF or CF.	On
Variable DLC		The DLC is adapted for SF, FC and last CF as indicated by addressing type and data amount.	Off

Ignore FC content		FC is required for proceeding but standard values are used instead of received ones.	Off
TX Handle Changeable		The used CAN Driver handle can be changed while runtime – has to be used with special care	Off
No STmin after FC		No STmin time is kept after receiving a FC before sending next CF.	Off
TX min timer		If the database attribute 'GenMsgDelayTime' has a value unequal to zero, the TP observes this minimum time between two transmissions.	Off
Special Features			
Gateway API		Extended API to support Gateway requirements (TP message routing)	
Multiple ECU NR		Source- and TargetAddress can be modified while runtime	
Multiple ECU		Optimized support for physical multiple ECU configurations.	
Multiple Base Address	Extended	More than one Base Address can be used	
BufferOverrun Indication		If the request size exceeds the buffer size, this feature can be used to receive the request anyway, without copy the CF data.	off
Queue in ISR	Dynamic TP-classes	The next queued element (if available) will be transmitted within TX-ISR.	on
ISO Compliancy		Distinguish between early ISO spec drafts and newer ones concerning STmin interpretation, DataLength = 0 behavior and CF sequence error treatment.	on
Frame Padding		SF and last CF frame are padded out with a pattern given in the generation tool.	oem, off
Priority inversion protect		Prevents TPMC to interrupt a multi frame transmission/reception when transmission and reception events are in wrong order processed (RX event with higher priority than Tx event). See also "2.5.1".	on
Runtime checks		Runtime condition checks	off
Strict message flow check		Illegal FlowControl frames will suspend a running transmission – with same addressing information	on
Diag Functional channel	CANDesc (basic)	Capability to handle functional diagnostic requests within TPMC (only for Vector Diag components e.g.: CANDesc)	on

Table 1-3 Feature List

2 Architecture Overview

This chapter describes the basic functionality of the Transport Protocol and its main ideas applying to the Vector implementation of the Transport Protocol. Particular functions of the Transport Protocol modules, as well as its configuration are described in later chapters.

The main idea of the Vector implementation is to provide an interface, which is easy in operation and adequate for most applications. The implementation is quite efficient regarding ROM and RAM as well as run-time requirements.

2.1 Requirements

This chapter shows basic requirements of the implementation of the Transport Protocol.

2.1.1 Protocol-Overview

The Task of the transport protocol is to transmit messages, which are generally longer than a CAN message. If a message is very short, it is transmitted unsegmented within TP.

2.1.1.1 Construction of unsegmented messages

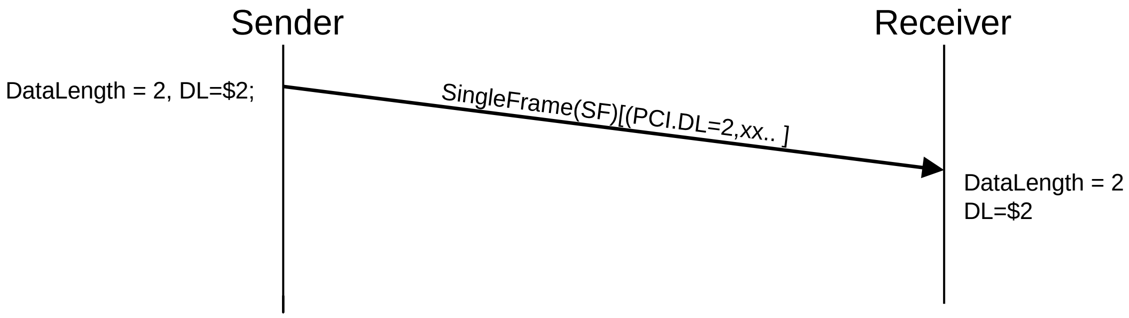


Figure 2-1 Example of unsegmented message

Unsegmented messages are transmitted by a SingleFrame message. SingleFrame messages can have a length of 7 data bytes at a maximum (normal addressing s.b.) respectively 6 data bytes (extended addressing, s.b.). There is no Flow-Control (s.b.).

2.1.1.2 Construction of segmented messages

Messages, which do not fit into a SingleFrame are sent by a sequence of single CAN frames. The receiver is informed of the length of the whole message in the FirstFrame by the sender. ISO/TF2 defines here a maximum length of 4095 bytes for user data. The receiver answers with a FlowControl. The receiver gives the BlockSize and the SeparationTime ST_{min} to the sender in this FlowControl. The BlockSize controls the number of ConsecutiveFrames, which might be sent by the sender before waiting for the receivers' FlowControl (status). The minimum value of the SeparationTime ST_{min} describes the minimum sending distance between two ConsecutiveFrames, which can be processed by the receiver. The sender transmits the maximum BlockSize ConsecutiveFrames after the reception of the FlowControl. The receiver does not answer it with a FlowControl, if all data has been transmitted.

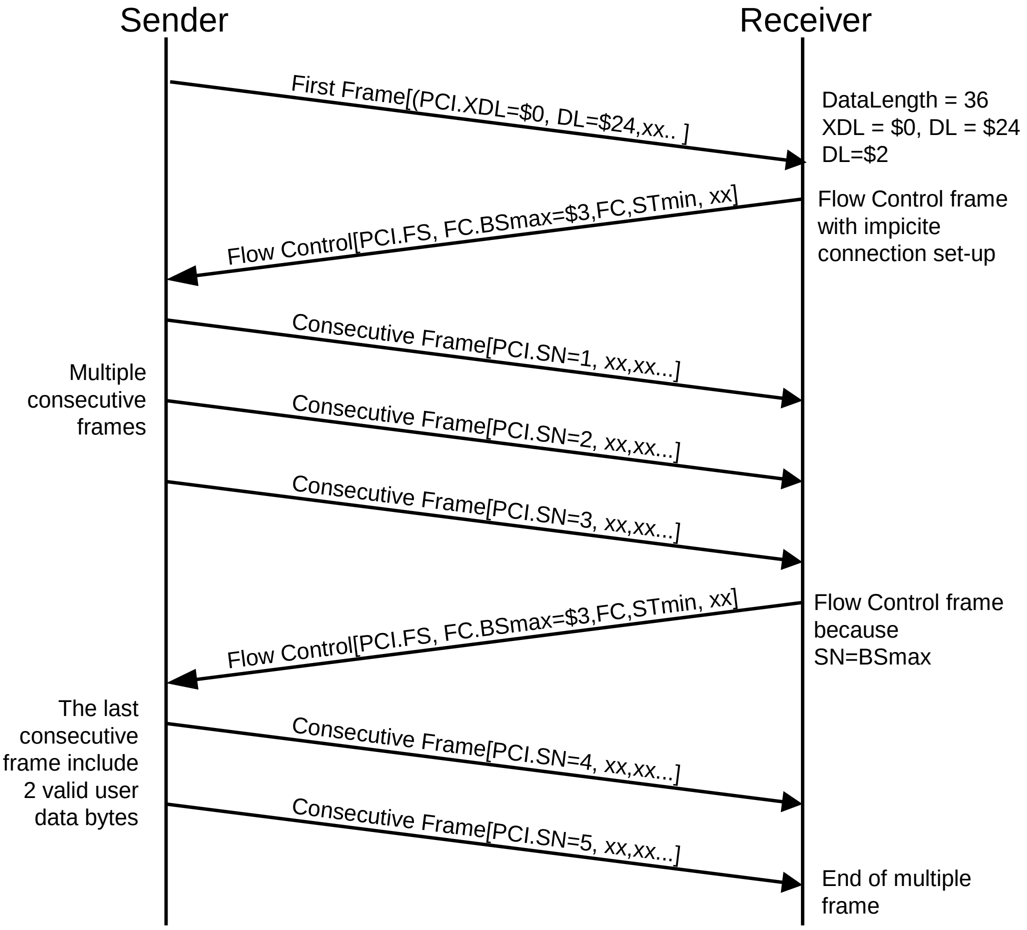

Sender
Receiver
First Frame[(PCI.XDL=$0, DL=$24,xx..]
DataLength = 36
XDL = $0, DL = $24
DL=$2
Flow Control[PCI.FS, FC.BSmax=$3,FC,STmin, xx]
Flow Control frame
with impicite
connection set-up
Multiple
consecutive
frames
Consecutive Frame[PCI.SN=1, xx,xx...]
Consecutive Frame[PCI.SN=2, xx,xx...]
Consecutive Frame[PCI.SN=3, xx,xx...]
Flow Control[PCI.FS, FC.BSmax=$3,FC,STmin, xx]
Flow Control frame
because
SN=BSmax
The last
consecutive
frame include
2 valid user
data bytes
Consecutive Frame[PCI.SN=4, xx,xx...]
Consecutive Frame[PCI.SN=5, xx,xx...]
End of multiple
frame

Figure 2-2 Construction of segmented message

2.1.2 Addressing modes

To handle the communication the Transport Protocol is using a Point-to-Point connection. To establish a Point-to-Point transfer on a broadcast protocol like CAN additional address information is needed (a source address for the transmit node and a target address for the receive node).

The ISO/TF2 transport protocol defines four modes of addressing:

„Normal“ addressing	The CAN ID contains the complete addressing information (to each source- and target address combination a unique CAN ID is assigned)
„Extended“ addressing	The CAN ID contains only the source address and the first data byte contains the target addressing information.
“Normal fixed” addressing	The extended CAN ID contains the complete addressing information according J1939
“Mixed” addressing	Additionally to the extended CAN ID, according J1939, the first data byte contains a second target address information. Since ISO15765-2: 2003 the additional addressing mode mixed addressing on 11-bit CAN IDs is defined. The address extension is stored in the first byte followed by the TPCI information.

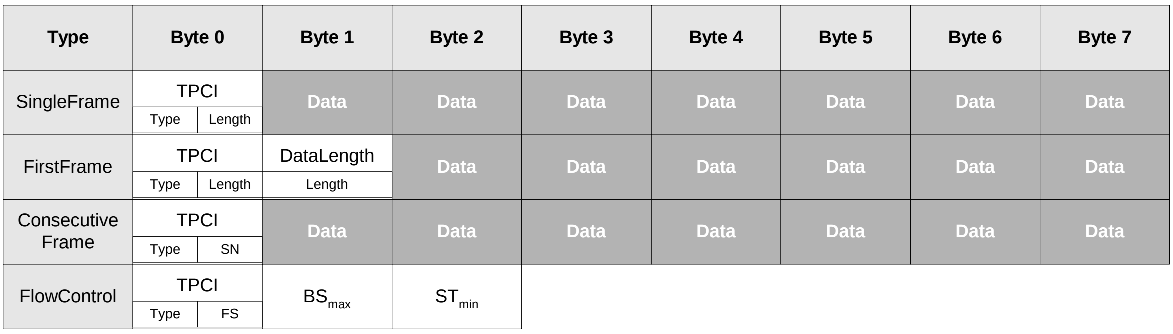
Table 2-1 Addressing Modes

The Vector TP implementation supports all addressing mode. The used addressing method is normally determined at compile-time regarding ROM and RAM as well as run-time requirements. For special purpose it is also possible to determine the used addressing method at run-time (special version of the TPMC-module is needed).

2.1.2.1 Normal Addressing

The address information is coded in a unique CAN Identifier.

The Transport Protocol uses the 1st and sometimes 2nd data byte. The data length is coded in 12bits. Therefore the maximum length of a message is limited to 4095 bytes. The receivers' control information (maximum block size and minimum SeparationTime) is transmitted to the sender within a FlowControl.



Type	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
SingleFrame	TPCI Type Length	Data	Data	Data	Data	Data	Data	Data
FirstFrame	TPCI Type Length	DataLength Length	Data	Data	Data	Data	Data	Data
Consecutive Frame	TPCI Type SN	Data	Data	Data	Data	Data	Data	Data
FlowControl	TPCI Type FS	BS_{max}	ST_{min}					

Table 2-2 Frame size on normal addressing

2.1.2.2 Mixed 11-bit ID Addressing

Mixed 11-bit addressing is a sub-format of normal addressing (refer above) where the mapping of the address information is further defined (see ISO 15765-2:2004).

The target **address extension** information is placed in the first data byte of the CAN frame (see ISO 15765-2:2004) followed by the TPCI information in byte two.

2.1.2.3 Normal Fixed Addressing

Normal fixed addressing is a sub-format of normal addressing (refer above) where the mapping of the address information into the (extended) CAN-Identifier is further defined (see ISO 15765-2).

J1939 name	P	R/DP	PF	PS	SA	Data field
Bits	3	2	8	8	8	64
Content	Priority	Reserved	ProtocolGroup Identification	Target-Address	Source-Address	TPCI/Data
CAN Id Bits	26-28	24-25	16-23	8-15	0-7	CAN data bytes
CAN Field	Identifier					Data

Table 2-3 CAN ID normal fixed addressing

For information about the "data field" see 2.1.2.1.

2.1.2.4 Extended Addressing

The source address is coded into the CAN ID by adding the address to a base CAN ID (e.g.: with a base CAN ID 0x600 and a source address of 0x10 the used CAN ID are 0x610)

The target address information is placed in the first data byte of the CAN frame (see ISO 15765-2).

Type	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
SingleFrame	**ext Addr**	TPCI Type Length	**Data**	**Data**	**Data**	**Data**	**Data**	**Data**
FirstFrame	**ext Addr**	TPCI Type Length	DataLength Length	**Data**	**Data**	**Data**	**Data**	**Data**
Consecutive Frame	**ext Addr**	TPCI Type SN	**Data**	**Data**	**Data**	**Data**	**Data**	**Data**
FlowControl	**ext Addr**	TPCI Type FS	BS$_{max}$	ST$_{min}$				

Table 2-4 Frame size extended addressing

2.1.2.5 Mixed 29-bit ID Addressing

Mixed 29-bit ID addressing is a sub-format of normal fixed addressing (refer above) where the mapping of the address information into the (extended) CAN-Identifier is further defined (see ISO 15765-2).

The target address extension information is placed in the first data byte of the CAN frame (see ISO 15765-2).

Type	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
SingleFrame	**Address Extension**	TPCI Type Length	**Data**	**Data**	**Data**	**Data**	**Data**	**Data**
FirstFrame	**Address Extension**	TPCI Type Length	DataLength Length	**Data**	**Data**	**Data**	**Data**	**Data**
Consecutive Frame	**Address Extension**	TPCI Type SN	**Data**	**Data**	**Data**	**Data**	**Data**	**Data**
FlowControl	**Address Extension**	TPCI Type FS	BS$_{max}$	ST$_{min}$				

Table 2-5 Frame size extended addressing

2.1.2.6 Structure of TPCI-Byte

The coding of the TPCI of each frame type is shown in Table 2-6 Structure of TPCI-bytes.

Encoding of Protocol Control Information (PCI)

1.	Network Protocol Control Information (N_PCI) bytes			
2.	Byte #1		Byte #2	Byte #3
N_PDU name	**Bits 7-4**	**Bits 3-0**		
SingleFrame	N_PCItype = 0	SF_DL	N/A	N/A
FirstFrame	N_PCItype = 1	FF_DL		N/A
ConsecutiveFrame	N_PCItype = 2	SN	N/A	N/A
FlowControl	N_PCItype = 3	FS	BS	STmin

Table 2-6 Structure of TPCI-bytes

Hex value	Description
0	**SingleFrame** For unsegmented message, the network layer protocol provides an optimised implementation of the network protocol with the message length embedded in the PCI byte only. SingleFrame (SF) shall be used to support the transmission of messages that can fit in a single CAN frame.
1	**FirstFrame** A first frame (FF) shall only be used to support the transmission of messages that cannot fit in a single CAN frame, i.e. segmented message. The first frame of a segmented message is encoded as a FirstFrame (FF). On receipt of a FirstFrame the receiving network layer entity shall start assembling the segmented message.
2	**ConsecutiveFrame** When sending segmented data, all consecutive frames following the first frame (FF) are encoded as ConsecutiveFrames (CF). On receipt of a Consecutive Frame (CF) the receiving network layer entity shall assemble the received data bytes until the whole message is received. The receiving entity shall pass the assembled message to the adjacent upper protocol layer after the last frame of the message has been received without error.
3	**FlowControl** The purpose of Flow Control is to regulate the rate at which Consecutive Frame network protocol data unit are sent to the receiver. Three distinct types of Flow Control protocol control information are specified to support this function. The type is indicated by a field of the protocol control information called Flow Status (FS) as defined hereafter.
4 - F	**Reserved** This range of values is reserved by this document.

SF_DL on SingleFrame	Contains the data length of the message (up to 7 bytes with normal resp. up to 6 bytes with extended addressing).
FF_DL on FirstFrame	Contains the data length of the message. The most significant 4 bit of the data length in byte #1, the remaining 8 bits are transmitted in byte #2.
SN on ConsecutiveFrame	The Sequence Number is used to discover a doubling or the loss of a data frame. The SN starts with ‘1’ and is calculated modulo ‘16’ (4 bit calculation).
FS on FlowControlFrame	‘0’ means CTS (ClearToSend): sender can continue sending ’1’ means WT (Wait): sender is not allowed to continue sending, it has to wait until FC.CTS is received ‘2’ means OVF (Overflow): sender is not allowed to continue sending, the transfer is stopped.

Table 2-7 Frames

2.2 Transmission

Only dynamic classes

TpTxGetFreeChannel: Associate channel to connection (only for dynamic classes)

The application has to allocate a free transport channel.

TpTxSet...: Adjust transmit state (only for dynamic classes)

The new allocated TpChannel has only blank communication parameters included, which await to be adjusted by the application. Which parameters have to be attuned depends on the used TpClass (see chapter 4.2 Functions of the Transport Protocol)

TpTransmit: Start the transmission

ApplTpTxCopyToCan: Copy data to CAN

The Transport Layer supports two copy mechanisms: an internal and an application specific copy mechanism.

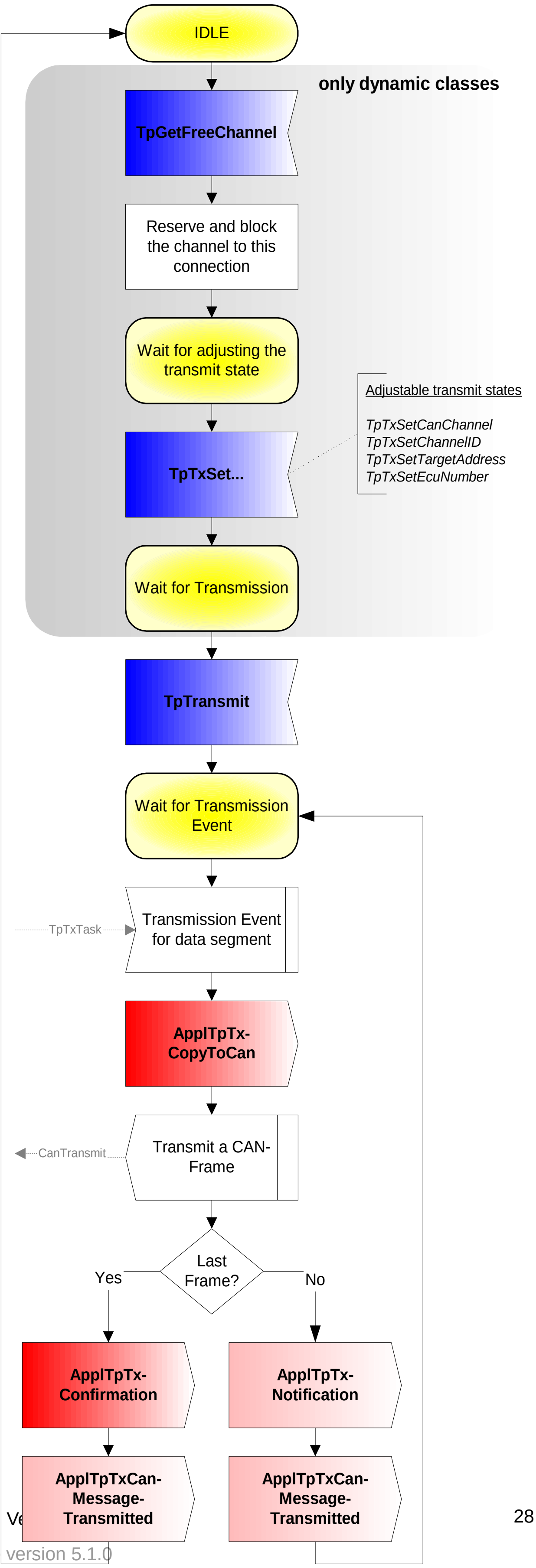
With the application specific copy mechanism the Transport Layer will call a callback function to request data each time data has to be transmitted.

ApplTpTxNotification / -CanMessageTransmitted

Each time a transport frame (every frame or only with pay load) will be transmitted, the Transport Layer notifies the application.

ApplTpTxConfirmation: Confirm the transmission

After a successful transmission the application will be notified. This would be a good point in time to release unused resources / buffers for example.



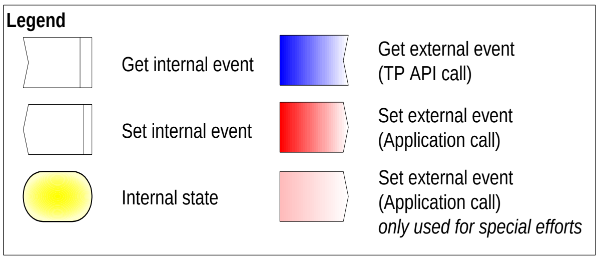

Legend
Get internal event
Get external event (TP API call)
Set internal event
Set external event (Application call)
Internal state
Set external event (Application call)
only used for special efforts

Figure 2-3 Transmission Architecture

2.3 Reception

ApplTpPrecopyCheck: Should receive or not?

The ApplTpPrecopy will be called immediately after the reception of each TP-Frame. The return value of the function determines whether or not the TP-Frame is received

ApplTpRxGetBuffer: Associate a buffer

The Transport Layer asks the application for a buffer. The application has to return a valid buffer, in which the received data will be stored. If the buffer is not valid, the reception will be abort.

ApplTpRxCopyFromCan: Copy data from CAN

The Transport Layer supports two copy mechanisms: an internal and an application specific copy mechanism.

The internal copy mechanism can only be used with a flat-buffer-model.

With the application specific copy mechanism the Transport Layer will invoke a callback function each time data were received.

ApplTpRxGetTxId: Get FlowControl ID
(only with Dynamic Normal Addressing)

A corresponding transmit ID for a FlowControl is needed.

ApplTpRxIndication: Indicate a reception

A complete block of transport frames is received.

Important: The Transport Layer blocks the receive channel to prevent a double occupancy of this channel. To free the receive channel the application can call **TpRxResetChannel ()**.

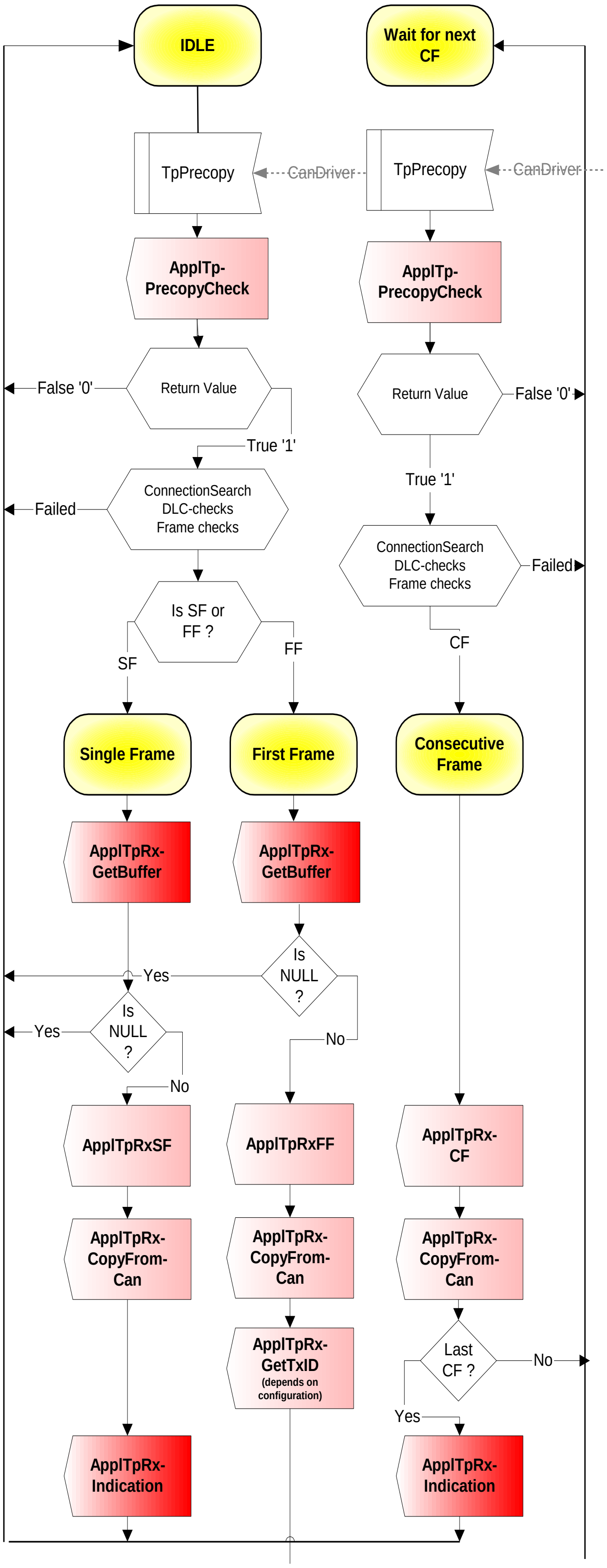


Figure 2-4 Reception Architecture

2.4 Working behaviors

2.4.1 Timings

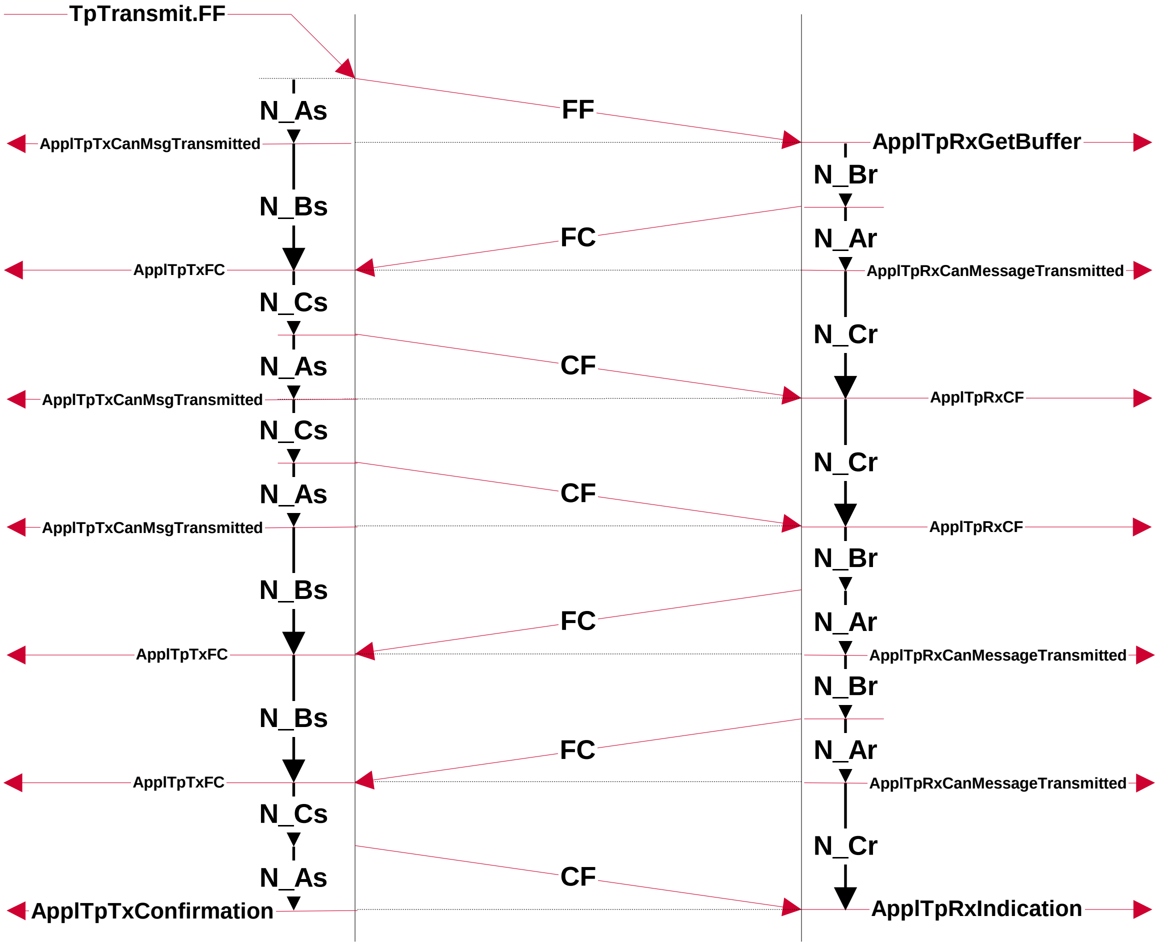


Figure 2-5 Transmission timings.

N_As	CAN message confirmation timeout	N_Ar	CAN message confirmation timeout
N_Bs	Timeout FC	N_Br	Always zero (0)
N_Cs	STmin (from FlowControl) But not lower than Transmit CF	N_Cr	Timeout CF

Table 2-8 Transmission timings

The TP needs the timings normalized to call cycles. Therefore all timings will be rounded up to an integer multiple of call cycles.

The timings have an inaccuracy while runtime (based on the technical concept where timers are set on interrupt level and decremented on task level). The jitter is either plus a call cycle or minus a call cycle.

In general the ‘Timings’ are calculated with a jitter plus a call cycle – that means the value of the timing is the first possible time after i.e. a timeout can occur.

The TP uses the following algorithm for calculation:

- Timings: (STmin-Value + (TpCallCycle-1)) / TpCallCycle + 1

2.4.2 Error detection

2.4.2.1 Reception of a SingleFrame

7	6	5	4	3	2	1	0
0	0	0	0	DL			

Single Frame

Data Lenth

Figure 2-6 Single Frame TPCI

A SingleFrame will be ignored if the DataLength exceeds the maximum length of a SingleFrame (6 / 7 bytes).

2.4.2.2 Reception of a FirstFrame

7	6	5	4	3	2	1	0
0	0	0	1	XDL			

First Frame

High Nibble of Data Length

Figure 2-7 First Frame TPCI

A FirstFrame will be ignored (until version 2.28) if the TPCIlength is lower than the maximum length of a SingleFrame (6 / 7 bytes).

2.4.2.3 Reception of a FlowControl

A FlowControl will be ignored if no suitable transmission is running (suitable means: the Source- and TargetAddresses must fit). It will be also ignored if the TPCIbyte misfit the valid values.

7	6	5	4	3	2	1	0
0	0	1	1	FS			

Flow Control

Flow State

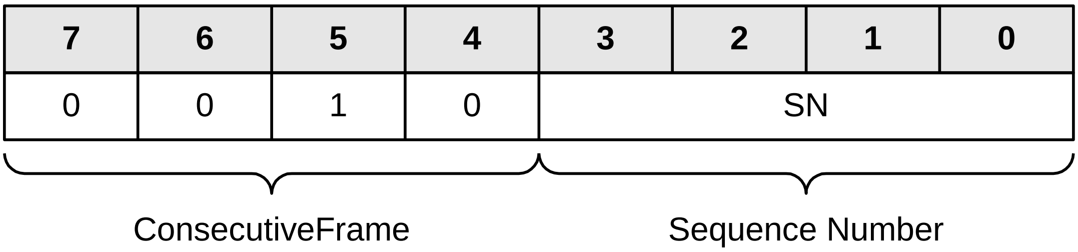
0	Continue To Send
1	Wait
2	Overflow (15765:2003)

Figure 2-8 FlowFrameTPCI

If a suitable transmission is found the state machine is checked for waiting for a FlowControl (except CAN Driver polling mode is used).

2.4.2.4 Reception of a ConsecutiveFrame

A ConsecutiveFrame will be ignored if no suitable reception is running (suitable means: the Source- and TargetAddresses must fit).



7	6	5	4	3	2	1	0
0	0	1	0	SN			

Figure 2-9 Consecutive Frame TPCI

If a suitable reception is found the state machine is checked for waiting of a ConsecutiveFrame (except CAN Driver polling mode is used). If the estimated Sequence Number does not fit the current reception will be stopped.

2.4.2.5 Observing CAN frame DLC (Data Length Code)

The CAN frame DLC should be set by the sender to a value greater than or equal to the values indicated in the table below.

Frame Type	Normal (fixed) Addressing	Extended/Mixed Addressing
SingleFrame	SF_DL+1	SF_DL+2
FirstFrame	8	8
FlowControl	3	4
ConsecutiveFrame (except the last ConsecutiveFrame)	8	8
Last ConsecutiveFrame	1+ ((FF_DL-6) mod[7])	2+ ((FF_DL-5) mod[6])

Table 2-9 CAN frame DLC

The CAN frame DLC check can be configured for the following different ways:

none:

CAN frames are accepted if they are 8 bytes or less.

The frames are NOT checked against a minimum length.

only DLC 8:

CAN frames are ONLY accepted if they are exactly 8 bytes long.

variable DLC:

CAN frames are accepted if they are 8 bytes or less.

The frames are also checked against the required minimum length.

depend on driver:

CAN frames are accepted if they pass the DLC check configured on driver level. Refer to [3] on how to set up the DLC check.

2.4.3 Buffer consistency

The application programmer has to guarantee consistency of transmission and reception buffers.

Transmission

Between the call of `TpTransmit()` and `ApplTpTxConfirmation()` or `ApplTpTxErrorIndication()` writing access to the transmission data buffer must be blocked (except the `ApplTpCopyToCan()` function is used to copy the data).

Reception

Between the call of `ApplTpRxGetBuffer()` and `ApplTpRxIndication()` or `ApplTpRxErrorIndication()` writing access to the reception data buffer must be blocked (except the `ApplTpCopyFromCan()` function is used to copy the data).

2.4.4 Function re-entrancy

The TP re-entrancy is based on the different tpChannels. So for static TP classes, with separate resources for each single connection, there is no re-entrancy problem. For dynamic TP classes the re-entrancy is guaranteed too from the viewpoint of TP, as long as the application handles the connection specific API properly.

2.5 Restriction

2.5.1 Restrictions to ISO/TF2 specification

In this chapter you will find the restrictions of the current implementation relative to the ISO/TF2-specification:

Timing parameter:

> Timing Parameter N_Br is always zero (0)

> Timing Parameter N_As and N_Ar can only be defined by a common constant

WaitFrame support:

For versions until version 2.73.00:

> The reception of WaitFrames is supported. The transmission of WaitFrames is not supported, N_WFTmax is always zero (0).

For versions until version 2.88.00:

> Commencing with version 2.73.00 the transmission of WaitFrames is supported but N_WFTmax is not considered. The periodical transmission must be stopped by the application and does not stop by itself.

From version 2.89.00:

> Commencing with version 2.89.00 the maximal number of WaitFrames to be transmitted (N_WFTmax) is supported and the transmission of WaitFrames stops automatically when this limit is exceeded.

From version 3.01.00:

> Commencing with version 3.01.00 the maximal number of WaitFrames to be received (N_TxWFTmax) is supported and the reception of WaitFrames stops automatically when this limit is exceeded.

2.5.2 Limitations of Transport Protocol Implementation

The Transport Protocol is a complex state machine, which is triggered by external events like requests by the application, receive indications and transmit confirmations by the CAN driver.
The state machine expects those events in the order they appear in the “real world” to decide the next step to be performed. The state machine performs one event after the other and each decision is based on the current state.

Under some very specific conditions, events may be given to the Transport Protocol state machine in the incorrect order what can cause wrong decisions.

One requirement to the TP is that unexpected frames are to be ignored. Therefore it is important to discard e.g. received FlowControl frames before the FirstFrame or

ConsecutiveFrame has been sent. It may now happen that the transmit confirmation and the receive indication event occur “at the same time”. In such a situation the concrete behaviour depends on the sequence the underlying CAN driver handles such events. Unfortunately this sequence depends on the hardware implementation of the CAN controller and the interrupt concept of the µC. Usually RX handling is done first to prevent loss of incoming data whereas TX handling has a lower priority. Most CAN controllers do not support means to handle such events in the “real world order” later, if an immediate handling is not possible due to e.g. an long lasting ISR lock or the CAN driver polling is executed too slow.

Example:
The TP transmits its FirstFrame successfully to the bus and the tester answers very fast with the FlowControl and the notification of the FirstFrame transmit event is delayed due to (a) an ISR lock or (b) a too slow polling sequence, both events are valid at the same time. Now it is up to the CAN driver how the notification sequence is performed.
If TX is handled first, TP is in a state to accept the FlowControl and everything went well. If RX is handled first, TP is not aware that the FirstFrame has been already sent and will ignore the incoming FlowControl. In that case, the TP runs in a timeout due to the partner has sent its frame correctly but it was assigned to the wrong event sequence and was therefore ignored.

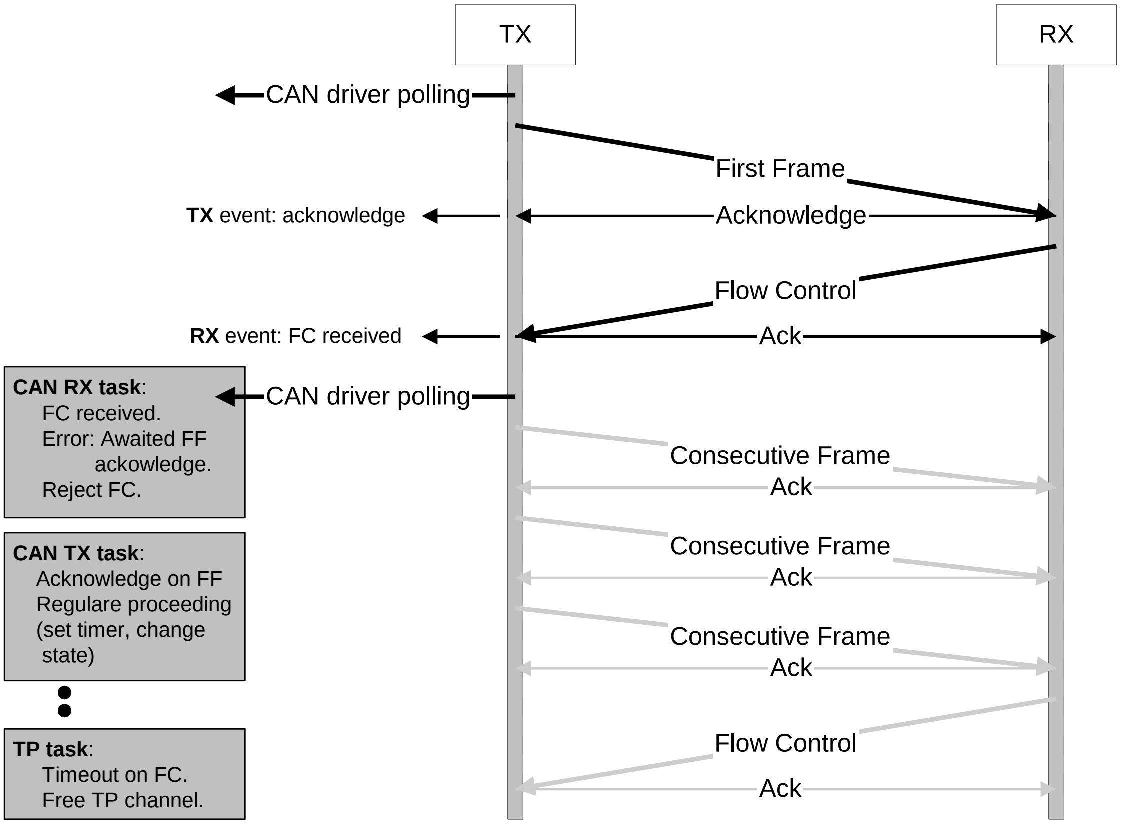


Figure 2-10 Accumulation of events during CAN Driver polling

Implemented solution 1:
The TP can be configured to handle the event sequence always in the way it is notified by the underlying driver. In that case it is fully compliant to the requirement that (timely)

incorrect frames are rejected. Unfortunately, the rejection can happen in a short time period also for correct transmitted frames. The time period where this can happen is equal to the runtime of the e.g. FlowControl frame on the bus (e.g. for DLC=8 and 500kBd this is approx. 200µs for an interrupt driven CAN driver or the CAN driver polling rate). Timely incorrect received frames outside of this time window are correctly handled/rejected.

As a result, the correctly transmitted TP sequence might be aborted by a timeout on the sender side and the tester has to repeat its request.

The configuration switch TP_HIGH_RX_LOW_TX_PRIORITY has to be kTpOff to select the implementation 1.

Implemented solution 2:
The TP can be configured to accept FlowControl frames also in the time window after the successful ECU internal FirstFrame transmit request till the frame is really on the bus. In that case it is not fully compliant to the requirement that (timely) incorrect frames are rejected. The length of the time period depends on the baudrate (message runtimes), the busload and if the CAN driver is used in ISR or polling mode. The shortest time range is some few 10µs up to a multiple of the CAN driver polling rate. Timely incorrect received frames outside of this time window are correctly handled/rejected.

As a result of this behaviour, a too early (timely incorrect) received FlowControl frame will be accepted by the TP and the transport layer continues to transmit its data.
Because this scenario does usually not or only rarely happen in the field but the performance of the whole diagnostic process is higher, the selection of that configuration is highly recommended.

The configuration switch TP_HIGH_RX_LOW_TX_PRIORITY has to be kTpOn to select the implementation 2.

ℹ	**Info** Please note that the content of the received frame is always analyzed and illegal frames are discarded as required. All above discussed issues are only valid if the frame is timely incorrect but all other facts are correct concerning the current TP status.

⚠	**Caution** Implementation solution 2 is automatically activated since version 2.36 of TPMC component while the CAN Driver is used in polling mode. It is activated as default for interrupt driven systems since version 2.63..

2.5.3 Deviations to ISO/TF2 specification

In this chapter you will find the deviations of the current implementation compared to the ISO/TF2-specification.

2.5.3.1 Handling of unexpected FlowControl / ConsecutiveFrame frames

Caution

This deviation is only in effect if the TP_HIGH_RX_LOW_TX_PRIORITY feature is kTpOn.

The normal operation assumes that a transmit is followed first by its confirmation interrupt and after that the next receive interrupt appears.

With a tester reacting very fast and simultaneously a controller that has a higher priority for Rx interrupts than for Tx interrupts the Rx interrupt may be detected before the Tx confirmation interrupt:

- Without the HighRx-LowTx feature the transmission stops at this point.
- With the activation of the HighRx-LowTx feature the TP implementation tries to clear this unexpected sequence and to proceed with the transmission. Nevertheless there are still some special situations left (see the description above) that can not be cleared by the TP and so the transmission might be stopped anyway.

Conclusion:

The HighRx-LowTx feature is activated by default to get a minimum of transmissions being stopped.

You can deactivate the feature e.g. if your configuration does not require the feature or if you prefer that the tester explicitly repeats requests after stopped transmissions.

Please see the description below to get an idea in which special situations some malfunction is still possible.

See also chapter 2.5.2 ‘Limitations of Transport Protocol ’ for further details.

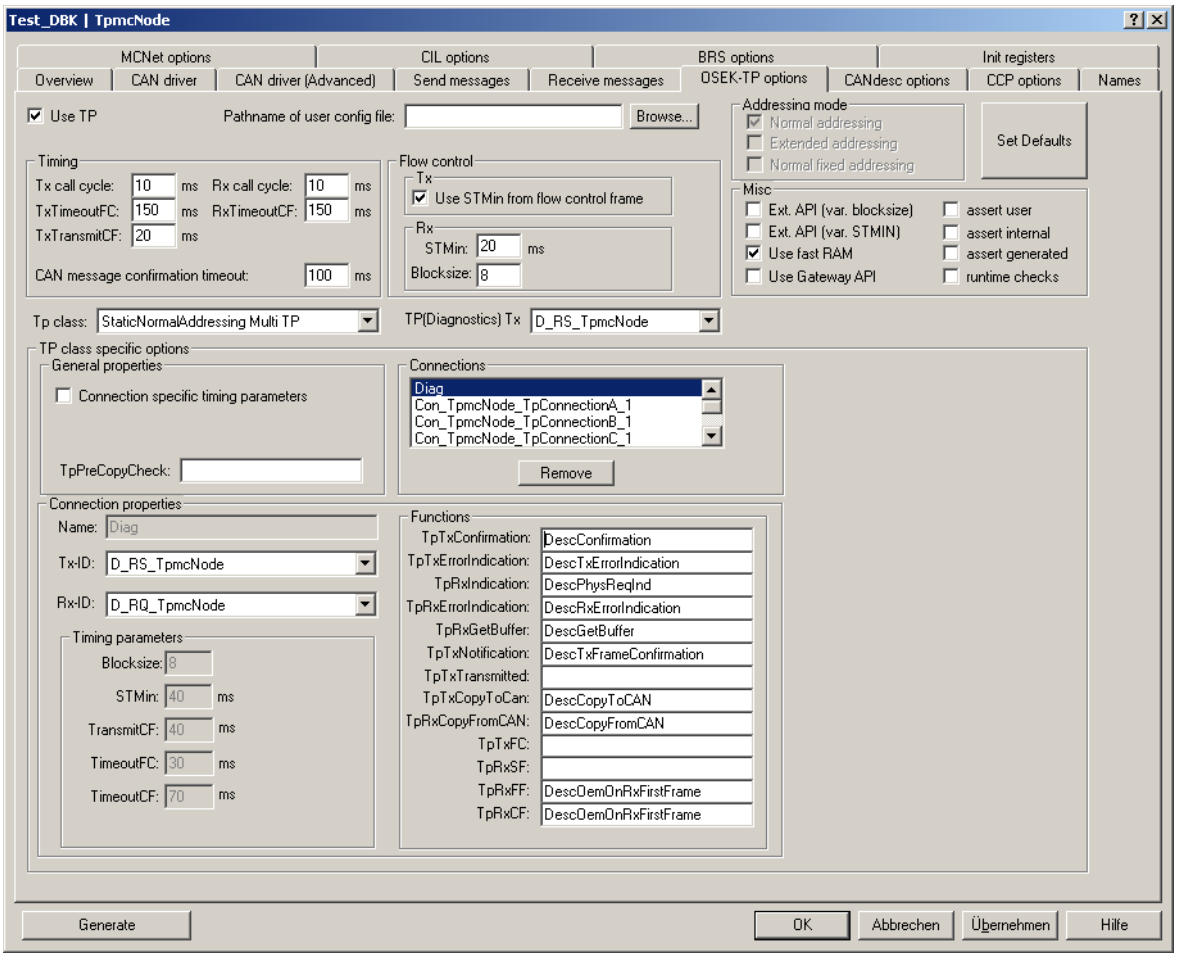
3 Settings for the MultiTP & SingleTP (multi-based)

To use the MultiConnection or the SingleConnection (multi-based) TP with the GENy CANGen or the DBKOMGen tool the “Manufacture” attribute in the database has to be set.

Additionally a License File for GENy and CANGen tool is needed, which includes a clearing for the MultiConnection Tp.

3.1 General settings with CANgen / DBKOMgen / GENy

In the following descriptions examples from the CANGen / DBKOMGen generation tool GUI are used.



☑ Use TP

Pathname of user config file:

Addressing mode
☑ Normal addressing
□ Extended addressing
□ Normal fixed addressing

Timing
Tx call cycle: 10 ms
Rx call cycle: 10 ms
TxTimeoutFC: 150 ms
RxTimeoutCF: 150 ms
TxTransmitCF: 20 ms
CAN message confirmation timeout: 100 ms

Flow control
Tx
☑ Use STMin from flow control frame
Rx
STMin: 20 ms
Blocksize: 8

Misc
□ Ext. API (var. blocksize)
□ Ext. API (var. STMIN)
☑ Use fast RAM
□ Use Gateway API
□ assert user
□ assert internal
□ assert generated
□ runtime checks

Tp class: StaticNormalAddressing Multi TP

TP(Diagnostics) Tx: D_RS_TpmcNode

TP class specific options

General properties
□ Connection specific timing parameters
TpPreCopyCheck:

Connections
Diag
Con_TpmcNode_TpConnectionA_1
Con_TpmcNode_TpConnectionB_1
Con_TpmcNode_TpConnectionC_1

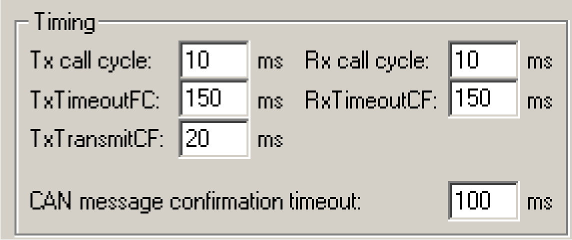
Connection properties
Name: Diag
Tx-ID: D_RS_TpmcNode
Rx-ID: D_RQ_TpmcNode

Timing parameters
Blocksize: 8
STMin: 40 ms
TransmitCF: 40 ms
TimeoutFC: 30 ms
TimeoutCF: 70 ms

Functions
TpTxConfirmation: DescConfirmation
TpTxErrorIndication: DescTxErrorIndication
TpRxIndication: DescPhysReqInd
TpRxErrorIndication: DescRxErrorIndication
TpRxGetBuffer: DescGetBuffer
TpTxNotification: DescTxFrameConfirmation
TpTxTransmitted:
TpTxCopyToCan: DescCopyToCAN
TpRxCopyFromCAN: DescCopyFromCAN
TpTxFC:
TpRxSF:
TpRxFF: DescOemOnRxFirstFrame
TpRxCF: DescOemOnRxFirstFrame

Figure 3-1 General settings in Generation Tools

3.1.1 Timing



Timing

Tx call cycle: 10 ms Rx call cycle: 10 ms

TxTimeoutFC: 150 ms RxTimeoutCF: 150 ms

TxTransmitCF: 20 ms

CAN message confirmation timeout: 100 ms

Figure 3-2 Timing settings in Generation Tools

3.1.1.1 Transmission timing

Tx call cycle

Together with this period, the function TpTxTask() has to be called periodically by the application

TxTimeoutFC

In the Timeout FC edit field, the FlowControl timeout value is specified. Within this time, the expected FC frame has to be received by the transmitting ECU.

TxTransmitCF

The Transmit CF time is the interval for the transmission of ConsecutiveFrames. This value is used as a constant in ECUs that don’t use the STmin value from FlowControl frame.

If this time should be defined as a constant at compile time the configuration switch “Use ST_{Min} from flow control frame” should be set to Off.

If the time ST_{Min} from the FlowControl message should be calculated, the configuration switch “Use ST_{Min} from flow control frame” has to be selected. Due to the problem to handle a non-linear buffer (e.g. ring-buffer mechanism) in the application (usage of ApplTpCopyToCAN or Vector Diagnostic Layer) the Transmit CF parameter set the fastest possible transmission.

Transmit CF set the lowest possible separation time.

Example: The Diagnostic Tester set the STmin value to zero. Which will mean to the ECU to transmit as fast as possible. If the application uses in this case a ring-buffer mechanism it has to fill the ring-buffer in the same fast way as the TP transmits the data. To prevent in such a case a buffer under-run it is possible to limit the TP, by setting the lowest possible separation time value, so that the calculated STmin cannot be smaller than the Transmit CF value.

3.1.1.2 Reception timing

Rx call cycle

Together with this period, the function TpRxTask() has to be called periodically by the application

RxTimeoutCF

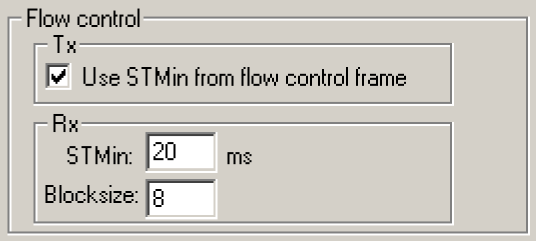
After the Timeout CF time expires, a time-out occurs with the transport layer between the receptions of two ConsecutiveFrames.

3.1.1.3 Common timing

CAN message confirmation timeout

Maximum time between a transmission request and the confirmation interrupt, indicating that the frame is sent successfully (it is at least accepted by one net node).

3.1.2 Flow Control



Flow control

Tx

☑ Use STMin from flow control frame

Rx

STMin: 20 ms

Blocksize: 8

Figure 3-3 Flow control settings in Generation Tools

3.1.2.1 Transmission

Use ST_{Min} from flow control frame

If the “Flow control” time ST_{Min} was defined as constant at compile time for the whole system, it won’t be necessary to calculate it at runtime. Setting the configuration switch „Use STMin from FlowControl frame“ to Off can parameterize this.

If the time ST_{Min} from the FlowControl message should be calculated, the configuration switch “Use ST_{Min} from FlowControl frame” has to be selected.

3.1.2.2 Reception

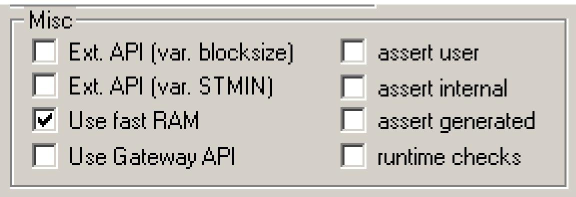
STMin

The STmin edit field contains the minimum separation time between two consecutive frames. The separation time will be at least as long as configured or longer. The value in this edit field will be transmitted to the sender ECU in the FlowControl frame from the current ECU. The STmin value can either be defined at compile time or changed at runtime (see also 3.1.3 Extended API STmin).

BlockSize requested

The BlockSize specifies the number of ConsecutiveFrames until a FlowControl is needed. The receiver defines the BlockSize. The sender always uses the BlockSize of the receiver. The BlockSize value can either be defined at compile time or changed at runtime (see also 3.1.3 Extended API BlockSize).

3.1.3 Misc



Misc

☐ Ext. API (var. blocksize)
☐ Ext. API (var. STMIN)
☑ Use fast RAM
☐ Use Gateway API
☐ assert user
☐ assert internal
☐ assert generated
☐ runtime checks

Figure 3-4 Misc. settings in Generation Tools

Extended API (variable BlockSize)

API extension, which can adjust the BlockSize value.

If the feature is enabled the BlockSize can be set at run-time by using the functions TpRxSetBS() and TpRxGetBS().

Default value after initializations: “BlockSize requested” (Section ‘Flow Control’)

Extended API(variable STmin)

API extension, which can adjust the STmin value.

If the feature is enabled the STmin value can be set at run-time by using the functions TpRxSetSTMIN() and TpRxGetSTMIN().

Default value after initializations: “STmin” (Section ‘Flow Control’)

Use fast RAM

The RAM demand and run-time can be reduced on some implementations, if some frequently used variables of the Transport Protocol are put into the „near“ memory.

If the feature is enabled (default) the less used variables are also set into the „near“-memory. The code is smaller and faster.

Otherwise less used variables are not set into the „near“-memory. The code is a little bit bigger and slower.

Use Gateway API

API extension, which was implemented for Gateway purpose, but it is also possible to use it in other fields of applications.

If the feature is enabled the API of ‘ApplTpRxGetBuffer’ and ‘ApplTpRxCheckTA’ is extended with the CanRxInfoStructPtr from the CAN Driver Precopy functions API (see /CANDrv/ manual).

Within this CanRxInfoSturctPtr parameter the CAN ID, pointer to the CAN data, etc. is included.

Assertions

To detect some incorrect internal conditions of the Transport Protocol during development, integration and software test, there are different categories of so called assertions configurable:

1. User interface (for example input parameters, reentrance if not allowed)

2. Generated data
3. Internal software errors (for example inconsistent internal states)

Each type of assertion can be configured independently.

These assertions will help in different development phases to deal with unexpected problems, which cannot be handled by the Transport-Protocol internally. In such case the following callback function will be called by the Transport-Protocol:

```
void ApplTpFatalError( vuint8 errorNumber );
```

This callback function has to be provided by the Application. The function parameter errorNumber gives more detailed information about the kind of error, which is occurred (see also 4.4.4.1 ApplTpFatalError: Fatal Error for the different error-codes).

Generally, the error number has to be checked to solve the underlying problem. The recovery strategy is application dependent, but mostly there is a complete reset necessary to set up the software correctly again.

⚠	**Caution** This callback function must not return to the Transport-Protocol afterwards.

assert user

User assertion will be activated.

Should be used while development of Application software

assert internal

Internal assertions will be activated.

Should be used for tests of software changes in the Transport-Protocol

(Vector internal)

assert generated

Internal assertions will be activated.

Should be used if a new version of the Generation Tool is used

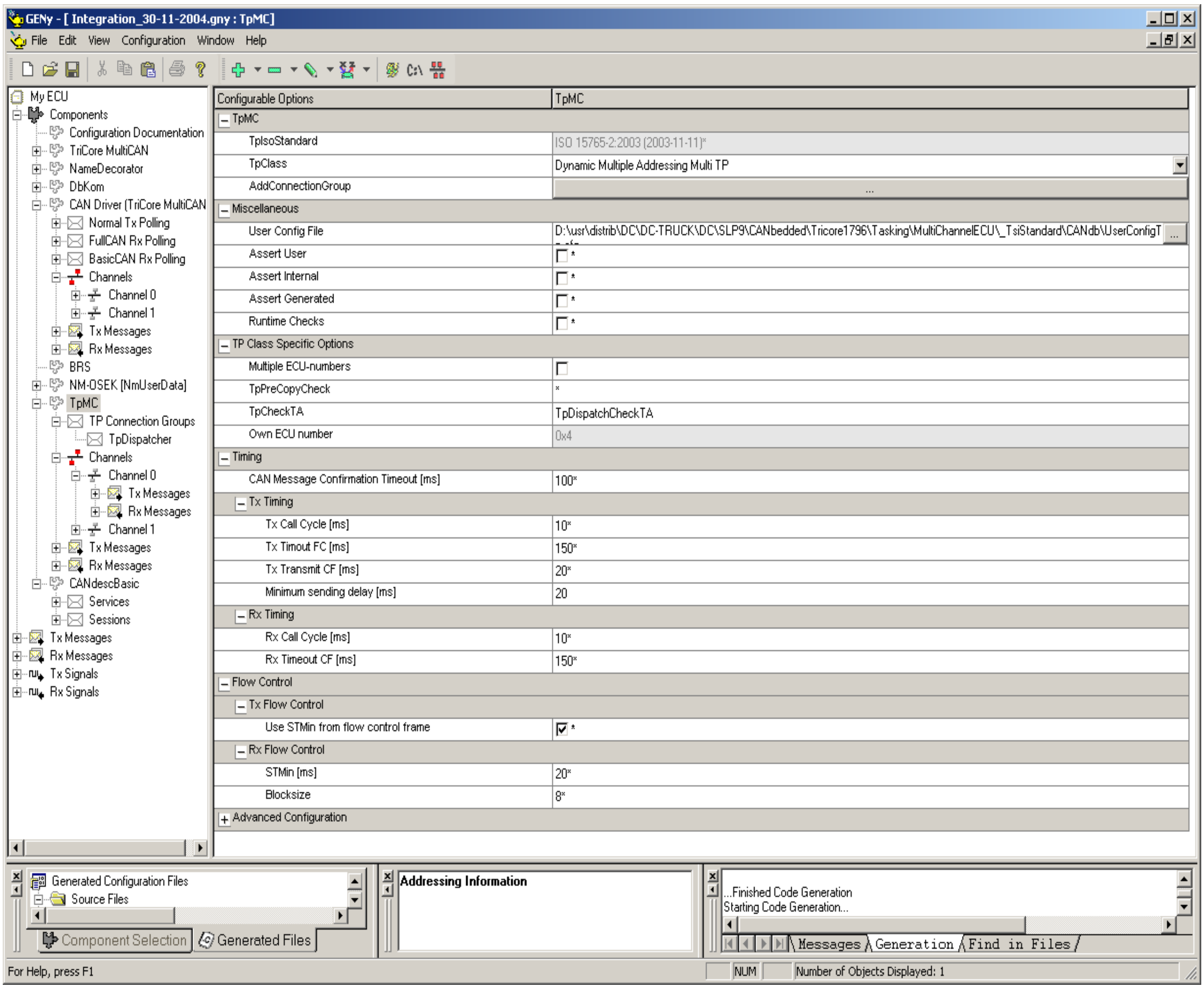
runtime checks

Runtime checks will be activated.

In contrast to the assertions the ‘runtime checks’ can also be used after the development phase and should guarantee a more reliable run. Checks for parameter plausibility, overwriting of memory like beyond access of tables, etc..

3.2 General settings with Generation Tool GENy

General settings can be done under the TPMC tree element. Most important is the selection of TpClass in the upper right window. Some online help is provided for the most settings in the OnScreenHelp window. Section “Advanced Configuration” is providing special features like Gateway APIs or padding of TP frames. Some features might be greyed which means that this features are preconfigured based on OEM or other constraints. It is necessary to configure for each Tp class at least one “TP Connection Group” object. Some static configured TP classes like “Static Normal Multi TP” require one Connection Group object for each TP connection whereas dynamic TP classes have always only one object. A Connection Group object represents a set of call back functions for the application to notify successful transmission or reception.



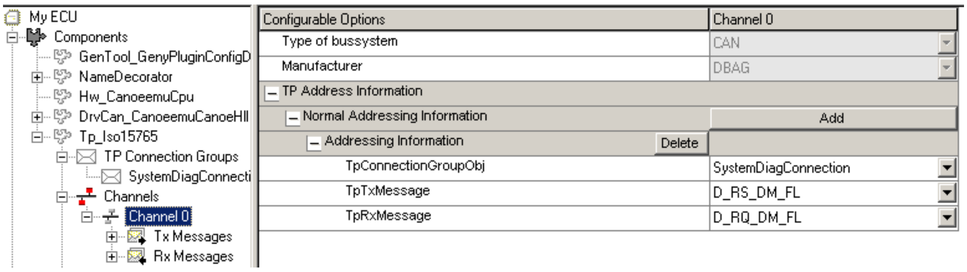
Configurable Options	TpMC
– TpMC	
TpIsoStandard	ISO 15765-2:2003 (2003-11-11)*
TpClass	Dynamic Multiple Addressing Multi TP
AddConnectionGroup	...
– Miscellaneous	
User Config File	D:\usr\distrib\DC\DC-TRUCK\DC\SLP9\CANbedded\Tricore1796\Tasking\MultiChannelECU_TsiStandard\CANdb\UserConfigT
Assert User	□ *
Assert Internal	□ *
Assert Generated	□ *
Runtime Checks	□ *
– TP Class Specific Options	
Multiple ECU-numbers	□
TpPreCopyCheck	*
TpCheckTA	TpDispatchCheckTA
Own ECU number	0x4
– Timing	
CAN Message Confirmation Timeout [ms]	100*
– Tx Timing	
Tx Call Cycle [ms]	10*
Tx Timout FC [ms]	150*
Tx Transmit CF [ms]	20*
Minimum sending delay [ms]	20
– Rx Timing	
Rx Call Cycle [ms]	10*
Rx Timeout CF [ms]	150*
– Flow Control	
– Tx Flow Control	
Use STMin from flow control frame	☑ *
– Rx Flow Control	
STMin [ms]	20*
Blocksize	8*
+ Advanced Configuration	

```
...Finished Code Generation
Starting Code Generation...
```

Figure 3-5 Main window of component TPMC within configuration tool GENy.

3.2.1 Configuration of Addressing Information

The addressing information is configured for each channel. The provided addressing elements like TpTxMessage (for NormalAddressing) depend on the selected TP class. It is required to assign a TpConnectionGroupObj for each Addressing information. In Dynamic Multiple Addressing Tp Classes any Addressing Information is assigned to only one TP Connection Group Object.



Configurable Options	Channel 0
Type of bussystem	CAN
Manufacturer	DBAG
TP Address Information	
Normal Addressing Information	
Addressing Information	
TpConnectionGroupObj	SystemDiagConnection
TpTxMessage	D_RS_DM_FL
TpRxMessage	D_RQ_DM_FL

Figure 3-6 Main window of component TPMC within configuration tool GENy.

3.2.2 Usage of Far RAM buffers

Due to reasons of RAM resource availability it may be necessary to locate the receive and transmit buffers handed to the TP in a far memory location. All message buffer related types and callbacks will then use far pointers.

To enable this option the “Use far RAM buffers” option within the “Advanced Configuration” tab must be enabled.

If that option does not suffice for your integration the “Memory Model Override” option can be used alternatively supporting the usage of a special qualification string that can be entered as plain text (e.g.: @page @far).

3.2.3 Non standard handling of Flow Control frames

3.2.3.1 Reserved STmin Handling

According to ISO 15765-2 the STmin values 0x80-0xF0 and 0xFA-0xFF are reserved.

If a received FC.CTS frame nevertheless uses one of these reserved values, it shall be interpreted from the TP as the maximum STmin time (0x7F) which is defined.

The TP supports two additional possibilities to handle reserved STmin values:

> If the switch ‘TP_ENABLE_IGNORE_FC_RES_STMIN’ is defined, then a FC frame with a reserved STmin value is silently ignored.

> If the switch ‘TP_ENABLE_CANCEL_FC_RES_STMIN is defined, then a FC frame with a reserved STmin value will lead to the cancellation of the Tx connection.

Note that each switch has only an effect if the STmin is evaluated at all. For cases where STmin might not be evaluated, please refer to 3.2.3.4 and 3.2.3.5.

3.2.3.2 Ignore Flow Control Overflow

According to ISO 15765-2 a received FC.OVFLW (0x32) will abort the ongoing transmission due to the lack of reception buffer at the receiver side.

If the switch 'TP_ENABLE_IGNORE_FC_OVFL' is defined then a FC.OVFLW frame is silently ignored instead.

3.2.3.3 Do not ignore unexpected Flow Control frames

According to ISO 15765-2 any unexpected FC frame shall be ignored.

If the switch TP_USE_UNEXPECTED_FC_CANCELATION is set to kTp_On, this behavior is changed. Then every unexpected FC frame will cancel the current transmission.

3.2.3.4 Use STmin of FC

According to ISO 15765-2, the STmin from an FC.CTS shall be used as separation time between two consecutive frames.

If the switch TP_USE_STMIN_OF_FC is set to kTp_Off, the STmin of the FC is ignored. Instead, the configured N_Cs timeout (TxTransmitCF parameter, see 3.1.1.1) is used as STmin.

3.2.3.5 Analyze first FC only

According to ISO 15765-2, the contents of each expected and received FC.CTS shall be evaluated by a transmitter in order to adjust its BS and STmin values.

If the switch TP_USE_ONLY_FIRST_FC is set to kTp_On, only the BS and the STmin of the first received FC.CTS are evaluated. These values are then used for the complete transmission. Further received FC.CTS are only used for synchronization and not to adjust BS and STmin.

3.3 Additional settings via user-configuration file

3.3.1 Dynamic Timing API

Using this feature the application can dynamically change connection specific timing values for:

- CAN confirmation timeout (N_Ar, N_As)
- Consecutive Frame timeout (N_Cr)
- Flow Control timeout (N_Bs).

The dynamic channel timing feature can be enabled via a user configuration file. If the pre-processor switch "TP_ENABLE_DYN_CHANNEL_TIMING" is included in this way then the TP takes the timing values from the following application provided variables:

tTpEngineTimer tpRxConfirmationTimeout [kTpRxChannelCount];

tTpEngineTimer tpTxConfirmationTimeout [kTpTxChannelCount];

```
tTpEngineTimer tpRxTimeoutCF              [kTpRxChannelCount];
tTpEngineTimer tpTxTimeoutFC              [kTpTxChannelCount];
```

tTpEngineTimer is usually of type canuint16, for 8-bit CPUs it might also be defined as canuint8.

These variables are initialized internally from the TP with the constant values that are configured in the generation tool. So all connection specific timing are equal after TP initialization.

⚠	Please note that the TP expects these variables, containing the connection specific timing values, to be supplied by the application.

For the further dynamic adaptation and differentiation of these connection specific values the following API functions are available in addition:

- `TpRxSetTimeoutConfirmation (see 4.2.2.25 )`
- `TpTxSetTimeoutConfirmation (see 4.2.3.26)`
- `TpRxSetTimeoutCF ( see 4.2.2.26 )`
- `TpTxSetTimeoutFC (see 4.2.3.27)`

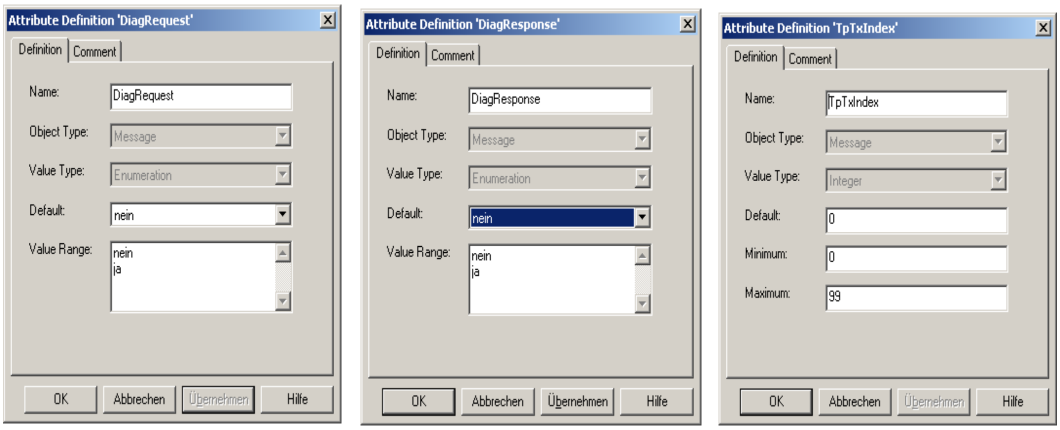
With these functions the belonging timeout values of the TP can be changed dynamically during runtime.

3.4 TP classes: SingleTP (multi-based)

These TP classes are based on the MultiTP core but running only with one connection and are optimized to consume a minimum of resources.

3.4.1 Database Attributes

Following Database attributes are needed:



Attribute Definition 'DiagRequest'

Definition | Comment

Name: DiagRequest

Object Type: Message

Value Type: Enumeration

Default: nein

Value Range: nein
ja

Attribute Definition 'DiagResponse'

Definition | Comment

Name: DiagResponse

Object Type: Message

Value Type: Enumeration

Default: nein

Value Range: nein
ja

Attribute Definition 'TpTxIndex'

Definition | Comment

Name: TpTxIndex

Object Type: Message

Value Type: Integer

Default: 0

Minimum: 0

Maximum: 99

Figure 3-7 Database Attributes for Single/Static TP classes

DiagRequest / DiagResponse: Used for diagnostic request messages to make special pre-settings for the Vector diagnosis's layers.
(Only available for some car-manufactures)

TpTxIndex: Used for application TP messages.

TP connections with FlowControl: bi-directional transmissions according to ISO 15765 standard

TP-connections without FlowControl: unidirectional transmissions nonconformance to the ISO 15765 standard

conventions to read a connection out of the database	
bidirectional with FC (standard)	The TX-Node and the RX-Node includes each a TX-TP-message with the same TpTxIndex value {Broadcast not possible}.
bidirectional without FC	not supported
unidirectional with FC	not supported
unidirectional without FC (not supported in SingleTP classes)	The RX-Node do not include a TX-TP-message with a same TpTxIndex as the TX-TP-msg. of the TX-Node {Broadcast is possible - TX-msg. can have more than one receiver}.

Table 3-1 Usage of TpTxIndex database attribute

GenMsgDelayTime:

If the database attribute 'GenMsgDelayTime' has a value unequal to zero, then the TP observes this time between two transmissions as a minimum time distance.

3.4.2 TP class SingleTP (multi-based): Normal Addressing

No special settings needed

3.4.3 TP class SingleTP (multi-based): Extended Addressing

No special settings needed

3.4.4 TP class SingleTP (multi-based):Normal Fixed Addressing

3.4.4.1 Database Attributes

Refer to chapter 3.6.6.1 Database Attributes

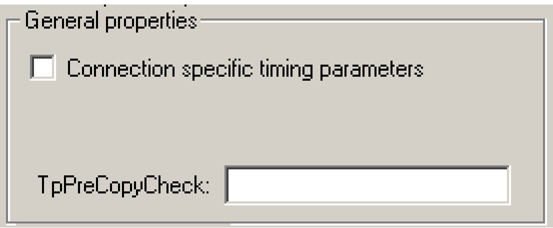
3.5 TP classes Static MultiTP

For each TP-communication between two ECUs static defined connections are available.

3.5.1 Database Attributes

Refer to chapter 3.4.1 Database Attributes

3.5.2 TP class specific settings



General properties

☐ Connection specific timing parameters

TpPreCopyCheck:

Figure 3-8 Additional TP settings (Static MultiTP) in Generation Tool

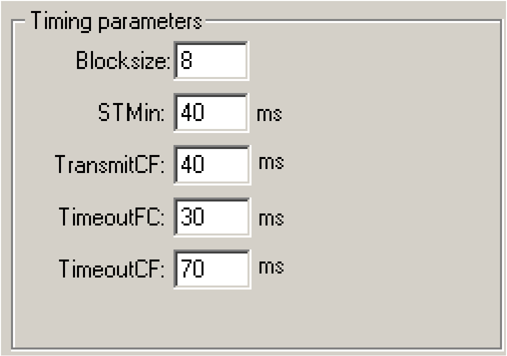
Connection specific timing parameters

If ‘Connection specific timing parameters’ are activated the timing parameters of each connection can override the global timing values for this connection.

TpPreCopyCheck

Just enter a function name to use this hook function.

3.5.3 Connection specific timing parameters



Timing parameters

Blocksize: 8

STMin: 40 ms

TransmitCF: 40 ms

TimeoutFC: 30 ms

TimeoutCF: 70 ms

Figure 3-9 Connection specific timing parameters

The following parameters can be configured individually for each connection:

Timings

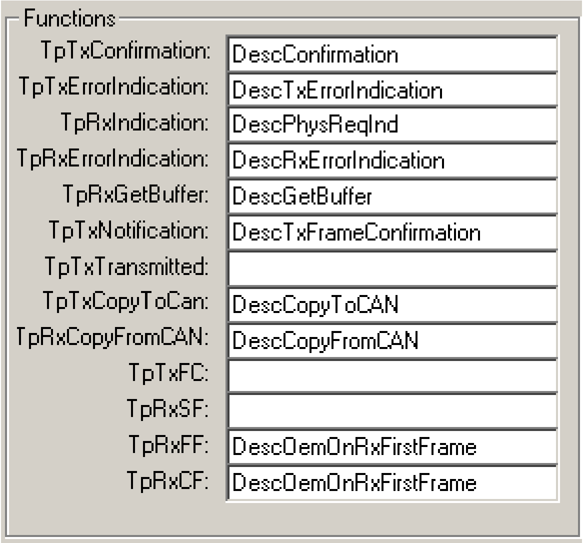
- TxTimeoutFC
- TxTimeoutCF
- RxTransmitCF

FlowControl

- STMin
- Requested BlockSize

For detailed descriptions refer chapter 3.1.1 Timing and the following

3.5.4 Functions



Functions

TpTxConfirmation:	DescConfirmation
TpTxErrorIndication:	DescTxErrorIndication
TpRxIndication:	DescPhysReqInd
TpRxErrorIndication:	DescRxErrorIndication
TpRxGetBuffer:	DescGetBuffer
TpTxNotification:	DescTxFrameConfirmation
TpTxTransmitted:	
TpTxCopyToCan:	DescCopyToCAN
TpRxCopyFromCAN:	DescCopyFromCAN
TpTxFC:	
TpRxSF:	
TpRxFF:	DescOemOnRxFirstFrame
TpRxCF:	DescOemOnRxFirstFrame

Figure 3-10 Hook-Functions (Static MultiTP)

Just enter a suitable function name to use the hook function in your application.

For a detailed description of each function refer chapter 4.4.

3.6 TP classes Dynamic MultiTP

In opposite to the static MultiTP there are no fix connections available. All connections are built-on during runtime and released after the transmission is complete. So the resources used per connection can be reused by other applications.

3.6.1 Properties

Tx channel count

Maximum possible number of parallel used TpChannel(s) for transmissions.

Rx channel count

Maximum possible number of parallel used TpChannel(s) for receptions.

Use Tx channels without FC

Enable the feature to use the non-ISO implementation ‘without FC’ for transmission.

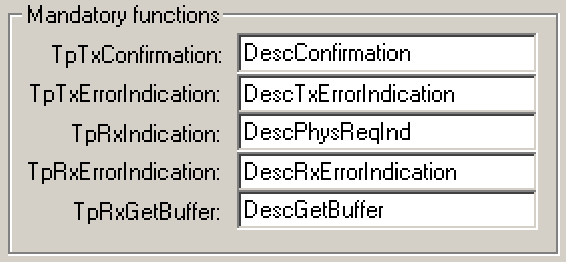
Use Rx channels without FC

Enable the feature to use the non-ISO implementation ‘without FC’ for reception.

3.6.2 Hook Functions

In opposite to the static MultiTP, where all hook functions are available once for each statically configured connection, here this set of hook functions is available only once for all connections. This means that all messages have to be dispatched to the belonging destination by the application for each connection.

These hook functions we recommend to use.



Mandatory functions

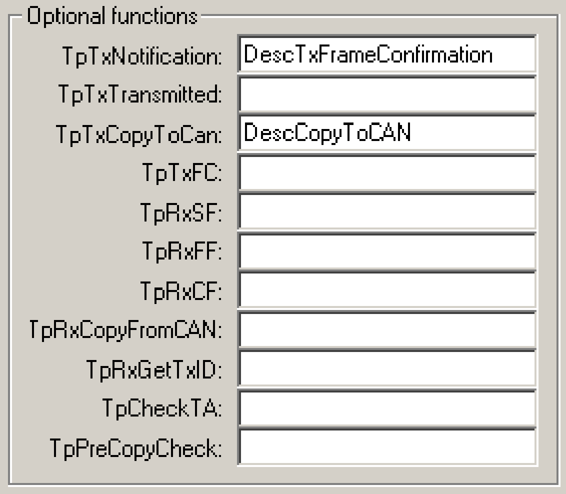
TpTxConfirmation: DescConfirmation
TpTxErrorIndication: DescTxErrorIndication
TpRxIndication: DescPhysReqInd
TpRxErrorIndication: DescRxErrorIndication
TpRxGetBuffer: DescGetBuffer

Figure 3-11 Mandatory functions for the usage of the CANdesc diagnostic component

Just enter a suitable function name to use the hook function in your application.

For a detailed description of each function refer to chapter 4.4.

These hook functions are optional.



Optional functions

TpTxNotification: DescTxFrameConfirmation
TpTxTransmitted:
TpTxCopyToCan: DescCopyToCAN
TpTxFC:
TpRxSF:
TpRxFF:
TpRxCF:
TpRxCopyFromCAN:
TpRxGetTxID:
TpCheckTA:
TpPreCopyCheck:

Figure 3-12 Optional functions (example for the usage of the CANdesc diagnostic component)

Be careful
while using a Vector Diagnostic Layer it is necessary to hand over only the function calls to the Diagnostic Layer, which belong to the diagnostic connection(s). An application example is present, see chapter 8.5.1.

3.6.3 Dynamic Objects

The MultiConnection Tp uses the “dynamic TxID” functionality (Dynamic TxID → On) of the CAN-Driver. However, you can specify additional dynamic objects for your application.

⚠	**Important** If you want to add dynamic objects for your application you have just to enter your count of dynamic objects. The Generation Tool adds the usage of dynamic objects for the MultiConnection Tp automatically.

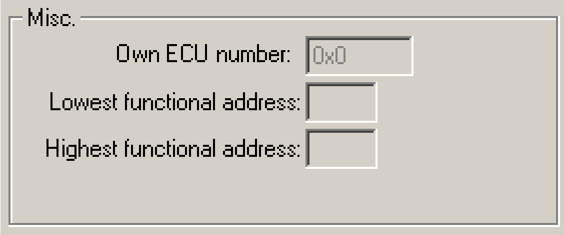
3.6.4 TP class Dynamic MultiTP: Normal Addressing

3.6.4.1 CANdriver settings

⚠	**Important** Actually the Generation Tool will not setup the reception messages automatically. The user itself has to insert for each message, which should be processed by the TP (or for a range of messages) a 'TpPrecopy'-function. Please refer the CAN-driver manual /CANdrv/ how to insert a Precopy-function.

3.6.5 TP class Dynamic MultiTP: Extended Addressing

3.6.5.1 TP class specific settings



Misc.

Own ECU number: 0x0

Lowest functional address:

Highest functional address:

Figure 3-13 Misc (Extended Addressing)

Own ECU number

It will be read out from the database attribute 'TpOwnSystemEcuNumber'.

Lowest functional address

The value should define the lowest value of an additional range of receivable TargetAddresses.

Not supported – use instead the hook function `ApplTpCheckTA()`

Highest functional address

The value should define the highest value of an additional range of receivable TargetAddresses.

Not supported – use instead the hook function `ApplTpCheckTA()`

3.6.5.2 Database Attributes

Name	Default	No TP used	Normal	Extended (example)
TpNodeBaseAddress	FFFF	Default	Default	0x600
TpOwnSystemEcuNumber	FF	Default	Default	0x01
TpNodeMesageCount	FF	Default	Default	0xff

Table 3-2 Data Base Attributes

TpNodeBaseAddress
The not valid value FFFF indicates, that there is no base address necessary.

TpOwnSystemEcuNumber
This value provides the own ECU Number, necessary for setting up the transmit identifier.

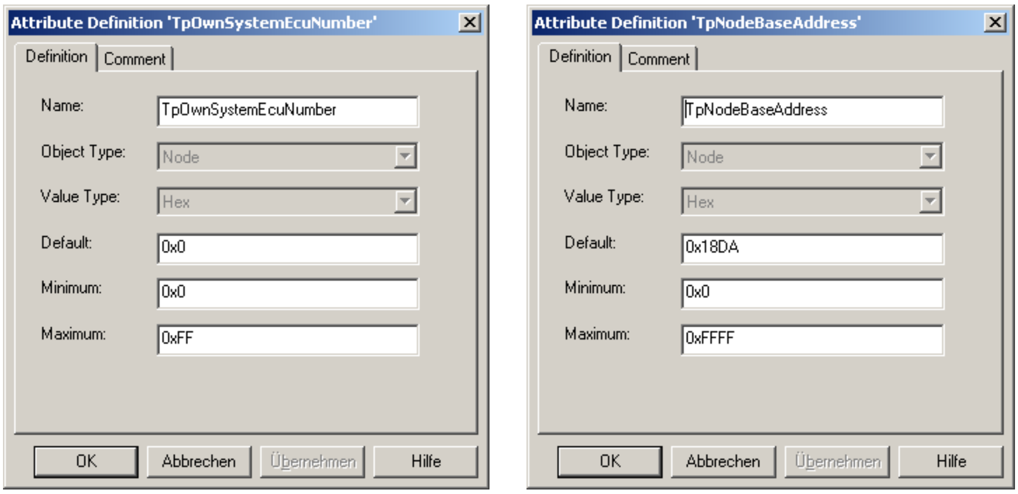
TpNodeMessageCount
This value determines how many messages are assigned to the ‘range’ together with the base address. This is necessary for the TP to calculate to which base the received CAN ID is assigned.
The values for extended addressing are just an example:
The CAN ID for this node is 0x600 + 0x01 = 0x601.

3.6.5.3 Multiple Base Addresses

For each connection a dedicated base address including an address offset and a message count can be specified.

3.6.6 TP class Dynamic MultiTP: Normal Fixed Addressing

3.6.6.1 Database Attributes



Attribute Definition 'TpOwnSystemEcuNumber'

Definition | Comment

Name: TpOwnSystemEcuNumber
Object Type: Node
Value Type: Hex
Default: 0x0
Minimum: 0x0
Maximum: 0xFF

OK | Abbrechen | Übernehmen | Hilfe

Attribute Definition 'TpNodeBaseAddress'

Definition | Comment

Name: TpNodeBaseAddress
Object Type: Node
Value Type: Hex
Default: 0x18DA
Minimum: 0x0
Maximum: 0xFFFF

OK | Abbrechen | Übernehmen | Hilfe

Figure 3-14 Database attributes for ‘Normal Fixed Addressing’

TpOwnSystemEcuNumber

Each ECU is represented in the network by an address / EcuNumber. If the EcuNumber 0xff is used the TP activates the ‘Multiple EcuNumber’ feature (refer 7.4.1 Virtual ECU’s).

TpNodeBaseAddress

This attribute includes the upper 13 bits (like priority, PGN) of the CAN-ID.

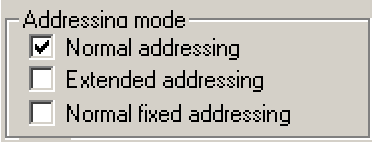
3.6.7 TP class Dynamic MultiTP: Mixed 29-bit Addressing

Currently open – support is only for generation tool GENy requested

3.6.8 TP class Dynamic MultiTP: Multiple Addressing

In this TP class it is possible to change the addressing mode in run-time.

3.6.8.1 Addressing mode



Addressing mode
☑ Normal addressing
☐ Extended addressing
☐ Normal fixed addressing

Figure 3-15 Addressing mode (Multiple Addressing)

Only the checked addressing modes will be supported.

3.6.8.2 CAN Driver settings

To distinguish the addressing mode while the reception different Precopy-functions will exist for each mode. It is possible to insert the Precopy-function for a message or for a range of messages (CAN-Driver Ranges).

- **NormalAddressing**: TpPrecopyNormal<DESTINATION>
- **NormalFixedAddressing**: TpPrecopyNormalFixed<DESTINATION>
- **ExtendedAddressing**: TpPrecopyExtended<DESTINATION>
- **Mixed29Addressing**: TpPrecopyMixed29<DESTINATION>
- **Mixed11Addressing**: TpPrecopyMixed11<DESTINATION>

Caution
Actually the Generation Tool will not setup the reception messages automatically.

<DESTINATION> is replaced by on of the following strings:

- Appl
- DiagFunc
- DiagPhys

These destinations identify the purpose of a given connection. DiagFunc will identify a functional Diagnostic message (1:n). DiagPhys is representing the standard physical diagnostic message (1:1) and Appl a standard TPMC connection used for application purpose (1:1).

E.g.: NormalFixedAddressing range 18DA0500 with mask 0xFF which is specified by the ISO standard as physical range would be configured in the CAN Driver as:

TpPrecopyNormalFixedDiagPhys

Using a dispatcher in combination with two macro functions it is possible to distinguish inside the TPMC callback function set between a diagnostic or applicational request message and direct it to the correct component like CANdesc.

`TpRxGetAddressingFormat(tpChannel)` can be used to check against

```
#define kTpNormalAddressing
#define kTpExtendedAddressing
#define kTpNormalFixedAddressing
#define kTpMixed29Addressing
#define kTpMixed11Addressing
```

`TpRxGetAssignedDestination( tpChannel)` can be used to check against

```
#define kTpRequestAppl
#define kTpRequestDiagFunctional
#define kTpRequestDiagPhysical
```

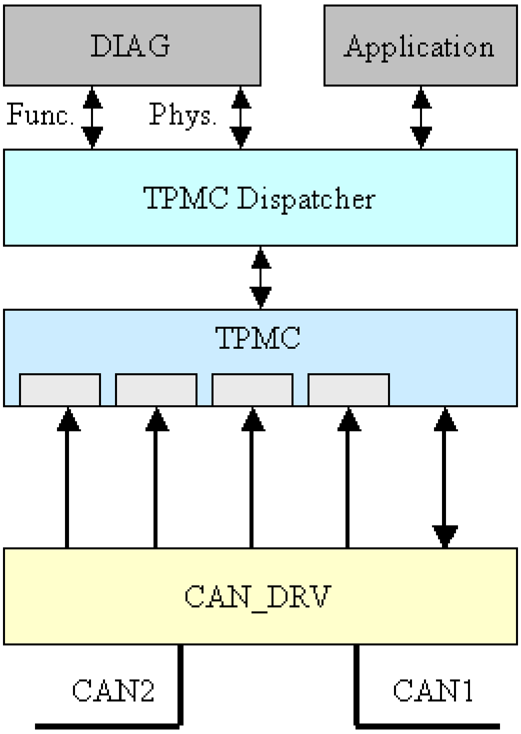


Figure 3-16 Dedicated call of Precopy functions in TPMC by the driver.

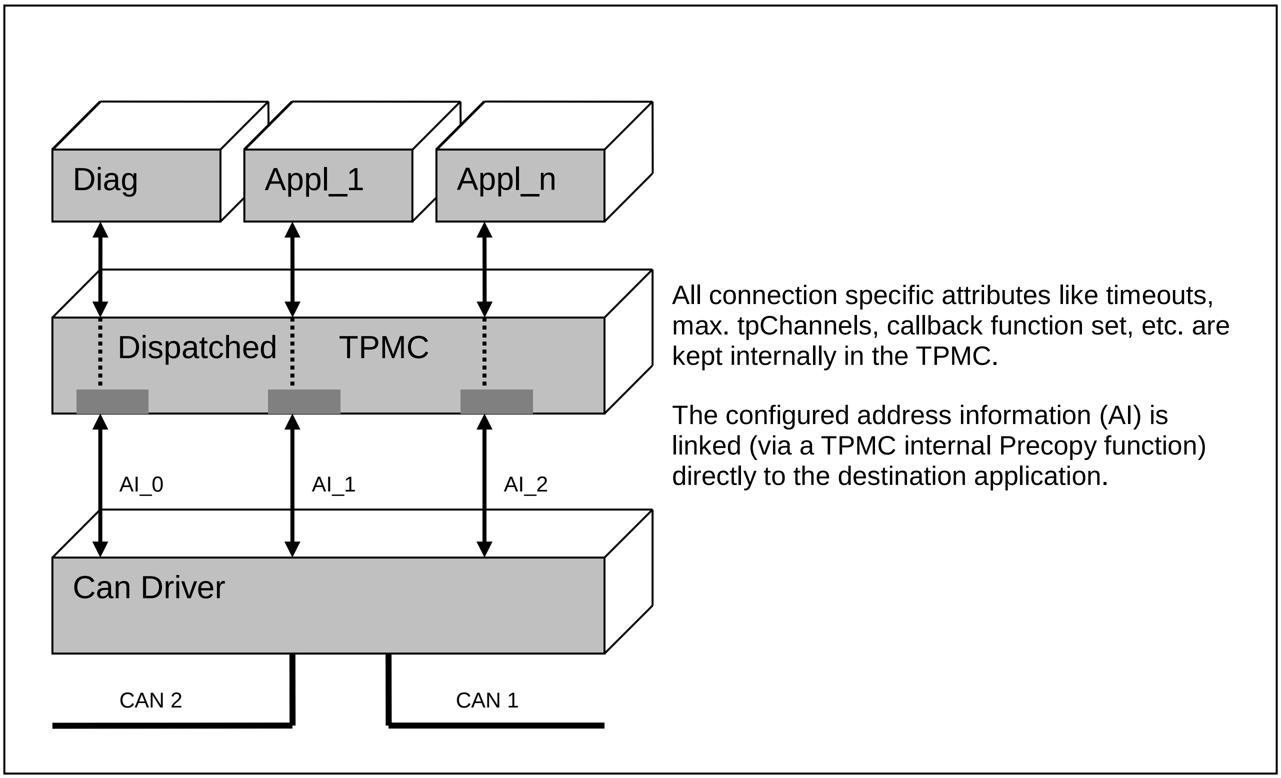
3.7 TP class Dispatched MultiTP

With the release version 3.00.00 of TPMC the “Dispatched” MultiTP class was introduced to disburden the application from the dispatching job.

Using the “Dynamic MultiTP” classes, which support only one single set of callback functions for all connections together, the dispatching of the actual destination has to be performed by the application.

Using the “Dispatched MultiTP” classes all of the dispatching work is done within the TPMC.

“Dispatched MultiTP” is located between static and dynamic TP classes. As well as Static TP it supports connection specific sets of callback functions and dispatches all connections, regarding the Address Information (AI), to these callback functions. Just as Dynamic TP resources are shared among the connections.



All connection specific attributes like timeouts, max. tpChannels, callback function set, etc. are kept internally in the TPMC.

The configured address information (AI) is linked (via a TPMC internal Precopy function) directly to the destination application.

Figure 3-17 Dedicated call of application callback functions in TPMC by the internal dispatcher.

Info
Please note that all existing applications are unaffected unless the new class is actually selected in the generation tool.

3.7.1 "Dynamic MultiTP" versus "Dispatched MultiTP" – a short analogy

3.7.1.1 Solution based on "Dynamic MultiTP":

Here all dynamic TpChannels are provided as a global resource and shared by all connections. So, if no Rx channel is currently available then the incoming message is simply discarded by the TPMC, no reception will occur and the application will not be notified. Otherwise the primal callback function to map an incoming request to a connection, the 'ApplTpRxGetBuffer' function, is called. The addressing data statically configured in GENy is not present for the dispatching application. There is no consistency provided by the TPMC.

To perform this mapping the addressing information statically configured has to be compared to the currently received CAN message. The scope of the addressing information to be compared can be different and depends on the used addressing type.

If a valid connection is found within the 'ApplTpRxGetBuffer' function then a valid pointer to the application buffer is handed to the TPMC, the FC status can be set and the FC addressing information must be set for usage by the TPMC. The identified reception is marked while using the 'TpRxSetConnectionNumber' API function with a unique number defined by the application. To distinguish the connections in later callbacks (e.g. ApplTpRxIndication(tpChannel)), the API TpRxGetConnectionNumber(tpChannel) must be used to get an application relevant handle. The tpChannel handle can and will be different for each reception.

Receive Example: (see also chapter 8.5)

```
/* get CAN-Id */
requestId = TpRxGetChannelID(channel);
if(requestId == MY_RECEIVE_ID) {
  /* store connection number */
  TpRxSetConnectionNumber(channel, kMyConnectionNo);
  /* set CAN-Id for response */
  TpRxSetTransmitID(channel, MY_TRANSMIT_ID);
  pBuf = myTpGetRxBuffer(channel, dataLength);
  /* handle FC status properly */
  if(pBuf == V_NULL) {
    TpRxSetFCStatus(channel, kTpFCStatusOverflow);
  }
  else {
    TpRxSetFCStatus(channel, kTpFCClearToSend);
  }
}
```

For the transmission a Tx channel has to be allocated, a connection number has to be assigned and the connection parameters have to be set according to the addressing type before the transmission can be started.

Transmit Example: (see also chapter 8.5)

```
/* acquire a tx channel */
vuint8 channel = TpTxGetFreeChannel(kMyConnection0);
if(channel != kTpNoChannel ) {
  /* set CAN channel */
```

```
    TpTxSetCanChannel(channel, kMyCanNo);
    /* set CAN identifiers */
    TpTxSetChannelID(channel, myTxCANId, myRxCANId); /* precalculated CAN Ids */
    TpTxSetTargetAddress(channel, target_address);   /* extended addressing   */
    /* trigger the transmission */
    TpTransmit(channel, data, length);
  }
```

For all this topics several API functions must be used in a correct manner what may result in a pretty complex dispatcher to be handled by the application.

3.7.1.2 Solution based on “Dispatched MultiTP”

Each connection group has a configurable number of TpChannels reserved for its own. This offers an improved availability for concurrent receptions with no interference to other TpChannel resources availability.

All Tp callbacks are dispatched internally in the TPMC. In addition to the passing of a raw tpChannel a connection handle ‘addrInfoHandle’ is handed to the application. Behind this ‘addrInfoHandle’ all address information is available based on the static configuration information. Only dynamic runtime address information (e.g. target address in case of Extended- or NormalFixed- addressing) has to be handled extra.

Info

Please note that all application callback functions do not change their API. Additional API functions are provided to get the ‘addressInfoHandle’ from the corresponding tpChannel :

□ `TpRxGetAddressInfoHandle(tpChannel)`: within reception callbacks

□ `TpTxGetAddressInfoHandle(tpChannel)`: within transmission callbacks

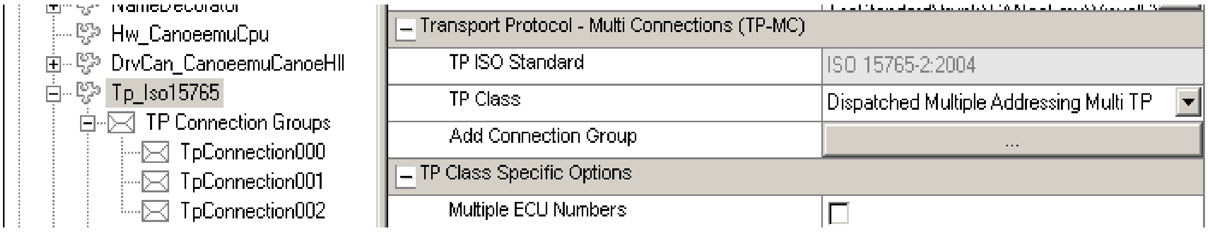
A connection specific precopy function is introduced which is called when the dispatching is already completed and resulted in exactly the call of this connection specific function. To identify the connection later on just the ‘addressInfoHandle’ has to be stored by the application.

The handles are provided in the form “kTp<Addressing Info Name>” in the generated code. So the application can easily differentiate within the callback functions which connection is present just by checking the ‘addressInfoHandle’ using the API ‘TpRxGetAddressInfoHandle()’. Please note that the differentiation in the callback functions is only necessary if more than one AI is configured for one connection or if the same callback functions are configured for more than one connection. Otherwise the corresponding callback function is dedicated unambiguously to one connection.

Of course also here free TpChannels must be available (per connection group) or the reception (transmission) will fail.

Example:

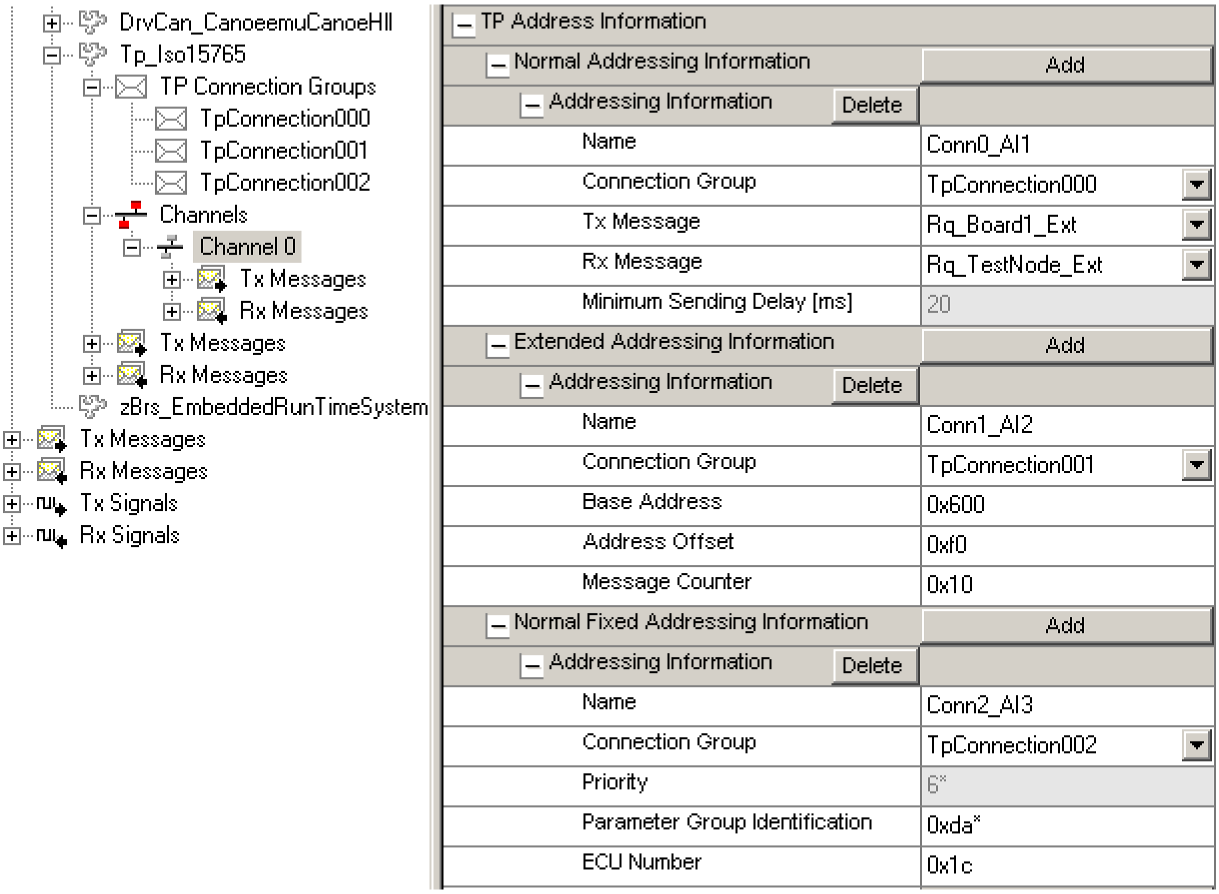
The following example shows a “Dispatched Multiple Addressing Multi TP” configuration containing 3 connections (TpConnection000/001/002).



- NameDecorator
- Hw_CanoeemuCpu
- DrvCan_CanoeemuCanoeHll
- Tp_Iso15765
  - TP Connection Groups
    - TpConnection000
    - TpConnection001
    - TpConnection002

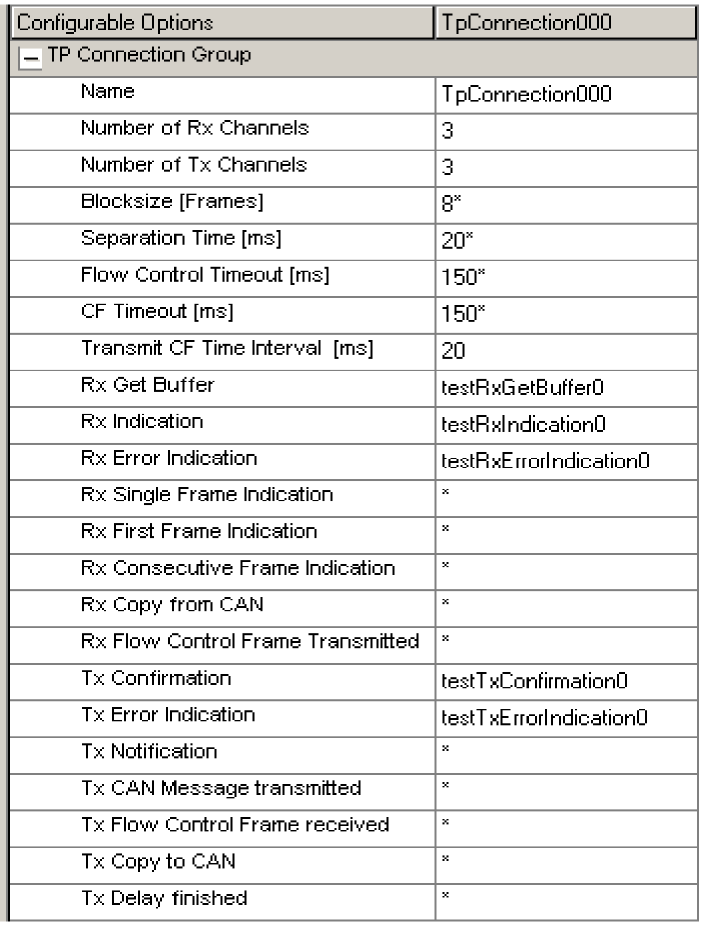
Transport Protocol - Multi Connections (TP-MC)	
TP ISO Standard	ISO 15765-2:2004
TP Class	Dispatched Multiple Addressing Multi TP
Add Connection Group	...
TP Class Specific Options	
Multiple ECU Numbers	□

One AI is configured per connection and each connection uses a different addressing type (Normal-, Extended-, NormalFixed- addressing).



- DrvCan_CanoeemuCanoeHll
- Tp_Iso15765
  - TP Connection Groups
    - TpConnection000
    - TpConnection001
    - TpConnection002
  - Channels
    - Channel 0
      - Tx Messages
      - Rx Messages
  - Tx Messages
  - Rx Messages
- zBrs_EmbeddedRunTimeSystem
- Tx Messages
- Rx Messages
- Tx Signals
- Rx Signals

TP Address Information	
Normal Addressing Information	Add
Addressing Information (Delete)	
Name	Conn0_AI1
Connection Group	TpConnection000
Tx Message	Rq_Board1_Ext
Rx Message	Rq_TestNode_Ext
Minimum Sending Delay [ms]	20
Extended Addressing Information	Add
Addressing Information (Delete)	
Name	Conn1_AI2
Connection Group	TpConnection001
Base Address	0x600
Address Offset	0xf0
Message Counter	0x10
Normal Fixed Addressing Information	Add
Addressing Information (Delete)	
Name	Conn2_AI3
Connection Group	TpConnection002
Priority	6*
Parameter Group Identification	0xda*
ECU Number	0x1c



Configurable Options	TpConnection000
TP Connection Group	
Name	TpConnection000
Number of Rx Channels	3
Number of Tx Channels	3
Blocksize [Frames]	8*
Separation Time [ms]	20*
Flow Control Timeout [ms]	150*
CF Timeout [ms]	150*
Transmit CF Time Interval [ms]	20
Rx Get Buffer	testRxGetBuffer0
Rx Indication	testRxIndication0
Rx Error Indication	testRxErrorIndication0
Rx Single Frame Indication	*
Rx First Frame Indication	*
Rx Consecutive Frame Indication	*
Rx Copy from CAN	*
Rx Flow Control Frame Transmitted	*
Tx Confirmation	testTxConfirmation0
Tx Error Indication	testTxErrorIndication0
Tx Notification	*
Tx CAN Message transmitted	*
Tx Flow Control Frame received	*
Tx Copy to CAN	*
Tx Delay finished	*

Each connection has an appropriate connection specific set of callback functions beneath some other connection specific attributes.

In the generated code the following constants are available for usage by the application.

The connections groups:

```
#define kTpGroupTpConnection000           0
#define kTpGroupTpConnection001           1
#define kTpGroupTpConnection002           2
```

The connection handles:

```
#define kTpConn0_AI1                      0
#define kTpConn1_AI2                      1
#define kTpConn2_AI3                      2
```

The connection specific transmit macros:

```
#define TpTransmit_Conn0_AI1( data ,length)          \
                  TpTransmitNormal(     kTpConn0_AI1, data, length)
#define TpTransmit_Conn1_AI2( TA ,data ,length)      \
                  TpTransmitExtended(   kTpConn1_AI2, TA, data, length)
#define TpTransmit_Conn2_AI3( TA ,data ,length)      \
                  TpTransmitNormalFixed(kTpConn2_AI3, TA, data, length)
```

Now the application can easily differentiate within the connection specific callback functions and decide how to proceed:

```
if(TpRxGetAddressInfoHandle(tpChan) == kTpConn1_AI2) {
  ...
    TpTransmit_Conn1_AI2( TA ,data ,length);
  ...
}
```

3.7.2 Dispatched MultiTP API

⚠ **Caution**

To avoid collisions it is prohibited to use API-functions from the application site that are used internally by the TPMC dispatcher.

This means that all API functions marked as “done internally by TP” in the tables below are neither necessary nor available anymore!

3.7.2.1 Reception side

Dynamic MultiTP class	**Dispatched MultiTP class** **Since version 3.00.00**
`TpRxSetConnectionNumber`	`done internally by TP`
`TpRxGetConnectionNumber`	`done internally by TP`
`TpRxGetAddressingFormat` `TpRxGetAssignedDestination`	`done internally by TP`
`TpRxResetChannel`	`available for application usage`
`TpRxSetTransmitID`	
`TpRxGetStatus`	
`TpRxSetBS` `TpRxGetBS`	
`TpRxSetSTMIN` `TpRxGetSTMIN`	
`TpRxGetChannelID`	
`TpRxGetCanChannel`	
`TpRxGetSourceAddress` `TpRxGetReceivedTargetAddress`	
`TpRxGetEcuNumber`	
`TpRxSetBufferOverrun`	
`TpRxGetAddressExtension`	
`TpRxGetCanbuffer`	
`TpRxSetWaitCorrectSN`	
`TpRxSetTimeoutConfirmation` `TpRxSetTimeoutCF`	

`TpRxSetFCStatus`
`TpRxGetFCStatus`
`TpRxSetClearToSend`	

- New API functions for Dispatched classes:

Please find a more detailed description in chapter 4.

`TpGetConnectionGroup(AI_handle)`	Get the connection group (kTpGroup<ConnectionName>)
`TpGetAddressingType (AI_handle)`	Get the addressing type info (only for multiple addressing class): kTpNormalAddressing, kTpExtendedAddressing, kTpNormalFixedAddressing, kTpMixed11Addressing, kTpMixed29Addressing
`TpGetCanChannel(AI_handle)`	Get the pertaining CAN channel (only for multiple CAN channels)
`TpGetRxId( AI_handle)`	Get the Rx CAN-Identifier (only for normal addressing)
`TpGetTxId( AI_handle)`	Get the Tx CAN-Identifier (only for normal addressing)
`TpGetBaseAddress( AI_handle)`	Get the base address (only for extended addressing)
`TpGetAddressOffset( AI_handle)`	Get the address offset pertaining to a base address (only for extended addressing)
`TpGetPriority( AI_handle)`	Get the priority info from a 29-bit CAN identifier (only for NormalFixed or Mixed29 addressing)
`TpGetPGN( AI_handle)`	Get the parameter group identification from a 29-bit CAN identifier (only for NormalFixed or Mixed29 addressing)
`TpGetEcuNumber( AI_handle)`	Get the ECU address (only for NormalFixed or Mixed29 addressing)

3.7.2.2 Transmission side

Info
Please note that the TpTransmit function is the only API that has to be adapted in the application code.

Dynamic MultiTP class	**Dispatched MultiTP class** **Since version 3.00.00**
TpTxSetChannelID	done internally by TP
TpTxSetCanChannel	done internally by TP
TpTxSetTargetAddress	done internally by TP
TpTxSetEcuNumber	done internally by TP
TpTxSetBaseAddress	done internally by TP
TpTxGetFreeChannel	done internally by TP
TpTxSetAddressingFormat	done internally by TP
TpTxGetConnectionNumber	done internally by TP
TpTxGetConnectionStatus	done internally by TP
TpTxSetAddressExtension	done internally by TP
TpTxSetResponse	done internally by TP
TpTxLockChannel	done internally by TP (see note 1.) below)
TpTxUnlockChannel	
TpTransmit	Either you can use the generated connection specific macros: TpTransmit_<ConnectionName>(data,len),

	TpTransmit_<ConnectionName>(TA,data,len), TpTransmit_<ConnectionName>(AE,data,len), TpTransmit_<ConnectionName>(TA,AE,data,len), or directly the referenced functions: TpTransmitNormal (AI, data, len), TpTransmitExtended (AI, TA, data, len), TpTransmitNormalFixed(AI, TA, data, len), TpTransmitMixed11 (AI, AE, data, len), TpTransmitMixed29 (AI, TA, AE, data, len). Please refer to the API description in chapter 4.
TpTxGetDataBuffer	available for application usage
TpTxGetDataIndex	
TpTxResetChannel	
TpTxGetSTminInFrame	
TpTxPrepareSendImmediate	
TpTxSendImmediate	
TpRxSetStrictFlowControl	

[1.)] Note: The Locking and Unlocking of tpChannels is no longer necessary. Due to the possibility to configure a connection with a dedicated exclusive tpChannel the tpChannel resource is ‘locked’ implicitly.

- New API functions for Dispatched classes:
  Please find a more detailed description in chapter 4.

TpTransmitNormal(AI_handle,data,len)	Instead of using the addressing type specific transmit functions we recommend to use the connection specific macros which are generated.
TpTransmitExtended(AI_handle,data,len)	
TpTransmitNormalFixed(AI_handle,data,len)	
TpTransmitMixed11(AI_handle,data,len	
TpTransmitMixed29(AI_handle,data,len	

4 API

4.1 Use of ISO15765-Transport Protocol

Please use the services of the ISO15765 Transport Protocol in your application according to the instructions in this manual.

Please include the tpmc.h definition file in all modules requiring Transport Protocol Services. All available services, the types for the interface, and symbolic constants are defined in this file.

After a power on reset and before any other call of the Transport Protocol the function void `TpInitPowerOn(void)` has to be called. The main program of the Transport Protocol `TpTxTask()` and `TpRxTask()` has to be called periodically by the application.

All other services of the Transport Protocol are called on those points in your application where services are required.

4.2 Functions of the Transport Protocol

Field description of the following tables

Name of the function

Prototype	
SingleConnectionTp	
	`Function prototype for SingleConnectionTP`
MultipeConnectionTP	
	`Function prototype for MultipleConnectionTP`
Parameter	
Name of the parameter	Description of the parameter
Return code	
name	Meaning of the return code
Availability	
The API is included in all versions, except a restriction is given here	
Description	
Explanation of the functionality	
Pre-condition(s)	
Required preconditions before the function can be used.	
Post-condition(s)	
If a state change was done, it will be described here	
Call context	
The restrictions from where the function can be used are described here.	

Please note
Some additional notes
Examples
A short code example is given

4.2.1 Administrative Functions

4.2.1.1 TpInitPowerOn: Initialization

TpInitPowerOn

Prototype	
SingleConnectionTp	
	`void` **`TpInitPowerOn`** `( void )`
MultipeConnectionTP	
	`void` **`TpInitPowerOn`** `( void )`
Parameter	
-	-
Return code	
-	-
Availability	
No restrictions	
Description	
Power On Initialization of the TP. This function has to be called before all other functions of the Transport Protocol once after Power On.	
Pre-condition(s)	
Global interrupts are disabled and CAN-driver with function `CanInitPowerOn()` and are initialized correctly.	
Post-condition(s)	
The Transport Layer is ready for reception after calling `TpInitPowerOn()`.	
Call context	
Background-loop level with global disabled interrupts	
Please note	
Call the `TpInitPowerOn()` before the application wants to reserve own dynamic transmission objects.	
Examples	
-	

4.2.1.2 TpInit: Re-initialization

TpInit

Prototype	
SingleConnectionTp	
	`void **TpInit** ( void )`
MultipeConnectionTP	
	`void **TpInit** ( void )`
Parameter	
-	-
Return code	
-	-
Availability	
No restrictions	
Description	
The Transport Layer is re-initialized after calling `TpInit()`.	
Pre-condition(s)	
`TpInitPowerOn()` was called before. No TP functionality is used at this time.	
Post-condition(s)	
The Transport Layer is re-initialized after calling `TpInit()`.	
Call context	
Background-loop level with global disabled interrupts	
Please note	
-	
Examples	
-	

4.2.1.3 TpTask: Observing timing conditions

TpTask

Prototype	
SingleConnectionTp	
	`void TP_API_CALL_TYPE **TpTask**(void)`
MultipeConnectionTP	
	`void TP_API_CALL_TYPE **TpTask**(void)`
Parameter	
-	-

Return code	
-	-
Availability	
No restrictions	
Description	
Function calls both TpRxTask and TpTxTask in correct order.	
Pre-condition(s)	
`TpInitPowerOn()` was called before.	
Post-condition(s)	
-	
Call context	
Cyclic task base.	
Please note	
-	
Examples	
-	

4.2.1.4 TpCanChannelInit: CAN channel specifiic re-initialization

TpCanChannelInit

Prototype	
SingleConnectionTp	
	`void TP_API_CALL_TYPE` **`TpCanChannelInit`**`(canuint8 canChannel)`
MultipeConnectionTP	
	`void TP_API_CALL_TYPE` **`TpCanChannelInit`**`(canuint8 canChannel)`
Parameter	
`canChannel`	-
Return code	
-	-
Availability	
Since TPMC version 2.41	
Description	
Any reception / transmission on this CAN channel will be stopped. If a connection was running the application will be informed by calling the function `ApplTpXxErrorIndication()`.	
Pre-condition(s)	
TpInitPowerOn() was called before. No TP functionality is used at this time.	

Post-condition(s)
All running TP channels on this CAN channel are re-initialized.
Call context
Background-loop level with global disabled interrupts
Please note
-
Examples
-

4.2.1.5 TpRxTask: time base for reception timeouts

TpRxTask

Prototype	
SingleConnectionTp	
	`void` **`TpRxTask`** `(void)`
MultipeConnectionTP	
	`void` **`TpRxTask`** `(void)`
Parameter	
-	-
Return code	
-	-
Availability	
No restrictions	
Description	
The function `TpRxTask()` has to be called periodically (cycle time $T_{TpRxCallCycle}$) by the application. This function performs all Rx-Tasks of the Transport Layer and monitors the timings.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
Background-loop level or OSEK-osTask with low priority. **Important note:** This function **must not** be called in interrupt context!	
Please note	
-	
Examples	
-	

4.2.1.6 TpTxTask: time base for timeouts/transmission

TpTxTask

Prototype	
SingleConnectionTp	
	`void` **`TpTxTask`**`(void)`
MultipeConnectionTP	
	`void` **`TpTxTask`**`(void)`
Parameter	
-	-
Return code	
-	-
Availability	
No restrictions	
Description	
The function `TpTxTask()` has to be called periodically (cycle time $T_{TpTxCallCycle}$) by the application. This function performs all Tx-Tasks of the Transport Layer and monitors the timings.	
Pre-condition(s)	
The TP is initialized with `TpInitPowerOn()`.	
Post-condition(s)	
-	
Call context	
Background-loop level or OSEK-OSTask with low priority. **Important note:** This function **must not** be called in interrupt context!	
Please note	
-	
Examples	
-	

4.2.1.7 TpRxStateTask: optional transmission retry

TpRxStateTask

Prototype	
SingleConnectionTp	
	void **TpRxStateTask**(void)
MultipeConnectionTP	
	void **TpRxStateTask**(vuint8 tpChannel)
Parameter	
tpChannel	-
Return code	
-	-
Availability	
Since TPMC version 2.35	
Description	
The function TpRxStateTask() can be called optionally by the application. This function performs the link from the Transport Layer to the CAN-Driver.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
Examples	
-	

4.2.1.8 TpRxAllStateTask: optional transmission retry

TpRxAllStateTask

Prototype	
SingleConnectionTp	
	void **TpRxAllStateTask** (void)
MultipeConnectionTP	
	void **TpRxAllStateTask** (void)
Parameter	
–	-

Return code	
-	-
Availability	
Since TPMC version 2.35	
Description	
The function `TpRxAllStateTask()` can be called optionally by the application. This function performs the link from the Transport Layer to the CAN-Driver for all running Rx-connections.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.1.9 TpTxStateTask: optional transmission retry

TpTxStateTask

Prototype	
SingleConnectionTp	
	`void` **`TpTxStateTask`** `(void)`
MultipeConnectionTP	
	`void` **`TpTxStateTask`** `(vuint8 tpChannel)`
Parameter	
`tpChannel`	-
Return code	
-	-
Availability	
Since TPMC version 2.35	
Description	
The function `TpTxStateTask()` can be called optionally by the application. This function performs the link from the Transport Layer to the CAN-Driver.	
Pre-condition(s)	
The TP is initialized with `TpInitPowerOn ()`.	

Post-condition(s)
-
Call context
-
Please note
It is prohibited to call TpTxStateTask () nested.
Examples
-

4.2.1.10 TpTxAllStateTask: optional transmission retry

TpTxAllStateTask

Prototype	
SingleConnectionTp	
	void **TpTxAllStateTask** (void)
MultipeConnectionTP	
	void **TpTxAllStateTask** (void)
Parameter	
tpChannel	-
Return code	
-	-
Availability	
Since TPMC version 2.35	
Description	
The function TpTxAllStateTask() can be called optionally by the application. This function performs the link from the Transport Layer to the CAN-Driver for all running Tx-connections.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn ().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.2 Receive Functions

4.2.2.1 TpRxSetConnectionNumber: Assign a Connection-Number to a channel

TpRxSetConnectionNumber

Prototype	
SingleConnectionTp	
	–
MultipeConnectionTP	
	void **TpRxSetConnectionNumber**(vuint8 tpChannel, void connection)
Parameter	
tpChannel	Underlying tpChannel used for this connection.
connection	Connection number that shall be assigned to this tpChannel.
Return code	
void	-
Availability	
Only for dynamic TP classes	
Description	
This function assigns an application specific connection-number to this tpChannel.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
Use this function only inside the callback function **ApplTpRxGetBuffer()** !	
Please note	
-	
Examples	
-	

4.2.2.2 TpRxGetConnectionNumber: Get the Corresponding Connection-Number

TpRxGetConnectionNumber

Prototype	
SingleConnectionTp	
	–
MultipeConnectionTP	
	vuint8 **TpRxGetConnectionNumber**(vuint8 tpChannel)

Parameter	
`tpChannel`	-
Return code	
`vuint8`	-
Availability	
Only for dynamic TP classes	
Description	
This function returns the connection-number, which is assigned to this `tpChannel`.	
Pre-condition(s)	
The TP is initialized with `TpInitPowerOn()`. This `tpChannel` is not reset and a connection-number was previously assigned to it by the application. (See TpRxSetConnectionNumber())	
Post-condition(s)	
-	
Call context	
-	
Please note	
A correct value will only be returned, if a connection number was set previously in the callback function `ApplTpRxGetBuffer()` with `TpRxSetConnectionNumber()`.	
Examples	
-	

4.2.2.3 TpRxGetAddressingFormat: Get the current addressing type

TpRxGetAddressingFormat

Prototype	
SingleConnectionTp	
	–
MultipeConnectionTP	
	`canbittype` **`TpRxGetAddressingFormat`**`(canuint8 tpChannel)`
Parameter	
`tpChannel`	Underlying TP channel
Return code	

	`One of the following constants (canbittype:3):`
	`#define kTpNormalAddressing`
	`#define kTpExtendedAddressing`
	`#define kTpNormalFixedAddressing`
	`#define kTpMixed29Addressing`
	`#define kTpMixed11Addressing`

Availability
Only for Multiple Addressing TP
Description
This macro is used to retrieve the required addressing information in a multiple addressing environment. Using a dispatcher in combination with the macro function it is possible to distinguish inside the TPMC callback function set between the different addressing types and handle additional pertaining information.
Pre-condition(s)
A TP Channel is successful allocated.
Post-condition(s)
-
Call context
-
Please note
-
Examples
-

4.2.2.4 TpRxGetAssignedDestination: Get the currently assigned destination

TpRxGetAssignedDestination

Prototype	
SingleConnectionTp	
	–
MultipeConnectionTP	
	`canbittype` **`TpRxGetAssignedDestination`**`(canuint8 tpChannel)`
Parameter	
`tpChannel`	Underlying tp channel
Return code	

	One of the following constants (canbittype:3):
	`#define kTpRequestAppl             // Application`
	`#define kTpRequestDiagFunctional   // Functional Diag.`
	`#define kTpRequestDiagPhysical     // Physical Diag.`
	is delivered to differentiate between application, functional or physical diagnostic requests.

Availability	
Only for Multiple Addressing TP	

Description
This macro is used to retrieve the required destination information in a multiple addressing environment. Using a dispatcher in combination with the macro function it is possible to distinguish inside the TPMC callback function set between the different destinations and handle the correct dispatching of the message to the pertaining destination.
Pre-condition(s)
A tpChannel is successful allocated.
Post-condition(s)
-
Call context
-
Please note
-
Examples
-

4.2.2.5 TpRxResetChannel: Free Rx-TpChannel

TpRxResetChannel

Prototype	
SingleConnectionTp	
	`void TP_API_CALL_TYPE` **`TpRxResetChannel`**`(void)`
MultipeConnectionTP	
	`void TP_API_CALL_TYPE` **`TpRxResetChannel`**`(canuint8 tpChannel)`
Parameter	
`tpChannel`	-
Return code	
-	-
Availability	
No restriction	
Description	
Each time a transport-frame was received the used channel is blocked. To receive another transport-frame on this channel the application has to free this channel. This is, in case of an erroneous reception, not required for the TpRxErrorIndication() callback. The function is called within or after the Rx-Indication - callback. If the application calls the reset-function then the application itself is also responsible to handle the reset values inside the application in further processing steps.	

Pre-condition(s)
The TP is initialized with TpInitPowerOn().
Post-condition(s)
-
Call context
Background-loop level or OSEK-OSTask with lower or same priority as TpTasks.
Please note
-
Examples
-

4.2.2.6 TpRxGetStatus: Rx-Channel Status

TpRxGetStatus

Prototype	
SingleConnectionTp	
	`vuint8 TpRxGetStatus(void)`
MultipeConnectionTP	
	`vuint8 TpRxGetStatus(vuint8 channel)`
Parameter	
`channel`	-
Return code	

`vuint8`	kTpChannelInUse = 0x01
	kTpChannelNotInUse =0x00

Availability	
No restriction	
Description	
This function returns the actual status of the Rx-Channel.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
The returned status can have more than two values! The InUse-Flag is always coded in the lowest bit (0x01)	

Examples

Because it is a status-field there are two possibilities for checking if the channel is InUse:

```
if ( TpRxGetStatus(user_channel) != kTpChannelNotInUse )
{
...
or:
if ( TpRxGetStatus(user_channel) & kTpChannelInUse )
{
...
```

4.2.2.7 TpRxSetBS: Setting up BlockSize on Reception Side

TpRxSetBS

Prototype	
SingleConnectionTp	
	`void` **`TpRxSetBS`**`(vuint8 newBlockSize)`
MultipeConnectionTP	
	`void` **`TpRxSetBS`**`(vuint8 channel, vuint8 newBlockSize)`
Parameter	
`newBlockSize`	-
`channel`	
Return code	
–	
Availability	
Extended API-BS must be activated	
Description	
The BlockSize-Value within the FlowControl can be adjusted by this function.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.2.8 TpRxGetBS: Get BlockSize on Reception Side

TpRxGetBS

Prototype	
SingleConnectionTp	
	`vuint8 `**`TpRxGetBS`**`(void)`
MultipeConnectionTP	
	`vuint8 `**`TpRxGetBS`**`(vuint8 channel)`
Parameter	
`channel`	-
Return code	
–	
Availability	
Extended API-BS must be activated	
Description	
The BlockSize-Value within the FlowControl can be read by this function.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.2.9 TpRxSetSTMIN: Setting up STMin time on Reception Side

TpRxSetSTMIN

Prototype	
SingleConnectionTp	
	`void `**`TpRxSetSTMIN`**`(vuint8 newSTMinSize)`
MultipeConnectionTP	
	`void `**`TpRxSetSTMIN`**`(vuint8 channel, vuint8 newSTMinSize)`
Parameter	
`Channel`	-

`newSTMinSize`	
Return code	
–	
Availability	
Extended API-STMIN must be activated	
Description	
The STmin-Value within the FlowControl can be adjusted by this function.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.2.10 TpRxGetSTMIN: Get STMin time on Reception Side

TpRxGetSTMIN

Prototype	
SingleConnectionTp	
	`vuint8 **TpRxGetSTMIN**(void)`
MultipeConnectionTP	
	`vuint8 **TpRxGetSTMIN**(vuint8 channel)`
Parameter	
`Channel`	-
Return code	
–	
Availability	
Extended API-STMIN must be activated	
Description	
The STmin-Value within the FlowControl can be read by this function.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	

Post-condition(s)
-
Call context
-
Please note
-
Examples
-

4.2.2.11 TpRxGetChannelID: Get Received CAN-Id

TpRxGetChannelID

Prototype	
SingleConnectionTp	
	–
MultipeConnectionTP	
	`vuint16 **TpRxGetChannelID**(vuint8 channel)`
Parameter	
`Channel`	-
Return code	
	CAN-ID
Availability	
Only for dynamic TP class: Normal Addressing.	
Description	
This function returns the CAN-Identifier, of the last transport-frame	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.2.12 TpRxGetChannelExtID: Get Received Extended CAN-Id

TpRxGetChannelExtID

Prototype	
SingleConnectionTp	
	–
MultipeConnectionTP	
	`vuint32 **TpRxGetChannelExtID**(vuint8 channel)`
Parameter	
`Channel`	-
Return code	
	Extended CAN-ID
Availability	
For - Dynamic TP class Normal Addressing and - Dispatched Normal Multi TP	
Description	
This function returns the extended CAN-Identifier, of the last transport-frame	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.2.13 TpRxGetCanChannel: Get physical CAN channel

TpRxGetCanChannel

Prototype	
SingleConnectionTp	
	–
MultipeConnectionTP	
	`vuint8 **TpRxGetCanChannel**(vuint8 channel)`
Parameter	
`Channel`	-

Return code	
–	
Availability	
Only multiple CAN-channel systems	
Description	
This function returns the (physical) CAN-channel, through which the last transport-frame has been received.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.2.14 TpRxGetSourceAddress: Get received Source Address

TpRxGetSourceAddress

Prototype	
SingleConnectionTp	
	`vuint8 TpRxGetSourceAddress(void)`
MultipeConnectionTP	
	`vuint8 TpRxGetSourceAddress(vuint8 channel)`
Parameter	
`Channel`	-
Return code	
–	
Availability	
Only for dynamic TP classes: Extended- and Normal Fixed Addressing	
Description	
This function returns the destination address, which has been received in the last transport-frame.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn()	
Post-condition(s)	
-	

Call context
-
Please note
-
Examples
-

4.2.2.15 TpRxGetReceivedTargetAddress: Get received Target Address

TpRxGetReceivedTargetAddress

Prototype	
SingleConnectionTp	
	`vuint8 TpRxGetReceivedTargetAddress(void)`
MultipeConnectionTP	
	`vuint8 TpRxGetReceivedTargetAddress(vuint8 channel)`
Parameter	
`Channel`	
Return code	
`TargetAddress`	
Availability	
Only for TP classes: Extended-, Normal Fixed-, and Mixed addressing with the extended gateway API enabled.	
Description	
This function returns the destination address, which has been received in the last transport-frame. Normally it is only used for gateway applications.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.2.16 TpRxGetEcuNumber: Get ECU Number

TpRxGetEcuNumber

Prototype	
SingleConnectionTp	
	`vuint8 **TpRxGetEcuNumber** (void)`
MultipeConnectionTP	
	`vuint8 **TpRxGetEcuNumber** (vuint8 channel)`
Parameter	
`Channel`	-
Return code	
–	
Availability	
Multiple EcuNumber feature must be activated	
Description	
This function returns the ECU Number, which has been received in the last transport-frame.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.2.17 TpRxGetParameterGroupIdentification: Get Identification of PGN

TpRxGetParameterGroupIdentification

Prototype	
SingleConnectionTp	
	~~vuint8 **TpRxGetParameterGroupIdentification** (void)~~
MultipeConnectionTP	
	~~vuint8 **TpRxGetParameterGroupIdentification** (vuint8 channel)~~

Parameter	
Channel	-
Return code	
–	
Availability	
⚠	**Caution** Currently not available. Only for dynamic TP class: Normal Fixed Addressing with extended API
Description	
This function returns the Identification of the Parameter Group, which has been received in the last transport-frame.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.2.18 TpRxSetBufferOverrun: Enable partial acceptance

TpRxSetBufferOverrun

Prototype	
SingleConnectionTp	
	void TP_API_CALL_TYPE **TpRxSetBufferOverrun**(void)
MultipeConnectionTP	
	void TP_API_CALL_TYPE **TpRxSetBufferOverrun**(canuint8 tpChannel)
Parameter	
Channel	-
Return code	
–	
Availability	
Since TPMC version 2.41.00. The buffer overrun feature must be enabled	

Description
A reception can be received without copying the received data. This could be useful if the reception buffer is too small, but the request must be received to reject it by a special response. The data of a Single- or FirstFrame are copied, but no data are copied for ConsecutiveFrames. Due to this a buffer must be provided with at least the maximum length of Single- or FirstFrame.
Pre-condition(s)
Only useful if a FF has been received
Post-condition(s)
-
Call context
Within function `ApplTpRxGetBuffer()`
Please note
-
Examples
-

4.2.2.19 TpRxSetTransmitID: Set transmission CAN-Id

TpRxSetTransmitID

Prototype	
SingleConnectionTp	
	-
MultipeConnectionTP	
	`void TP_API_CALL_TYPE` **`TpRxSetTransmitID`** `(canuint8 tpChannel, canuint16 transmitID)`
Parameter	
`tpChannel`	-
`transmitID`	CAN-ID
Return code	
–	
Availability	
Only TP-class 'Dynamic NormalAddressing MultiTP'	
Description	
While receiving a multiple frame request the TP needs the CAN-ID for the transmission of the FlowControl message. Additionally the Diagnostic/TP will need it to calculate the response transmission (`TpTxSetResponse()`), why it is necessary to set it each time `ApplTpRxGetBuffer()` gets called.	
Pre-condition(s)	
-	
Post-condition(s)	
Response can be calculated automatically by the Function `TpTxSetResponse()`.	

Call context
Within function `ApplTpRxGetBuffer()`
Please note
-
Examples
-

4.2.2.20 TpRxSetTransmitExtID: Set transmission Extended CAN-Id

TpRxSetTransmitExtID

Prototype	
SingleConnectionTp	
	-
MultipeConnectionTP	
	`void TP_API_CALL_TYPE` **`TpRxSetTransmitExtID`** `(canuint8 tpChannel, canuint32 transmitID)`
Parameter	
`tpChannel`	-
`transmitID`	Extended CAN-ID (29 bits)
Return code	
–	
Availability	
Only TP-class ‘Dynamic NormalAddressing MultiTP’ and TP-class ‘Dispatched NormalAddressing MultiTP’	
Description	
While receiving a multiple frame request the TP needs the CAN-ID for the transmission of the FlowControl message. Additionally the Diagnostic/TP will need it to calculate the response transmission (`TpTxSetResponse()`), why it is necessary to set it each time `ApplTpRxGetBuffer()` gets called.	
Pre-condition(s)	
-	
Post-condition(s)	
Response can be calculated automatically by the Function `TpTxSetResponse()`.	
Call context	
Within function `ApplTpRxGetBuffer()`	
Please note	
-	
Examples	
-	

4.2.2.21 TpRxGetChannelIDType: Get the type of the received CAN-Id

TpRxGetChannelIDType

Prototype	
SingleConnectionTp	
	-
MultipeConnectionTP	
	`canuint8 TP_API_CALL_TYPE` **`TpRxGetChannelIDType`** `(canuint8 tpChannel)`
Parameter	
`tpChannel`	-
Return code	
`canuint8`	Either kTpCanIdTypeStd (11-Bit) or kTpCanIdTypeExt (29-Bit).
Availability	
Only TP-class ‘Dynamic NormalAddressing MultiTP’.	
Description	
If mixed CAN-IDs, as well 11-Bit identifiers as also 29-Bit identifiers are used during runtime then this API can be used to get the type of the identifier.	
Pre-condition(s)	
-	
Post-condition(s)	
Response can be calculated automatically by the Function `TpTxSetResponse().`	
Call context	
Within function `ApplTpRxGetBuffer()`	
Please note	
-	
Examples	
-	

4.2.2.22 TpRxGetAddressExtension: Get address extension information

TpRxGetAddressExtension

Prototype	
SingleConnectionTp	

	`canuint8 TP_API_CALL_TYPE`
	`TpRxGetAddressExtension``(void)`

MultipeConnectionTP	

	`canuint8 TP_API_CALL_TYPE`
	`TpRxGetAddressExtension``(canuint8 tpChannel)`

Parameter	
`tpChannel`	-
Return code	
`canuint8`	
Availability	
For mixed 29-bit ID and 11-bit ID addressing	
Description	
This function returns the address extension information from the first byte.	
Pre-condition(s)	
Running reception. Valid after callback function ApplTpRxGetBuffer().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.2.23 TpRxGetCanBuffer: Get CAN buffer pointer

TpRxGetCanBuffer

Prototype	
SingleConnectionTp	
	`CanChipDataPtrTP_API_CALL_TYPE` `**TpRxGetCanBuffer**(void);`
MultipeConnectionTP	
	`CanChipDataPtr TP_API_CALL_TYPE` `**TpRxGetCanbuffer**(canuint8 tpChannel);`
Parameter	
`tpChannel`	-
Return code	
–	
Availability	
Since TPMC version 2.41.00	
Description	
Returns a pointer to the first payload byte of the last received CAN frame in the hardware data buffer	

Pre-condition(s)
Reception must be in progress
Post-condition(s)
-
Call context
-
Please note
-
Examples
-

4.2.2.24 TpRxSetWaitCorrectSN: Force to wait for a correct sequence number

TpRxSetWaitCorrectSN

Prototype	
SingleConnectionTp	
	`void TP_API_CALL_TYPE` **`TpRxSetWaitCorrectSN`** `(tpBool wait);`
MultipeConnectionTP	
	`void TP_API_CALL_TYPE` **`TpRxSetWaitCorrectSN`** `(canuint8 tpChannel, tpBool wait);`
Parameter	
`tpChannel` `wait`	- kTpTrue, kTpFalse
Return code	
–	
Availability	
Since TPMC version 2.73.00. Only for Dynamic TP. The following constant must be defined via a user-config file: `#define TP_ENABLE_DYN_AWAIT_CORRECT_SN`	
Description	
The behaviour of the TPMC component in case of a wrong or missing sequence number can be changed: By default (wait = kTpFalse) the TPMC behaviour is like described in ISO 15765-2. By setting the 'wait' parameter to 'kTpTrue' the behaviour can be changed in the way that TPMC does not re-init the connection, but ignores the current frame and continues waiting for the correct sequence number.	
Pre-condition(s)	
-	
Post-condition(s)	
-	

Call context
Within function `ApplTpRxGetBuffer()`.
Please note
-
Examples
-

4.2.2.25 TpRxSetTimeoutConfirmation: Set CAN confirmation timeout

TpRxSetTimeoutConfirmation

Prototype	
SingleConnectionTp	
MultipeConnectionTP	
	`void TP_API_CALL_TYPE` ***`TpRxSetTimeoutConfirmation`***`(canuint8 tpChannel, tTpEngineTimer time);`
Parameter	
`tpChannel`	-
`time`	In timer ticks. The TpTask cycle time is equivalent to one timer tick.
Return code	
–	
Availability	
Since TPMC version 2.73.00. Only for Dynamic Multi TP. The following constant must be defined via a user-config file: `#define TP_ENABLE_DYN_CHANNEL_TIMING.`	
Description	
The CAN message confirmation timeout value (N_Ar) can be changed dynamical.	
Pre-condition(s)	
A tpChannel is successful allocated.	
Post-condition(s)	
-	
Call context	
Within function `ApplTpRxGetBuffer()`.	
Please note	
-	
Examples	
-	

4.2.2.26 TpRxSetTimeoutCF: Set Consecutive Frame confirmation timeout

TpRxSetTimeoutCF

Prototype	
SingleConnectionTp	
MultipeConnectionTP	
	`void TP_API_CALL_TYPE` ***`TpRxSetTimeoutCF`***`(canuint8 tpChannel, tTpEngineTimer time);`
Parameter	
`tpChannel` `time`	- In timer ticks. The TpTask cycle time is equivalent to one timer tick.
Return code	
–	
Availability	
Since TPMC version 2.73.00. Only for Dynamic Multi TP. The following constant must be defined via a user-config file: `#define TP_ENABLE_DYN_CHANNEL_TIMING.`	
Description	
The CF timeout value (N_Cr) can be changed dynamical.	
Pre-condition(s)	
A tpChannel is successful allocated.	
Post-condition(s)	
-	
Call context	
Within function `ApplTpRxGetBuffer().`	
Please note	
-	
Examples	
-	

4.2.2.27 TpRxSetFCStatus: set up Flow Control on reception side

`TpRxSetFCStatus`

Prototype	
SingleConnectionTp	
	`void TpRxSetFCStatus(canuint8 FCStatus)`
MultipeConnectionTP	

	void TpRxSetFCStatus(canuint8 tpChannel, canuint8 FCStatus)
Parameter	
FCStatus	KTpFCClearToSend kTpFCStatusWait kTpFCSupressFrame kTpFCStatusOverflow
tpChannel	
Return code	
	None
Availability	
Only available with at least one of the following switches defined: #define TP_ENABLE_FC_WAIT #define TP_ENABLE_FC_SUPRESS #define TP_ENABLE_FC_OVERFLOW Each of these defines corresponds to the belonging status.	
Description	
The Flow Control content and also the further behaviour can be adjusted by this function. By default the FC status is set to 'kTpFCClearToSend'. In case of 'kTpFCStatusWait' WaitFrames are sent until an explicit clear to send is initiated with the corresponding API function 'TpRxSetClearToSend()'.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
May only be used within the application callback 'ApplTpRxGetBuffer()'.	
Please note	
-	

4.2.2.28 TpRxGetFCStatus: get the Flow Control setup on reception side

TpRxGetFCStatus

Prototype	
SingleConnectionTp	
	canuint8 TpRxGetFCStatus(void)
MultipeConnectionTP	
	canuint8 TpRxGetFCStatus(canuint8 tpChannel)
Parameter	
tpChannel	

Return code	
canuint8	One of the possible status constants: ○ kTpFCClearToSend, ○ kTpFCStatusWait, ○ kTpFCSupressFrame, ○ kTpFCStatusOverflow

Availability

Only available with at least one of the following switches defined:
`#define TP_ENABLE_FC_WAIT`
`#define TP_ENABLE_FC_SUPRESS`
`#define TP_ENABLE_FC_OVERFLOW`
Each of these defines corresponds to the belonging status.

Description
The Flow Control content and also the further behaviour of the TP component depends on the FC status. With this function the effective FC status can be questioned.

Pre-condition(s)
The TP is initialized with TpInitPowerOn().

Post-condition(s)
-

Call context
May be used in application context.

Please note
-

Examples
-

4.2.2.29 TpRxSetClearToSend: proceed with the transmission after FC wait frames

TpRxSetClearToSend

Prototype	
SingleConnectionTp	
	`void TpRxSetClearToSend(canuint8 *pBuffer)`
MultipeConnectionTP	
	`void TpRxSetClearToSend(canuint8 tpChannel, canuint8 *pBuffer)`

Parameter	
tpChannel	

Return code	
	None

Availability
Only available with the following switch defined: `#define TP_ENABLE_FC_WAIT`
Description
When a request that was delayed previously by sending WaitFrames is now ready for reception, then the reception can be started with this function. The already received data is handed to the application buffer passed as parameter and the transmission of a FC(CTS) is initiated.
Pre-condition(s)
The TP is initialized with TpInitPowerOn(). An activation of the WaitFames with a previous call of 'TpRxSetFCStatus(kTpFCStatusWait)' must have be done and must still be active (the effective FC status delivered by TpRxGetFCStatus() is 'kTpFCStatusWait'.), otherwise this function has no effect.
Post-condition(s)
-
Call context
May be used in application context.
Please note
-
Examples
-

4.2.2.30 TpRxWithoutFC: suppress FC frame usage at the Rx side

`TpRxWithoutFC`

Prototype	
SingleConnectionTp	
MultipeConnectionTP	
	`void TP_API_CALL_TYPE TpRxWithoutFC(canuint8 tpChannel)`
Parameter	
`tpChannel`	
Return code	
	None
Availability	
Only available for dynamic Tp classes and with the following switch set to kTpOn: `#define TP_USE_RX_CHANNEL_WITHOUT_FC  kTpOn`	

Description
If the usage of Flow Control frames on the Rx side shall be avoided then the enabling of this feature can be used to suppress all further FC frames within a distinct reception. In this case the suppression of FC frames must be disabled for each new reception by calling this API function for the belonging tpChannel within the ApplTpRxGetBuffer() callback function. For the reception of Single Frames this aspect is irrelevant.
Pre-condition(s)
The TP is initialized with TpInitPowerOn().
Post-condition(s)
-
Call context
Use this function only inside the callback function **`ApplTpRxGetBuffer()`** !
Please note
-
Examples
-

4.2.2.31 TpRxSetPGN: Set Parameter Group Number

TpRxSetPGN

Prototype	
SingleConnectionTp	
	`void` **`TpRxSetPGN`**`(vuint8 pgn)`
MultipeConnectionTP	
	`void` **`TpRxSetPGN`**`(vuint8 tpChannel, vuint8 pgn)`
Parameter	
`tpChannel`	-
`pgn`	Parameter Group Number to be used
Return code	
–	
Availability	
Only for dynamic TP class Normal Fixed or Mixed-29 addressing.	
Description	
This function sets the Parameter Group Number (bit no. 16 - 23) within an extended 29 bit CAN-Identifer to be used for the re-transmission of Flow Control frames for the current reception channel in case of a multi frame reception.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	

Call context
-
Please note
-
Examples
-

4.2.2.32 TpRxSetPriorityBits: Set Priority, Data Page and Reserved bits

TpRxSetPriorityBits

Prototype	
SingleConnectionTp	

	`void TpRxSetPriorityBits(vuint8 prio,`
	`vuint8 res,`
	`vuint8 dataPage)`

`MultipeConnectionTP`	

	`void TpRxSetPriorityBits(vuint8 tpChannel,`
	`vuint8 prio,`
	`vuint8 res,`
	`vuint8 dataPage)`

Parameter	
`tpChannel`	-
`prio`	Priority bits to be used (3 bits from bit position 26-28)
`res`	Reserved bit to be used (1 bit on bit position 25)
`dataPage`	Data Page bit to be used (1 bit on bit position 24)
Return code	
–	
Availability	
Only for dynamic TP class Normal Fixed or Mixed-29 addressing.	
Description	
This function sets beside the Priority Bits (bit no. 26,27,28) also the bits for the ‘Reserved’ bit position (no. 25) and the ‘Data Page’ bit position (no. 24) within an extended 29 bit CAN-Identifer to be used for the retransmission of Flow Control frames for the current reception channel in case of a multi frame reception.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	

Please note	
-	
Examples	
-	

4.2.3 Transmit Functions

4.2.3.1 TpTxGetFreeChannel: Assign Channel to Connection

TpTxGetFreeChannel

Prototype	
SingleConnectionTp	
	-
MultipeConnectionTP	
	`vuint8 TpTxGetFreeChannel(vuint8 connection)`
Parameter	
`connection`	-
Return code	
`vuint8`	
Availability	
Only for dynamic TP classes.	
Description	
This function returns a free channel handle, if possible. If no channel was free the return value will be `kTpNoChannel`. The Transport Layer assigns the connection-number to the channel. The application has got the possibility to get the connection-number by using the function `TpTxGetConnectionNumber(channel)`.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
Within function `ApplTpRxGetBuffer()`.	
Please note	
The connection-numbers starting at **`0xf0`** are reserved for internal usage.	
Examples	
-	

4.2.3.2 TpTxGetConnectionNumber: Get the assigned Connection-Number

TpTxGetConnectionNumber

Prototype	
SingleConnectionTp	
	-
MultipeConnectionTP	
	`vuint8` **`TpTxGetConnectionNumber`**`(vuint8 channel)`
Parameter	
`channel`	-
Return code	
`vuint8`	
Availability	
Only for dynamic TP classes.	
Description	
This function returns the connection-number which is assigned to this channel. The application has got the possibility to assign the connection-number by using the function `TpTxGetFreeChannel(connectionNumber)`.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.3.3 TpTxGetConnectionStatus: Get the Connection Status

TpTxGetConnectionStatus

Prototype	
SingleConnectionTp	
	-
MultipeConnectionTP	
	`vuint8` **`TpTxGetConnectionStatus`**`(vuint8 connection)`

Parameter	
connection	-
Return code	
vuint8	
Availability	
Only for dynamic TP classes.	
Description	
This function returns the corresponding channel-number if it exits. If no channel is assigned to this connection the return value is kTpNoChannel.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.3.4 TpTxGetTargetAddress: Get the target address used for transmission

TpTxGetTargetAddress

Prototype	
	TpTxGetTargetAddress(canuint8 tpChannel)
Parameter	
tpChannel	
Return code	
canuint8	Target address.
Availability	
Only available for “Dispatched Multi TP” classes and NormalFixed-, Extended- or Mixed- Addressing type. One of the following switches must be defined: #define TP_TYPE_MULTI_DISPATCHED_NORMAL_FIXED_ADDRESSING #define TP_TYPE_MULTI_DISPATCHED_EXTENDED_ADDRESSING #define TP_TYPE_MULTI_DISPATCHED_MIXED_29_ADDRESSING	
Description	
This API function enables the application to appoint confirmations to previously issued transmissions. Without this API the appointment of confirmations with parallel transmissions and Normal Fixed, Mixed or Extended addressing is not possible with “Dispatched Multi TP”.	

Pre-condition(s)
The TP is initialized with TpInitPowerOn().
Post-condition(s)
-
Call context
May be used in application context. Typically used in the application callback functions.
Please note
-
Examples
-

4.2.3.5 TpTxGetDataBuffer: Get the assigned Data Buffer

TpTxGetDataBuffer

Prototype	
SingleConnectionTp	
	`vuint8` **`TpTxGetDataBuffer`** `(void)`
MultipeConnectionTP	
	`vuint8` **`TpTxGetDataBuffer`** `(vuint8 channel)`
Parameter	
`channel`	-
Return code	
`vuint8`	
Availability	
Only for dynamic TP classes.	
Description	
This function returns the pointer to the buffer which is assigned to this channel.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.3.6 TpTxGetDataIndex: Get the assigned Data Index

TpTxGetDataIndex

Prototype	
SingleConnectionTp	
	`vuint8 TpTxGetDataIndex (void)`
MultipeConnectionTP	
	`vuint8 TpTxGetDataIndex (vuint8 channel)`
Parameter	
`channel`	-
Return code	
`vuint8`	
Availability	
No restrictions	
Description	
This function returns the current offset into the buffer which is assigned to this channel.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.3.7 TpTxSetChannelID: Set the CAN Transmit Id

TpTxSetChannelID

Prototype	
SingleConnectionTp	
	–
MultipeConnectionTP	

	`void TpTxSetChannelID(vuint8 channel,`
	`vuint16 transmitID,`
	`vuint16 receiveID)`

Parameter	
Channel	-
transmitID	
receivedID	
Return code	
–	
Availability	
Only for dynamic TP class: Normal Addressing	
Description	
This function sets the transmit CAN-Identifier for the next call of `TpTransmit()`. Also the receive CAN-Identifier (must be unique) to the corresponding FlowControl is set.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.3.8 TpTxSetChannelExtID: Set the CAN Transmit Extended Id

TpTxSetChannelExtID

Prototype	
SingleConnectionTp	
	–
MultipeConnectionTP	

	`void` **`TpTxSetChannelExtID`**`(vuint8 channel,`
	`vuint32 transmitID,`
	`vuint32 receiveID)`

Parameter	
Channel	-
transmitID	
receivedID	
Return code	
–	

Availability
For - Dynamic TP class Normal Addressing and - Dispatched Normal Multi TP
Description
This function sets the transmit extended CAN-Identifier (29 bits) for the next call of `TpTransmit()`. Also the receive extended CAN-Identifier (must be unique) to the corresponding FlowControl is set.
Pre-condition(s)
The TP is initialized with TpInitPowerOn().
Post-condition(s)
-
Call context
-
Please note
-
Examples
-

4.2.3.9 TpTxSetCanChannel: Set physical CAN Channel

TpTxSetCanChannel

Prototype	
SingleConnectionTp	
	`void` **`TpTxSetCanChannel`** `( vuint8 canChannel)`
MultipeConnectionTP	

	`void` **`TpTxSetCanChannel`** `(vuint8 channel,`
	`vuint8 canChannel)`

Parameter	
`Channel` `canChannel`	-
Return code	
–	
Availability	
Only for multiple CAN-channel systems and dynamic TP class.	
Description	
This function sets the (physical) CAN-channel for the next call of `TpTransmit()`.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	

Post-condition(s)
-
Call context
-
Please note
-
Examples
-

4.2.3.10 TpTxSetTargetAddress: Set Target Address

TpTxSetTargetAddress

Prototype	
SingleConnectionTp	
	`void TpTxSetTargetAddress(vuint8 targetaddress)`
MultipeConnectionTP	

	`void TpTxSetTargetAddress(vuint8 channel,`
	`                          vuint8 targetaddress)`

Parameter	
`Channel` `targetaddress`	-
Return code	
–	
Availability	
Only for dynamic TP classes: Extended- and Normal Fixed Addressing	
Description	
This function sets the destination address for the next call of `TpTransmit()`.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.3.11 TpTxSetEcuNumber: Set ECU Number

TpTxSetEcuNumber

Prototype	
SingleConnectionTp	
	`void TpTxSetEcuNumber(vuint8 ecuNr)`
MultipeConnectionTP	

	`void TpTxSetEcuNumber(vuint8 channel,`
	`vuint8 ecuNr)`

Parameter	
`Channel`	-
`ecuNr`	
Return code	
–	
Availability	
Only for dynamic TP classes: Extended- and Normal Fixed Addressing ‘Multiple EcuNumber’ feature must be activated	
Description	
This function sets the ECU Number for the next call of `TpTransmit()`.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.3.12 TpTxSetBaseAddress: Set Base Address

TpTxSetEcuNumber

Prototype	
SingleConnectionTp	
MultipeConnectionTP	

	`void TpTxSetBaseAddress(vuint8 channel,`
	`vuint8 baseAddress)`

Parameter	
`Channel`	-
`baseAddress`	
Return code	
–	
Availability	
Only for dynamic TP classes: Extended Addressing ‘Multiple EcuNumber’ feature must be activated.	
Description	
This function sets the base address for the next call of `TpTransmit()`.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.3.13 TpTxSetParameterGroupIdentification: Set Identification of PGN

TpTxSetParameterGroupIdentification

Prototype	
SingleConnectionTp	
	–
MultipeConnectionTP	
	~~`void` **`TpTxSetParameterGroupIdentification`**`(vuint8 channel, vuint8 identification)`~~
Parameter	
`Channel`	-
`identification`	
Return code	
–	

Availability	
⚠	**Caution** Currently not available. Only for dynamic TP class: Normal Fixed Addressing with extended API
Description	
This function sets the Identification of the ParameterGroup for the next call of **`TpTransmit()`**.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.3.14 TpTxSetPriority: Set Priority of the CAN-Frame

TpTxSetPriority

Prototype	
SingleConnectionTp	
	–
MultipeConnectionTP	
	~~void **TpTxSetPriority**(vuint8 channel,~~ ~~vuint8 priority)~~
Parameter	
`Channel`	-
`priority`	
Return code	
–	
Availability	
⚠	**Caution** Currently not available. Only for dynamic TP class: Normal Fixed Addressing with extended API
Description	
This function sets the Priority of the CAN-Frame for the next call of `TpTransmit()`.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	

Post-condition(s)
-
Call context
-
Please note
-
Examples
-

4.2.3.15 TpTxSetResponse: Assemble a Response

TpTxSetResponse

Prototype	
SingleConnectionTp	
	–
MultipeConnectionTP	

	`void` **`TpTxSetResponse`**`(vuint8 rxChannel,`
	`vuint8 txChannel)`

Parameter	
`rxChannel`	-
`txChanel`	
Return code	
–	
Availability	
Only for dynamic TP classes.	
Description	
This function assembles a Response based on a received transport-frame for the next call of `TpTransmit().`	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.3.16 TpTransmit: Send a Message

TpTransmit

Prototype	
SingleConnectionTp	

	`vuint8 TpTransmit(vuint8* data,`
	`                 vuint16 count)`

MultipeConnectionTP	

	`vuint8 TpTransmit (vuint8  tpChannel,`
	`                   vuint8* data,`
	`                   vuint16 count)`

Parameter	
`tpChannel`	
`Data`	Pointer to the data buffer that shall be transmitted.
`count`	Number of bytes to be transmitted.
Return code	
`vuint8`	kTpSuccess: No transmission in progress (ready to send) kTpBusy: Transmission in progress kTpFailed: If the data length is zero or the tpChannel is not allocated.
Availability	
No restrictions	
Description	
Send a message. The Transport Layer decides which transmission protocol (SingleFrame with up to 6/7 data bytes depending on the addressing type) is used by checking the given count.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
After a transmission the channel is released, except the channel is explicitly locked. Since version 2.35 the transmission request will be only queued within the context of TpTransmit. The transmission to the bus starts within the TpTxStateTask (TpTxTask) calls. **kTpFailed**: In previous versions (2.34.xx and earlier) it is possible that TpTransmit returns ‘kTpFailed’, because the CANdriver (CanTransmit returns failed) is busy. Starting with version 2.35.00 only dynamic TP-classes return this value in case of wrong attributes/parameters. **kTpBusy**: A transmission is already running or GenMsgDelayTime is not kept. **kTpSuccess**: Successful queued message that will be transmitted with the next task cycle.	
Examples	
-	

4.2.3.17 TpTxLockChannel: Lock Channel

TpTxLockChannel

Prototype	
SingleConnectionTp	
	–
MultipeConnectionTP	
	void **TpTxLockChannel** (vuint8 channel)
Parameter	
channel	-
Return code	
–	
Availability	
Only for dynamic TP classes.	
Description	
If a channel is locked, it will not be released after a transmission.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.3.18 TpTxUnlockChannel: Unlock TX Channel

TpTxUnlockChannel

Prototype	
SingleConnectionTp	
	–
MultipeConnectionTP	
	void **TpTxUnlockChannel** (vuint8 channel)
Parameter	
channel	-

Return code	
–	
Availability	
Only for dynamic TP classes.	
Description	
Unlock the lock of the channel. The channel will be released with the next call of `TpTxResetChannel()` or `TpTransmit()`.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.3.19 TpTxResetChannel: Free TX-Channel

TpTxResetChannel

Prototype	
SingleConnectionTp	
	`void TP_API_CALL_TYPE` **`TpTxResetChannel`**`(void)`
MultipeConnectionTP	
	`void TP_API_CALL_TYPE` **`TpTxResetChannel`**`(canuint8 tpChannel)`
Parameter	
`tpChannel`	-
Return code	
–	
Availability	
No rectrictions	
Description	
The channel will be released by the Transport Layer. At the next call of `TpTxGetFreeChannel()` it can be assigned to another connection.	
Pre-condition(s)	
The TP is initialized with `TpInitPowerOn()`.	

Post-condition(s)
-
Call context
Background-loop level or OSEK-OSTask with lower priority as TpTasks.
Please note
The tpChannel will be released in any case and immediately. If a transmission is in progress the application will be informed by calling the function `ApplTpTxErrorIndication()`.
Examples
-

4.2.3.20 TpTxSetAddressExtension: Set Address Extension information

TpTxSetAddressExtension

Prototype	
SingleConnectionTp	
	`void TP_API_CALL_TYPE` **`TpTxSetAddressExtension`**`(canuint8 addressExtension);`
MultipeConnectionTP	
	`void TP_API_CALL_TYPE` **`TpTxSetAddressExtension`**`(canuint8 tpChannel, canuint8 addressExtension);`
Parameter	
`adressExtension`	-
`tpChannel`	
Return code	
–	
Availability	
For mixed 29-bit ID and mixed 11-bit ID addressing	
Description	
This function is used to set the address extension information.	
Pre-condition(s)	
This function must be called in advance of calling TpTransmit().	
Post-condition(s)	
-	
Call context	
-	

Please note
-
Examples
-

4.2.3.21 TpTxGetSTminInFrame: Get STmin from FC frame

TpTxGetSTminInFrame

Prototype	
SingleConnectionTp	

	`canuint8 TP_API_CALL_TYPE`
	`TpTxGetSTminInFrame``(void)`

MultipeConnectionTP	

	`canuint8 TP_API_CALL_TYPE`
	`TpTxGetSTminInFrame``(canuint8 tpChannel)`

Parameter	
`tpChannel`	-
Return code	
`canuint8`	STmin value
Availability	
The STmin value must be taken out of the received FC frames `(TP_USE_STMIN_OF_FC == kTpOn)` and the fast transmission feature `(TP_USE_FAST_TX_TRANSMISSION == kTpOn)` must be activated.	
Description	
Function is returning the STmin value of the last FC frame.	
Pre-condition(s)	
This function must be called in advance of calling TpTransmit().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.3.22 TpTxPrepareSendImmediate: Prepare CF transmission by application

TpTxPrepareSendImmediate

Prototype	
SingleConnectionTp	
	`TP_EXTERNAL_INLINE canuint8 TP_API_CALL_TYPE` **`TpTxPrepareSendImmediate`**`(void)`
MultipeConnectionTP	
	`TP_EXTERNAL_INLINE canuint8 TP_API_CALL_TYPE` **`TpTxPrepareSendImmediate`**`(canuint8 tpChannel)`
Parameter	
`tpChannel`	-
Return code	
`canuint8`	kTpSuccess, kTpFailed
Availability	
The fast transmission feature `(TP_USE_FAST_TX_TRANSMISSION)` must be set to kTpOn.	
Description	
If the TP is not in the state for preparing a new CF-Frame (i.e. it is waiting for a FC) the function will return a ‘kTpFailed’. Otherwise if the preparation is successful it will return a ‘kTpSuccess’. **Note**: In the case of ‘kTpSuccess’ the application is responsible for the transmission of the next ConsecutiveFrame. If the application does not call TpTxSendImmediate() the TP stays blocked.	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
The call of this function is only allowed in the context of the TpTxCanMessageTransmitted() / ApplTpTxFC() Hook-function.	
Please note	
-	
Examples	
-	

4.2.3.23 TpTxSendImmediate: Start CF transmission by application

TpTxSendImmediate

Prototype	
SingleConnectionTp	
	`TP_EXTERNAL_INLINE void TP_API_CALL_TYPE` **`TpTxSendImmediate`**`(void)`
MultipeConnectionTP	

	TP_EXTERNAL_INLINE void TP_API_CALL_TYPE **TpTxSendImmediate**(canuint8 tpChannel)
Parameter	
	-
Return code	
Availability	
The fast transmission feature (TP_USE_FAST_TX_TRANSMISSION) must be set to kTpOn.	
Description	
Prepares the ConsecutiveFrame and calls the TpTxStateTask() to transmit the frame.	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.3.24 TpTxSetAddressingFormat: Store the current addressing type

TpTxSetAddressingFormat

Prototype	
MultipeConnectionTP	
	void TP_API_CALL_TYPE **TpTxSetAddressingFormat**(canuint8 tpChannel, SupportInfoStruct supportInfo)
Parameter	
tpChannel	-
supportInfo	
Return code	
	–
Availability	
Multiple Addressing TP	

Description

This function is used to prepare the required addressing information in a multiple addressing environment and internally to assign a given connection to the right component.

```
#define kTpNormalAddressing
#define kTpExtendedAddressing
#define kTpNormalFixedAddressing
#define kTpMixed29Addressing
#define kTpMixed11Addressing

#define kTpRequestAppl            // Application connection
#define kTpRequestDiagFunctional  // Functional Diag connect.
#define kTpRequestDiagPhysical    // Physical Diag connection

SupportInfoStruct supportInfo;

supportInfo.addressingFormat     = kTpNormalAddressing;

supportInfo.assignedDestination  = kTpRequestDiagPhysical;

TpTxSetAddressingFormat(DiagPhysChannel, supportInfo);
```

Pre-condition(s)

A tpChannel is successful allocated.

Post-condition(s)

-

Call context

-

Please note

-

Examples

-

4.2.3.25 TpTxSetStrictFlowControl: Enable/Disable ISO conformant FC handling

TpTxSetStrictFlowControl

Prototype	
SingleConnectionTp	
	void TP_API_CALL_TYPE **TpTxSetStrictFlowControl** (tpBool strict)
MultipeConnectionTP	
	void TP_API_CALL_TYPE **TpRxSetStrictFlowControl** (canuint8 tpChannel, tpBool strict)
Parameter	
tpChannel	-
strict	kTpTrue, kTpFalse
Return code	
	–

Availability
Since TPMC version 2.73.00. Only for Dynamic TP. The following constant must be defined via a user-config file: `#define TP_ENABLE_FC_MSG_FLOW_DYN_CHECK.`
Description
The behaviour of the TPMC component in case of a missing FC frame can be changed: By default (strict = kTpTrue) the TPMC behaviour is like described in ISO 15765-2. By setting the ‘strict’ parameter to ‘kTpFalse’ the behaviour can be changed in the way that TPMC does not re-init the connection, but ignores the current frame in case of a missing FC.
Pre-condition(s)
A tpChannel is successful allocated.
Post-condition(s)
-
Call context
Call before `TpTransmit()`
Please note
-
Examples
-

4.2.3.26 TpTxSetTimeoutConfirmation: Set the CAN confirmation timeout

TpTxSetTimeoutConfirmation

Prototype	
SingleConnectionTp	
MultipeConnectionTP	
	`void TP_API_CALL_TYPE` ***`TpTxSetTimeoutConfirmation`***`(canuint8 tpChannel, tTpEngineTimer time)`
Parameter	
`tpChannel`	-
`Time`	In timer ticks. The TpTask cycle time is equivalent to one timer tick.
Return code	
	–

Availability
Since TPMC version 2.73.00. Only for Dynamic Multi TP. The following constant must be defined via a user-config file: `#define TP_ENABLE_DYN_CHANNEL_TIMING.`
Description
The CAN message confirmation timeout (N_As) value can be changed dynamical.
Pre-condition(s)
A tpChannel is successful allocated.
Post-condition(s)
-
Call context
Call before TpTransmit().
Please note
-
Examples
-

4.2.3.27 TpTxSetTimeoutFC: Set the FC confirmation timeout

TpTxSetTimeoutFC

Prototype	
SingleConnectionTp	
MultipeConnectionTP	
	`void TP_API_CALL_TYPE` ***`TpTxSetTimeoutFC`***`(canuint8 tpChannel, tTpEngineTimer time)`
Parameter	
`tpChannel`	-
`Time`	In timer ticks. The TpTask cycle time is equivalent to one timer tick.
Return code	
	–
Availability	
Since TPMC version 2.73.00. Only for Dynamic Multi TP. The following constant must be defined via a user-config file: `#define TP_ENABLE_DYN_CHANNEL_TIMING.`	
Description	
The FC timeout value (N_Bs) can be changed dynamical per channel.	

Pre-condition(s)
A tpChannel is successful allocated.
Post-condition(s)
-
Call context
Call before TpTransmit().
Please note
-
Examples
-

4.2.3.28 TpTxWithoutFC: suppress FC frame usage at the Tx side

`TpTxWithoutFC`

Prototype	
SingleConnectionTp	
MultipeConnectionTP	
	`void TP_API_CALL_TYPE TpTxWithoutFC(canuint8 tpChannel)`
Parameter	
`tpChannel`	
Return code	
	None
Availability	
Only available for dynamic Tp classes and with the following switch set to kTpOn: `#define TP_USE_TX_CHANNEL_WITHOUT_FC  kTpOn`	
Description	
If the usage of Flow Control frames on the Tx side shall be avoided then the enabling of this feature can be used to suppress all further FC frames within a distinct transmission. In this case the suppression of FC frames must be disabled for each new transmission by calling this API function for the belonging tpChannel before calling TpTransmit. For the transmission of Single Frames this aspect is irrelevant.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
Call from task context before calling TpTransmit.	

Please note
-
Examples
-

4.2.3.29 TpTxSetPGN: Set Parameter Group Number

TpTxSetPGN

Prototype	
SingleConnectionTp	
	`void **TpTxSetPGN**(vuint8 pgn)`
MultipeConnectionTP	
	`void **TpTxSetPGN**(vuint8 tpChannel, vuint8 pgn)`
Parameter	
tpChannel	-
pgn	Parameter Group Number to be used
Return code	
–	
Availability	
Only for dynamic TP class Normal Fixed or Mixed-29 addressing.	
Description	
This function sets the parameter group number (bit no. 16 - 23) within an extended 29 bit CAN-Identifer for the next call of **`TpTransmit()`**.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.2.3.30 TpTxSetPriorityBits: Set Priority, Data Page and Reserved bits

TpTxSetPriorityBits

Prototype	
SingleConnectionTp	

	`void TpTxSetPriorityBits(vuint8 prio,`
	`vuint8 res,`
	`vuint8 dataPage)`

`MultipeConnectionTP`	

	`void TpTxSetPriorityBits(vuint8 tpChannel,`
	`vuint8 prio,`
	`vuint8 res,`
	`vuint8 dataPage)`

Parameter	
tpChannel	-
prio	Priority bits to be used (3 bits from bit position 26-28)
res	Reserved bit to be used (1 bit on bit position 25)
dataPage	Data Page bit to be used (1 bit on bit position 24)
Return code	
–	
Availability	
Only for dynamic TP class Normal Fixed or Mixed-29 addressing.	
Description	
This function sets beside the Priority Bits (bit no. 26,27,28) also the bits for the ‘Reserved’ bit position (no. 25) and the ‘Data Page’ bit position (no. 24) within an extended 29 bit CAN-Identifer for the next call of **`TpTransmit()`**.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.3 Dispatched Multi TP class API

4.3.1 TpGetConnectionGroup: Get the connection group identification

TpGetConnectionGroup

Prototype	
	`TpGetConnectionGroup(canuint8 addressInfoHandle)`
Parameter	
`addressInfoHandle`	
Return code	
`canuint8`	kTpGroup<ConnectionName> constant
Availability	

Only available for “Dispatched Multi TP” classes.
One of the following switches must be defined:
`#define TP_TYPE_MULTI_DISPATCHED_NORMAL_ADDRESSING`
`#define TP_TYPE_MULTI_DISPATCHED_EXTENDED_ADDRESSING`
`#define TP_TYPE_MULTI_DISPATCHED_NORMAL_FIXED_ADDRESSING`
`#define TP_TYPE_MULTI_DISPATCHED_MIXED_29_ADDRESSING`
`#define TP_TYPE_MULTI_DISPATCHED_MULTIPLE_ADDRESSING`	

Description	
Deliver the appropriate connection group identification as a constant.	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
May be used in application context. Typically used in the application callback functions.	
Please note	
The TP is initialized with TpInitPowerOn().	
Examples	
-	

4.3.2 TpGetAddressingType: Get the addressing type identification

TpGetAddressingType

Prototype	
	`TpGetAddressingType(canuint8 addressInfoHandle)`
Parameter	
`addressInfoHandle`	
Return code	

`canuint8`	One of the possible status constants:
	`kTpNormalAddressing,`
	`kTpExtendedAddressing,`
	`kTpNormalFixedAddressing,`
	`kTpMixed29Addressing`

Availability	
Only available for “Dispatched Multi TP” classes and “Multiple Addressing” type. The following switch must be defined: `#define TP_TYPE_MULTI_DISPATCHED_MULTIPLE_ADDRESSING`	
Description	
Deliver the appropriate addressing type as a constant.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
May be used in application context. Typically used in the application callback functions.	
Please note	
-	
Examples	
-	

4.3.3 TpGetCanChannel: Get the CAN channel

TpGetCanChannel

Prototype	
	TpGetCanChannel(canuint8 addressInfoHandle)
Parameter	
addressInfoHandle	
Return code	
canuint8	Number of the CAN channel.
Availability	
Only available for “Dispatched Multi TP” classes and multiple CAN channels configured. One of the following switches must be defined: #define TP_TYPE_MULTI_DISPATCHED_NORMAL_ADDRESSING #define TP_TYPE_MULTI_DISPATCHED_EXTENDED_ADDRESSING #define TP_TYPE_MULTI_DISPATCHED_NORMAL_FIXED_ADDRESSING #define TP_TYPE_MULTI_DISPATCHED_MIXED_29_ADDRESSING #define TP_TYPE_MULTI_DISPATCHED_MULTIPLE_ADDRESSING	
Description	
Deliver the appropriate CAN channel.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
May be used in application context. Typically used in the application callback functions.	
Please note	
-	
Examples	
-	

4.3.4 TpGetRxId: Get the received CAN-Id

TpGetRxId

Prototype	
	`TpGetRxId(canuint8 addressInfoHandle)`
Parameter	
`addressInfoHandle`	
Return code	
`canuint8`	`CAN identifier.`
Availability	
Only available for “Dispatched Multi TP” classes and “Normal Addressing” type. The following switches must be defined: `#define TP_TYPE_MULTI_DISPATCHED_NORMAL_ADDRESSING`	
Description	
Deliver the appropriate Rx CAN identifier.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
May be used in application context. Typically used in the application callback functions.	
Please note	
-	
Examples	
-	

4.3.5 TpGetTxId: Get the CAN-Id to be used for transmission

TpGetTxId

Prototype	
	`TpGetTxId(canuint8 addressInfoHandle)`
Parameter	
`addressInfoHandle`	
Return code	
`canuint8`	`CAN identifier.`

Availability
Only available for “Dispatched Multi TP” classes and “Normal Addressing” type. The following switches must be defined: `#define TP_TYPE_MULTI_DISPATCHED_NORMAL_ADDRESSING`
Description
Deliver the appropriate Tx CAN identifier.
Pre-condition(s)
The TP is initialized with TpInitPowerOn().
Post-condition(s)
-
Call context
May be used in application context. Typically used in the application callback functions.
Please note
-
Examples
-

4.3.6 TpGetBaseAddress: Get the Base Address

TpGetBaseAddress

Prototype	
	`TpGetBaseAddress(canuint8 addressInfoHandle)`
Parameter	
`addressInfoHandle`	
Return code	
`canuint8`	`Base address.`
Availability	
Only available for “Dispatched Multi TP” classes and “Extended Addressing” type. The following switches must be defined: `#define TP_TYPE_MULTI_DISPATCHED_EXTENDED_ADDRESSING`	
Description	
Deliver the appropriate base address.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	

Call context
May be used in application context. Typically used in the application callback functions.
Please note
-
Examples
-

4.3.7 TpGetAddressOffest: Get the Address Offset

TpGetAddressOffset

Prototype	
	`TpGetAddressOffset(canuint8 addressInfoHandle)`
Parameter	
`addressInfoHandle`	
Return code	
`canuint8`	`Address offset.`
Availability	
Only available for “Dispatched Multi TP” classes and “Extended Addressing” type. The following switches must be defined: `#define TP_TYPE_MULTI_DISPATCHED_EXTENDED_ADDRESSING`	
Description	
Deliver the appropriate address offset.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
May be used in application context. Typically used in the application callback functions.	
Please note	
-	
Examples	
The address 0x06F0 is separated in 2 parts: - base address 0x0600 and - address offset 0x00F0	

4.3.8 TpGetPriority: Get the priority info from a 29 bit CAN-Id

TpGetPriority

Prototype	
	`TpGetPriority(canuint8 addressInfoHandle)`
Parameter	
`addressInfoHandle`	
Return code	
`canuint8`	`Priority value (0..7)`
Availability	

Only available for “Dispatched Multi TP” classes and “NormalFixed Addressing” or “Mixed29” addressing type.
The following switches must be defined:
`#define TP_TYPE_MULTI_DISPATCHED_NORMAL_FIXED_ADDRESSING`
`#define TP_TYPE_MULTI_DISPATCHED_MIXED_29_ADDRESSING`	

Description	
Deliver the appropriate address offset.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
May be used in application context. Typically used in the application callback functions.	
Please note	
-	
Examples	
-	

4.3.9 TpGetPGN: Get the parameter group identification from a 29 bit CAN-Id

TpGetPGN

Prototype	
	`TpGetPGN(canuint8 addressInfoHandle)`
Parameter	
`addressInfoHandle`	
Return code	
`canuint8`	PGN value.

Availability

Only available for “Dispatched Multi TP” classes and “NormalFixed Addressing” or “Mixed29” addressing type.
The following switches must be defined:
`#define TP_TYPE_MULTI_DISPATCHED_NORMAL_FIXED_ADDRESSING`
`#define TP_TYPE_MULTI_DISPATCHED_MIXED_29_ADDRESSING`

Description
Deliver the appropriate address offset.
Pre-condition(s)
The TP is initialized with TpInitPowerOn().
Post-condition(s)
-
Call context
May be used in application context. Typically used in the application callback functions.
Please note
-
Examples
-

4.3.10 TpGetEcuNumber: Get the ECU number

TpGetEcuNumber

Prototype	
	`TpGetEcuNumber(canuint8 addressInfoHandle)`
Parameter	
`addressInfoHandle`	
Return code	
`canuint8`	ECU number.
Availability	

Only available for “Dispatched Multi TP” classes and “NormalFixed Addressing” or “Mixed29” addressing type.
The following switches must be defined:
`#define TP_TYPE_MULTI_DISPATCHED_NORMAL_FIXED_ADDRESSING`
`#define TP_TYPE_MULTI_DISPATCHED_MIXED_29_ADDRESSING`	

Description	
Deliver the appropriate ECU number.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	

Post-condition(s)
-
Call context
May be used in application context. Typically used in the application callback functions.
Please note
-
Examples
-

4.3.11 TpTransmit

There are two alternatives available to transmit data. Either you use the generated connection specific TpTransmit macros or you use the addressing type specific functions behind the macros.

4.3.11.1 TpTransmit connection specific macros

The data pointer (type canuint8) and the data length (type canuint16) are always necessary. Depending on the addressing type additional information like the Target Address (TA) for Extended / NormalFixed addressing or the Address Extension (AE) for Mixed addressing is necessary.

Addressing Type	Macro name

`Normal`	`TpTransmit_<ConnectionName>(canuint8  data,`
	`canuint16 len)`

`Extended`
`NormalFixed`
	`TpTransmit_<ConnectionName>(canuint8  TA,`
	`canuint8  data,`
	`canuint16 len)`

`Mixed29`	`TpTransmit_<ConnectionName>(canuint8  TA,`
	`canuint8  AE,`
	`canuint8  data,`
	`canuint8  len)`

4.3.11.2 TpTransmitNormal: transmit function for normal addressing

TpTransmitNormal

Prototype	

	`TpTransmitNormal(canuint8  addressInfoHandle,`
	`canuint8  data,`
	`canuint16 length)`

Parameter	
addressInfoHandle	
data	Pointer to the transmit data.
length	Length of the transmit data (in bytes).
Return code	
canuint8	kTpSuccess: No transmission in progress (ready to send) kTpBusy: Transmission in progress kTpFailed: Data length is zero kTpNoChannel: No TP channel available
Availability	
Only available for “Dispatched Multi TP” classes. The following switch must be defined: #define TP_TYPE_MULTI_DISPATCHED_NORMAL_ADDRESSING	
Description	
Send the data with the given length to the CAN bus.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
May be used in application context.	
Please note	
-	
Examples	
-	

4.3.11.3 TpTransmitExtended: transmit function for extended addressing

TpTransmitExtended

Prototype	
	TpTransmitExtended(canuint8 addressInfoHandle, canuint8 TA, canuint8 data, canuint16 length)
Parameter	
addressInfoHandle	
TA	Target Address.
data	Pointer to the transmit data.

length	Length of the transmit data (in bytes).
Return code	
canuint8	kTpSuccess: No transmission in progress (ready to send), kTpBusy: Transmission in progress, kTpFailed: Data length is zero, kTpNoChannel: No TP channel available.
Availability	
Only available for “Dispatched Multi TP” classes. The following switch must be defined: #define TP_TYPE_MULTI_DISPATCHED_EXTENDED_ADDRESSING	
Description	
Send the data with the given length to the CAN bus.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
May be used in application context.	
Please note	
-	
Examples	
-	

4.3.11.4 TpTransmitNormalFixed: transmit function for NormalFixed addressing

TpTransmitNormalFixed

Prototype	
	TpTransmitNormalFixed(canuint8 addressInfoHandle, canuint8 TA, canuint8 data, canuint16 length)
Parameter	
addressInfoHandle	
TA	Target Address.
data	Pointer to the transmit data.
length	Length of the transmit data (in bytes).

Return code	
canuint8	kTpSuccess: No transmission in progress (ready to send) kTpBusy: Transmission in progress kTpFailed: Data length is zero kTpNoChannel: No tpChannel available
Availability	
Only available for “Dispatched Multi TP” classes. The following switch must be defined: #define TP_TYPE_MULTI_DISPATCHED_NORMAL_FIXED_ADDRESSING	
Description	
Send the data with the given length to the CAN bus.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
May be used in application context.	
Please note	
-	
Examples	
-	

4.3.11.5 TpTransmitMixed29: transmit function for Mixed-29 addressing

TpTransmitMixed29

Prototype	
	TpTransmitMixed29(canuint8 addressInfoHandle, canuint8 TA, canuint8 AE, canuint8 data, canuint16 length)
Parameter	
addressInfoHandle	
TA	Target Address.
AE	Address Extension.
data	Pointer to the transmit data.
length	Length of the transmit data (in bytes).

Return code	
`canuint8`	kTpSuccess: No transmission in progress (ready to send), kTpBusy: Transmission in progress, kTpFailed: Data length is zero, kTpNoChannel: No TP channel available.

Availability
Only available for “Dispatched Multi TP” classes. The following switch must be defined: `#define TP_TYPE_MULTI_DISPATCHED_MIXED_29_ADDRESSING`
Description
Send the data with the given length to the CAN bus.
Pre-condition(s)
The TP is initialized with TpInitPowerOn().
Post-condition(s)
-
Call context
May be used in application context.
Please note
-
Examples
-

4.3.11.6 TpTransmitMixed29: transmit function for Mixed-29 addressing

`TpTransmitMixed29`

Prototype	

	`TpTransmitMixed29(canuint8 addressInfoHandle,`
	`canuint8 TA,`
	`canuint8 AE,`
	`canuint8 data,`
	`canuint16 length)`

Parameter	
`addressInfoHandle`	
`TA`	Target Address.
`AE`	Address Extension.
`data`	Pointer to the transmit data.
`length`	Length of the transmit data (in bytes).

Return code	
`canuint8`	kTpSuccess: No transmission in progress (ready to send), kTpBusy: Transmission in progress, kTpFailed: Data length is zero, kTpNoChannel: No TP channel available.
Availability	
Only available for “Dispatched Multi TP” classes. The following switch must be defined: `#define TP_TYPE_MULTI_DISPATCHED_MIXED_29_ADDRESSING`	
Description	
Send the data with the given length to the CAN bus.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
May be used in application context.	
Please note	
-	
Examples	
-	

4.3.11.7 TpTransmitMixed11: transmit function for Mixed-11 addressing

TpTransmitMixed29

Prototype	

	`TpTransmitMixed11(canuint8 addressInfoHandle,`
	`canuint8 AE,`
	`canuint8 data,`
	`canuint16 length)`

Parameter	
`addressInfoHandle`	
`AE`	Address Extension.
`data`	Pointer to the transmit data.
`length`	Length of the transmit data (in bytes).

Return code	
canuint8	kTpSuccess: No transmission in progress (ready to send) kTpBusy: Transmission in progress kTpFailed: Data length is zero kTpNoChannel: No TP channel available
Availability	
Only available for “Dispatched Multi TP” classes and at least one AI with mixed-11 as addressing type The following switch must be defined: #define TP_TYPE_MULTI_DISPATCHED_MULTIPLE_ADDRESSING	
Description	
Send the data with the given length to the CAN bus.	
Pre-condition(s)	
The TP is initialized with TpInitPowerOn().	
Post-condition(s)	
-	
Call context	
May be used in application context.	
Please note	
-	
Examples	
-	

4.4 Application callback functions

In the Generation Tool the user can define which callback functions he would like to use from the Transport Protocol. The names can be adjusted by the user. E.g. the prefix **User** can be used instead of **Appl.** These functions will only be provided, if they were configured in the Generation Tool what can be done by entering a function name.

4.4.1 Reception side

4.4.1.1 ApplTpPrecopyCheck: Reception of TP-Frame

ApplTpPrecopyCheck

Prototype	
Single Channel	
Single Receive Channel	canuint8 **ApplTpPrecopyCheck** (CanRxInfoStructPtr rxStruct)
Single Receive Buffer	canuint8 **ApplTpPrecopyCheck** (CanReceiveHandle rxObject)
Multiple Receive Buffer	canuint8 **ApplTpPrecopyCheck** (CanChipDataPtr rxRegPtr)
Multi Channel	
Indexed (MRC)	canuint8 **ApplTpPrecopyCheck** (CanRxInfoStructPtr rxStruct)

Code replicated (SRB)	`canuint8` **`ApplTpPrecopyCheck`** `( CanReceiveHandle rxObject )`
Code replicated (MRB)	`canuint8` **`ApplTpPrecopyCheck`** `( CanChipDataPtr rxRegPtr )`
Parameter	
`rxObject`	Handle of received object
`rxRegPtr`	Pointer to the received data in the CAN Controller receive register
`rxStruct`	Pointer to the receive structure
Return code	
`kCanCopyData`	Received data will be copied using the CAN Driver 's internal copy mechanism
`kCanNoCopyData`	CAN Driver doesn't copy data and doesn't perform indication
Availability	
since versions: TPMC: 2.35.00 | CANgen: 3.88.02 | DBKOMgen: 2.37.01	
Description	
Special functions for the application, which is immediately called after the reception of a TP-CAN-message. If e.g. several CAN-Ids are defined in an ECU for the TP (gateway or multiple ECU) it has to be decided, before the TP is able to make use of the CAN-message, whether the current CAN-message should be processed or not depending on the CAN-ID. This user- check function can be used for it, which is called by the TP on each data reception. If this function returns „1“, the CAN-message is processed by the TP. If this function returns „0“, the CAN- message is dismissed by the TP and the process is finished. The name of this callback-function can be adjusted as desired in the Generation Tool.	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.4.1.2 ApplTpCheckTA: Check if Target Address is valid (version <= 2.72.00)

ApplTpCheckTA

Prototype	
SingleConnectionTp	
	`vuint8 ApplTpCheckTA (vuint8 tpCurrentTargetAddress)`
MultipeConnectionTP	
	`vuint8 ApplTpCheckTA (vuint8 tpCurrentTargetAddress)`
SingleConnectionTp GATEWAY API	

	`vuint8 ApplTpCheckTA (vuint8 tpCurrentTargetAddress,`
	`CanRxInfoStructPtr infoStruct)`

MultipeConnectionTP GATEWAY API	

	`vuint8 ApplTpCheckTA (vuint8 tpCurrentTargetAddress,`
	`CanRxInfoStructPtr infoStruct)`

Parameter	
`tpCurrentTargetAddress`	-
`infoStruct`	
Return code	
`vuint8`	–
Availability	
Only for TP versions less than or equal to 2.72.00. See also chapter 4.4.1.3 for the changed API description available since version 2.73.00. Only for dynamic TP classes: Extended- and Normal Fixed Addressing	
Description	
This function will be called for every reception of a TP-CAN-message. Within this function the application has to decide, if the TargetAddress in the received CAN-frame is valid. If the TargetAddress is not valid and should not be received the return value must be 'kTpNoChannel'. If it should be received the TargetAddress should be returned. See also chapter 7.4.1 Virtual ECU's / 'Multiple EcuNumber' feature. The name of this callback-function can be adjusted as desired in the Generation Tool.	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
-	
Please note	
Until versions: TPMC: 2.35.00 | CANgen: 3.88.02 | DBKOMgen: 2.37.01the function name was called `ApplTpPrecopy()`	
Examples	
-	

4.4.1.3 ApplTpCheckTA: Check if Target Address is valid (since version 2.73.00)

ApplTpCheckTA

Prototype	
SingleConnectionTp	
	t_ta_type **ApplTpCheckTA** (vuint8 tpCurrentTargetAddress)
MultipeConnectionTP	
	t_ta_type **ApplTpCheckTA** (vuint8 tpCurrentTargetAddress)
SingleConnectionTp GATEWAY API	
	t_ta_type **ApplTpCheckTA** (vuint8 tpCurrentTargetAddress, CanRxInfoStructPtr infoStruct)
MultipeConnectionTP GATEWAY API	
	t_ta_type **ApplTpCheckTA** (vuint8 tpCurrentTargetAddress, CanRxInfoStructPtr infoStruct)
Parameter	
tpCurrentTargetAddress	-
infoStruct	
Return code	

t_ta_type	typedef enum
	{
	kTpNone = 0,
	kTpPhysical = 1,
	kTpFunctional = 2
	} t_ta_type;

Availability	
Only for TP versions greater than or equal to 2.73.00. See also the former API description in chapter 4.4.1.2 Only for dynamic TP classes: Extended- and Normal Fixed Addressing	
Description	
This function will be called for every reception of a TP-CAN-message. Within this function the application has to decide, if the TargetAddress in the received CAN-frame is valid and if it is a physical or functional identifer.. If the TargetAddress is not valid and should not be received the return value must be ‘kTpNone’. If the TargetAddress is identified as a physical identifier then ‘kTpPhysical’ should be returned. If the TargetAddress is identified as a functional identifier then ‘kTpFunctional’ should be returned. See also chapter 7.4.1 Virtual ECU’s / ‘Multiple EcuNumber’ feature. The name of this callback-function can be adjusted as desired in the Generation Tool.	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
-	

Please note
Until versions: TPMC: 2.35.00 | CANgen: 3.88.02 | DBKOMgen: 2.37.01the function name was called `ApplTpPrecopy()`
Examples
-

4.4.1.4 ApplTpRxSF: Reception of Single Frame

ApplTpRxSF

Prototype	
SingleConnectionTp	
	`void` **`ApplTpRxSF`**`(void)`
MultipeConnectionTP	
	`void` **`ApplTpRxSF`**`(vuint8 channel)`
Parameter	
`channel`	-
Return code	
	–
Availability	
No restriction	
Description	
This function is called after the reception of a single-frame. ApplTpRxGetBuffer() will be called before. The name of this callback-function can be adjusted as desired in the Generation Tool.	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.4.1.5 ApplTpRxFF: Reception of First Frame

ApplTpRxFF

Prototype	
SingleConnectionTp	
	`void` **`ApplTpRxFF`** `(void)`
MultipeConnectionTP	
	`void` **`ApplTpRxFF`**`(vuint8 channel)`
Parameter	
`channel`	-
Return code	
	–
Availability	
No restriction	
Description	
This function is called after the reception of a first-frame. ApplTpRxGetBuffer() will be called before. The name of this callback function can be adjusted as desired in the Generation Tool.	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.4.1.6 ApplTpRxCF: Reception of Consecutive Frame

ApplTpRxCF

Prototype	
SingleConnectionTp	
	`void` **`ApplTpRxCF`**`(void)`
MultipeConnectionTP	
	`void` **`ApplTpRxCF`**`(vuint8 channel)`
Parameter	
`channel`	-

Return code	
	–
Availability	
No restriction	
Description	
This function is called after the reception of a consecutive-frame. The name of this callback function can be adjusted as desired in the Generation Tool.	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.4.1.7 ApplTpRxCanMessageReceived: Reception of CAN-Frame

ApplTpRxCanMessageReceived

Prototype	
SingleConnectionTp	
	`void` **`ApplTpRxCanMessageReceived`**`(void)`
MultipeConnectionTP	
	`void` **`ApplTpRxCanMessageReceived`**`(vuint8 channel)`
Parameter	
`channel`	-
Return code	
	–
Availability	
until versions: TPMC: 2.35.00 CANgen: 3.88.02 DBKOMgen: 2.37.01 Will be not supported in the future.	
Description	
This function is called after the reception of a CAN-frame and is normally used only in gateways. The name of this callback function can be adjusted as desired in the Generation Tool.	
Pre-condition(s)	
-	

Post-condition(s)
-
Call context
-
Please note
-
Examples
-

4.4.1.8 ApplTpRxGetBuffer: Assign a buffer to a channel

TpTxSetStrictFlowControl

Prototype	
SingleConnectionTp	
	`unsigned char* ApplTpRxGetBuffer(vuint16 dataLength)`
MultipeConnectionTP	

	`unsigned char* ApplTpRxGetBuffer(vuint8 channel,`
	`vuint16 dataLength)`

SingleConnectionTp GATEWAY API	

	`unsigned char* ApplTpRxGetBuffer(vuint16 dataLength`
	`CanRxInfoStructPtr rxStruct)`

MultipeConnectionTP GATEWAY API	

	`unsigned char* ApplTpRxGetBuffer(vuint8 channel,`
	`vuint16 dataLength`
	`CanRxInfoStructPtr rxStruct)`

Parameter	
`dataLength`	-
`channel`	
`rxStruct`	
Return code	
`usigned char`	–
Availability	
No restriction	
Description	
This function is called after reception of the first data to get a buffer with a minimum length of `dataLength` from the application. The application has to return a pointer to this buffer. If the returned pointer is `NULL`, the transport-message will not be received anymore. The name of this callback function can be adjusted as desired in the Generation Tool.	

Pre-condition(s)
-
Post-condition(s)
-
Call context
-
Please note
-
Examples
-

4.4.1.9 ApplTpRxCopyFromCAN: Application Copy Function

ApplTpRxCopyFromCAN

Prototype	
SingleConnectionTp	

	`void ApplTpRxCopyFromCan(vuint8 * source,`
	`                         vuint16 count)`

MultipeConnectionTP	

	`void ApplTpRxCopyFromCan(vuint8 channel,`
	`                         vuint8 * source,`
	`                         vuint16 count)`

Parameter	
`Source`	-
`Count`	
`channel`	
Return code	
	–
Availability	
No restriction	
Description	
The buffer management is done by the application. This function is always called by the Transport Protocol while receiving a TP-CAN-message. The argument **source** points to the receive buffer of the CAN-controller; the argument **count** determines number of data, which has to be copied by the application function. The name of this callback-function can be adjusted as desired in the Generation Tool.	
Pre-condition(s)	
-	
Post-condition(s)	
-	

Call context

-

Please note

-

Examples

```
void ApplTpRxCopyFromCan(vuint8 channel, vuint8 * source, vuint16 count)
{
  (void)memcpy(&(TpRxGetDataBuffer(channel)[TpRxGetDataIndex(channel)]),
                 source, count);
}
```

4.4.1.10 ApplTpRxIndication: Reception closed successful

ApplTpRxIndication

Prototype

SingleConnectionTp	
	`void ApplTpRxIndication(vuint16 dataLength)`
MultipeConnectionTP	

	`void ApplTpRxIndication(vuint8 channel,`
	`vuint16 dataLength)`

Parameter

`dataLength`	-
`channel`	

Return code

	–

Availability

No restriction

Description

This function is called after the completely reception of a single frame message or a multiple frame message. `dataLength` is the number of received bytes in the reception buffer.

The name of this callback function can be adjusted as desired in the Generation Tool.

Pre-condition(s)

-

Post-condition(s)

-

Call context

-

Please note

-

Examples

-

4.4.1.11 ApplTpRxErrorIndication: Reception closed with error

ApplTpRxErrorIndication

Prototype	
SingleConnectionTp	
	`void` **`ApplTpRxErrorIndication`** `(vuint8 errorCode)`
MultipeConnectionTP	

	`void` **`ApplTpRxErrorIndication`** `(vuint8 channel,`
	`vuint8 errorCode)`

Parameter	
`errorCode`	> kTpRxErrFF_SfreceivedAgain: While a reception is in progress a new > Single- or FirstFrame is received, because the running reception will be canceled and set up new. > KTpRxErrWrongSNreceived: A ConsecutiveFrame with a wrong SequenceNumber is received, because of the current reception will be canceled. > KTpRxErrCFTimeout: An awaited ConsecutiveFrame is not received in the right time and a timeout occurs. > KTpRxErrConfIntTimeout: The FlowControl could not transmitted within the necessary time and a (confirmation) timeout occurs.
`channel`	
Return code	
	–
Availability	
No restriction	
Description	
This function will be called if an error occurs on the channel. The channel will be reinitialized afterwards. The name of this callback function can be adjusted as desired in the Generation Tool.	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.4.1.12 ApplTpRxGetTxID: Get CAN Transmit Id

ApplTpRxGetTxID

Prototype	
SingleConnectionTp	
	–
MultipeConnectionTP	
	vuint16 **ApplTpRxGetTxID**(vuint16 receiveId)
Parameter	
receiveId	
Return code	
	–
Availability	
Only for dynamic TP classes: Normal Addressing Insert: #define TP_USE_TX_ID_APPL_CHECK kTpOn in a user-config file to use this feature. !!! Attention: Only until TPMC version 2.60.00	
Description	
This function is called after reception of a First-Frame, to get the Transmit-ID for the FlowControl. The name of this callback function can be adjusted as desired in the Generation Tool.	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.4.2 Reception side for functional messages

Only available if a functional connection group exists.

4.4.2.1 ApplFuncTpPrecopy: Check if Target Address is valid

ApplFuncTpPrecopy

Prototype	
Normal Fixed addressing, Extended addressing:	
	`vuint8 ApplFuncTpPrecopy (vuint8 tpCurrentTargetAddress)`
Normal Fixed addressing, Extended addressing with GATEWAY - API:	

	`vuint8 ApplFuncTpPrecopy (vuint8 tpCurrentTargetAddress,`
	`CanRxInfoStructPtr infoStruct)`

Mixed addressing:	

	`vuint8 ApplFuncTpPrecopy (vuint8 tpCurrentTargetAddress,`
	`vuint8 tpCurrentAddressExtension)`

Mixed addressing with GATEWAY - API:	

	`vuint8 ApplFuncTpPrecopy (vuint8 tpCurrentTargetAddress,`
	`vuint8 tpCurrentAddressExtension,`
	`CanRxInfoStructPtr infoStruct)`

Parameter	
tpCurrentTargetAddress	Contains the N_TA byte of the received message.
tpCurrentAddressExtension	Contains the N_AE byte of the received message.
infoStruct	Pointer to a data structure containing more information concerning the received message (e.g. Raw Id, DLC).
Return code	
`vuint8`	–
Availability	
For TP classes: Extended-, Normal Fixed- and Mixed- Addressing. If a functional connection groups exists and a callback name is configured. The default callback name used is “TpFuncCheckTA”.	
Description	
This function will be called for every reception of a functional TP-CAN-message. Within this function the application has to decide, if the TargetAddress / AddressExtension in the received CAN-frame is valid. If the TargetAddress/AddressExtension is not valid and should not be received the return value must be ‘kTpNoChannel’. If it should be received the TargetAddress should be returned. If the Multiple EcuNumber feature is used, then the concerning EcuNumber must be returned. The name of this callback-function can be adjusted as desired in the Generation Tool.	
Pre-condition(s)	
-	
Post-condition(s)	
-	

Call context
-
Please note
Examples
-

4.4.3 Transmission side

4.4.3.1 ApplTpTxFC: Reception of a Flow Control Frame

ApplTpTxFC

Prototype	
SingleConnectionTp	
	`void ApplTpTxFC(void)`
MultipeConnectionTP	
	`void ApplTpTxFC(vuint8 channel)`
Parameter	
`receiveId`	
Return code	
	–
Availability	
since versions: TPMC: 2.35.00 CANgen: 3.88.02 DBKOMgen: 2.37.01	
Description	
This function is called after the reception of a FlowControl-frame. The name of this callback-function can be adjusted as desired in the Generation Tool.	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.4.3.2 ApplTpTxCanMessageTransmitted: CAN-Message transmitted

ApplTpTxCanMessageTransmitted

Prototype	
SingleConnectionTp	
	`void **ApplTpTxCanMessageTransmitted**(void)`
MultipeConnectionTP	
	`void **ApplTpTxCanMessageTransmitted**(vuint8 channel)`
Parameter	
`channel`	
Return code	
	–
Availability	
No description	
Description	
This function is called each time after a successful transmission of an CAN-message / frame (only for TX connections - .this will mean for SF; FF; CF and not for FC messages) The name of this callback function can be adjusted as desired in the Generation Tool.	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.4.3.3 ApplTpTxNotification: CAN-Frame transmitted

ApplTpTxNotification

Prototype	
SingleConnectionTp	
	`void **ApplTpTxNotification**(vuint8 count)`
MultipeConnectionTP	

	`void **ApplTpTxNotification**(vuint8 channel,`
	`vuint8 count)`

Parameter	
`channel`	

count	
Return code	
	–
Availability	
No restriction	
Description	
This function is called each time after sending Tp-Frames except “Single-Frames” and the “last Consecutive-Frame”. `Count` is the number of transmitted data. The name of this callback function can be adjusted as desired in the Generation Tool.	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

4.4.3.4 ApplTpTxCopyToCAN: Application Copy Function (≥16BIT Controller)

ApplTpTxCopyToCAN

Prototype	
SingleConnectionTp	
	`vuint8` **`ApplTpTxCopyToCAN`** `(TpCopyToCanInfoStructPtr infoStruct)`
MultipeConnectionTP	
	`vuint8` **`ApplTpTxCopyToCAN`** `(TpCopyToCanInfoStructPtr infoStruct)`
Parameter	
infoStruct	
Return code	
vuint8	If everything is fine return ‘`kTpSucces`’ otherwise ‘`kTpFailed`’.
Availability	
No restriction	

Description

The buffer management is done by the application. This function is always called by the Transport Protocol before sending a TP-CAN-message.

The parameter is a pointer to the following structure:

```
struct tTpCopyToCanInfoStruct_s
{
  canuint8   Channel;      /* TP Channel*/
  canuint8*  pDestination; /* Pointer to destination buffer */
  canuint8*  pSource;      /*Pointer to linear source buffer*/
  canuint16  Length;       /* The maximum length to copy */
};
```

The name of this callback-function can be adjusted as desired in the Generation Tool.

Pre-condition(s)

-

Post-condition(s)

-

Call context

-

Please note

Since version 2.35 the TPMC component tries to call `ApplTpCopyToCAN()` again and again until `kTpSuccess` is returned or ‘CAN message confirmation timeout’ occurs.

Examples

```
vuint8 ApplTpCopyToCan(TpCopyToCanInfoStructPtr infoStruct)
{
   (void)memcpy( infoStruct->pDestination, infoStruct->pSource,
   infoStruct->Length);
   return kTpSuccess;
}
```

4.4.3.5 ApplTpTxCopyToCAN: Application Copy Function (8BIT Controller)

ApplTpTxCopyToCAN

Prototype

SingleConnectionTp	

	`vuint8 ApplTpTxCopyToCAN(vuint8 offset,`
	`                         vuint8 count)`

MultipeConnectionTP	

	`vuint8 ApplTpTxCopyToCAN(vuint8 channel,`
	`                         vuint8 offset,`
	`                         vuint8 count)`

Parameter

`Offset`	

Count	
channel	

Return code

vuint8	If everything is fine return kTpSuccess otherwise kTpFailed.

Availability

Caution
Only until TPMC version 2.49.00.

Since TPMC version 2.50.00 the API described in 4.4.3.4 “ApplTpTxCopyToCAN: Application Copy Function (≥16BIT Controller)” is used instead.

Description

The buffer management is done by the application. This function is always called by the Transport Protocol before sending a TP-CAN-message.

The argument offset determines the offset into the sending buffer of CAN Driver (Offset=0..7); the argument “count” determines number of data, which has to be copied by the application function.

The name of this callback function can be adjusted as desired in the Generation Tool.

Pre-condition(s)

-

Post-condition(s)

-

Call context

-

Please note

Since version 2.35 the TPMC component tries to call ApplTpCopyToCAN() again and again until kTpSuccess is returned or ‘CAN message confirmation timeout’ occurs.

TpTxData(channel) can be used to access the transmit buffer of the CAN-driver.

Caution
Do not access the transmit buffer of the CAN-driver elsewhere

Examples

```
vuint8 ApplTpCopyToCan(vuint8 channel,
                       vuint8 offset,
                       vuint8 length)
{
  (void)memcpy( &TpTxData(channel)[ offset ],
                &TpTxGetDataBuffer(channel)[TpTxGetDataIndex(channel)],
                length);
  return kTpSuccess;
}
```

4.4.3.6 ApplTpTxConfirmation: Transmission closed successful

ApplTpTxConfirmation

Prototype	
SingleConnectionTp	
	`void` **`ApplTpTxConfirmation`**`(vuint8 state)`
MultipeConnectionTP	

	`void` **`ApplTpTxConfirmation`**`(vuint8 channel,`
	`vuint8 state)`

Parameter	
`State`	
`cannel`	
Return code	
	-
Availability	
No description	
Description	
This function is called after a single- or a multiple-frame message is transmitted completely. The `state` condition is given as a parameter and can be analyzed by the application. Please note that this is intended for further usage, currently the delivered state is always kTpSuccess. The name of this callback-function can be adjusted as desired in the Generation Tool.	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
-	
Please note	
Currently the ‘state’ parameter is not used. So the default of this parameter is ‘kTpSuccess’.	
Examples	

```
vuint8 ApplTpCopyToCan(TpCopyToCanInfoStructPtr infoStruct)
{
  (void)memcpy( infoStruct->pDestination, infoStruct->pSource,
  infoStruct->Length);
  return kTpSuccess;
}
```

4.4.3.7 ApplTpTxErrorIndication: Transmission closed with error

ApplTpTxErrorIndication

Prototype	
SingleConnectionTp	
	`vuint8` **`ApplTpTxErrorIndication`**`(vuint8 errorCode)`
MultipeConnectionTP	

	`vuint8` **`ApplTpTxErrorIndication`**`(vuint8 channel,`
	`vuint8 errorCode)`

Parameter	
errorCode	> kTpTxErrFCTimeout: An awaited FlowControl timed out > kTpTxErrConfIntTimeout: A TP-CAN-massage could not transmitted within the necessary time and a (confirmation) timeout occurs. > kTpTxErrFCWrongFlowStatus: An invalid FlowControl-frame is received. Only with activated strict message flow checking (TP_USE_STRICT_MSG_FLOW_CHECKING must be set to kTpOn in a user-config file to activate this feature). > kTpTxErrWFTmaxOverrun: WFTmax wait frames are received now (only for MCAN, if TP_ENABLE_MCAN is defined) > kTpTxErrFCOverrun: the receiver reported an Overrun, channel is terminated Old error codes Old error codes since TPMC version 2.35 > kTpTxErrBufferUnderrun: Within the ApplTpCopyToCAN function a buffer-underrun occurs.
cannel	
Return code	
	Hold the channel: kTpHoldChannel Reinitializing / free the channel: kTpFreeChannel
Availability	
No description	
Description	
This function will be called if an error occurs on the channel. The application has now to decide if the channel should be reinitialized or hold for reusing it (only for dynamic TP classes necessary). The name of this callback-function can be adjusted as desired in the Generation Tool.	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
-	

Please note
Currently the ‘state’ parameter is not used. So the default of this parameter is ‘kTpSuccess’.

Examples

```
vuint8 ApplTpCopyToCan(TpCopyToCanInfoStructPtr infoStruct)
{
  (void)memcpy( infoStruct->pDestination, infoStruct->pSource,
  infoStruct->Length);
  return kTpSuccess;
}
```

4.4.4 Administrative Functions

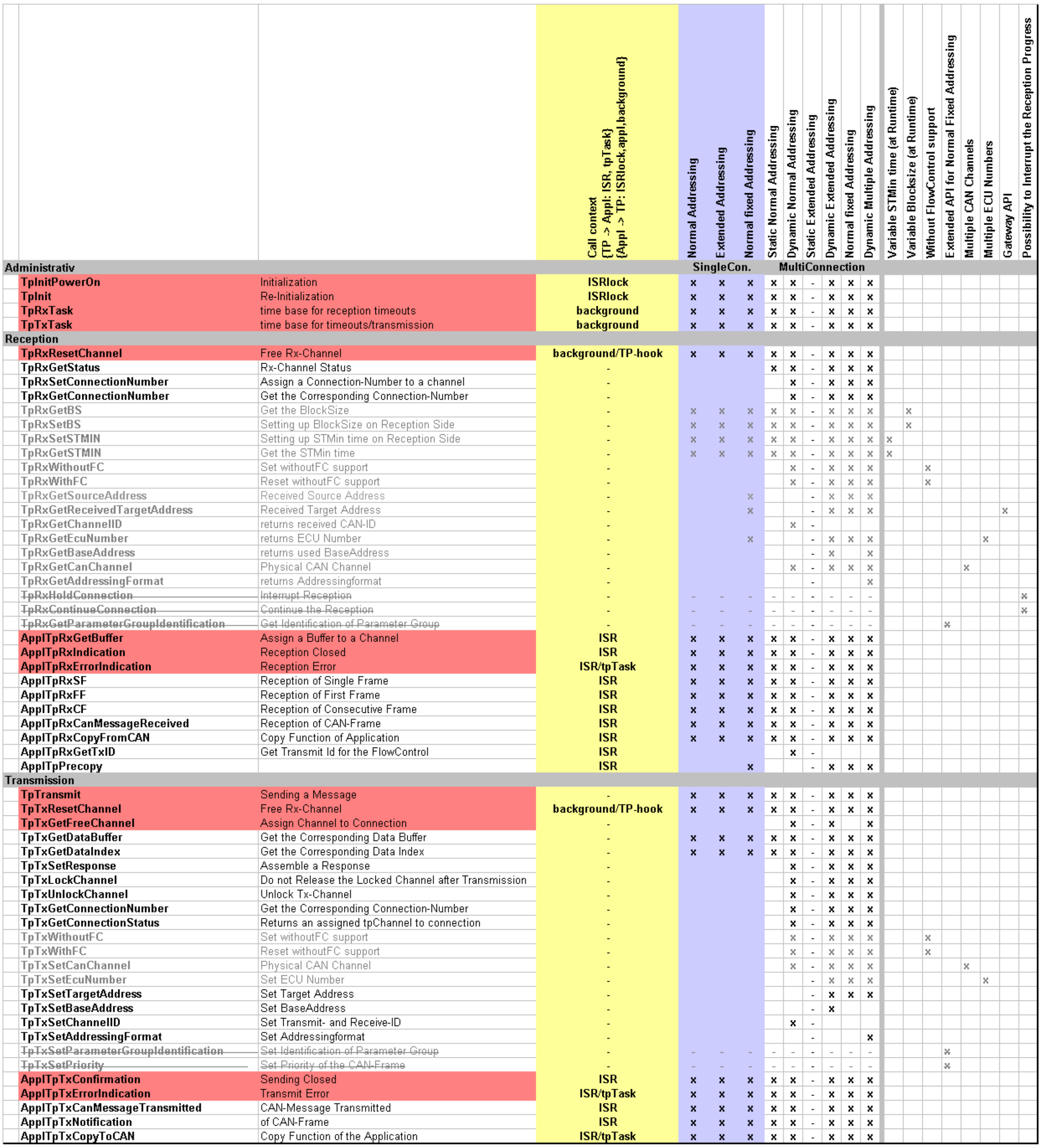
4.4.4.1 ApplTpFatalError: Fatal Error

ApplTpFatalError

Prototype	
SingleConnectionTp	
	void **ApplTpFatalError**(vuint8 errorCode)
MultipeConnectionTP	
	void **ApplTpFatalError**(vuint8 errorCode)
Parameter	
errorCode	User assertions: > KTpErrNoDynObjAtTpInit: Within TpInitPowerOn() it is not possible to allocate the necessary transmit-objects from CAN-driver – please check initialization order > KTpErrChannelNrTooHigh: Possible access of a invalid tpChannel – please check your application calls of the TP-API. > KTpRxErrFcCanIdIsMissing: The CAN-ID of the FlowControl was not set within the ApplTpRxGetBuffer() function for dynamic NormalAddressing – please check your application. > KtpTxErrDatalengthTooHigh: The application tried to transmit more than 4095 bytes of data – please check your application. > KTpTxErrWrongFrameAtPretransmitSpecified: Internal state-machine check – please get in contact with us. > KTpTxErrNoStateSpecified: Internal state-machine check – please get in contact with us. > kTpRxErrNoStateSpecified: Internal state-machine check – please get in contact with us. > kTpErrChannelNotInPreTransmitState: The application tried to configure a not assigned tpChannel in a dynamic TP class – please check your application.

	> KTpErrWrongAddressingFormat: The application tried to configure a tpChannel for a wrong AddressingMode (e.g. TpTxSetTargetAddress for NormalAddressing configured tpChannel) in a dynamic TP class – please check your application - please check your application. > KTpRxErrSetResponseWithoutFc: The function TpTxSetResponse() is called for without-FC configured tpChannel - please check your application. > KTpTxErrSetResponseWithoutFc: The function TpTxSetResponse() is called for without-FC configured tpChannel - please check your application. > KTpErrChannelNotInUse: The application tried to get information about an unused tpChannel – please check your application. Internal assertions: > KTpErrChannelNrTooHigh: Possible access of a invalid tpChannel – please check the stack-usage. > KTpRxErrNotInWaitCFState: Internal state-machine check – please get in contact with us. > KTpErrChannelNotInUse: Internal state-machine check – please get in contact with us. > KTpErrNoCanChannelFound: The CAN-driver confirmation function is called with a wrong Handle, because it is not possible to calculate the corresponding CAN-channel – please get in contact with us.
Return code	
	-
Availability	
Until versions CANgen: 3.88.02 DBKOMgen: 2.37.01 TP-assertions are activated if the “Debug level” in CAN-Driver includes “User”/”Internal”	
Description	
This function will be called if a fatal error occurs. The name of this callback function is not changeable	
Pre-condition(s)	
-	
Post-condition(s)	
-	
Call context	
-	
Please note	
-	
Examples	
-	

5 Transmission Attributes & Callback functions



		Call context (TP -> Appl: ISR, tpTask) (Appl -> TP: ISRlock,appl,background)	Normal Addressing	Extended Addressing	Normal fixed Addressing	Static Normal Addressing	Dynamic Normal Addressing	Static Extended Addressing	Dynamic Extended Addressing	Normal fixed Addressing	Dynamic Multiple Addressing	Variable STMin time (at Runtime)	Variable Blocksize (at Runtime)	Without FlowControl support	Extended API for Normal Fixed Addressing	Multiple CAN Channels	Multiple ECU Numbers	Gateway API	Possibility to Interrupt the Reception Progress
Administrativ			SingleCon.			MultiConnection													
TpInitPowerOn	Initialization	ISRlock	x	x	x	x	x	-	x	x	x								
TpInit	Re-Initialization	ISRlock	x	x	x	x	x	-	x	x	x								
TpRxTask	time base for reception timeouts	background	x	x	x	x	x	-	x	x	x								
TpTxTask	time base for timeouts/transmission	background	x	x	x	x	x	-	x	x	x								
Reception																			
TpRxResetChannel	Free Rx-Channel	background/TP-hook	x	x	x	x	x	-	x	x	x								
TpRxGetStatus	Rx-Channel Status	-				x	x	-	x	x	x								
TpRxSetConnectionNumber	Assign a Connection-Number to a channel	-					x	-	x	x	x								
TpRxGetConnectionNumber	Get the Corresponding Connection-Number	-					x	-	x	x	x								
TpRxGetBS	Get the BlockSize	-	x	x	x	x	x	-	x	x	x		x						
TpRxSetBS	Setting up BlockSize on Reception Side	-	x	x	x	x	x	-	x	x	x		x						
TpRxSetSTMIN	Setting up STMin time on Reception Side	-	x	x	x	x	x	-	x	x	x	x							
TpRxGetSTMIN	Get the STMin time	-	x	x	x	x	x	-	x	x	x	x							
TpRxWithoutFC	Set withoutFC support	-					x	-	x	x	x			x					
TpRxWithFC	Reset withoutFC support	-					x	-	x	x	x			x					
TpRxGetSourceAddress	Received Source Address	-			x			-	x	x	x								
TpRxGetReceivedTargetAddress	Received Target Address	-			x			-	x	x	x							x	
TpRxGetChannelID	returns received CAN-ID	-					x	-											
TpRxGetEcuNumber	returns ECU Number	-			x			-	x	x	x						x		
TpRxGetBaseAddress	returns used BaseAddress	-						-	x		x								
TpRxGetCanChannel	Physical CAN Channel	-					x	-	x	x	x					x			
TpRxGetAddressingFormat	returns Addressingformat	-						-			x								
~~TpRxHoldConnection~~	~~Interrupt Reception~~	-	-	-	-	-	-	-	-	-	-								x
~~TpRxContinueConnection~~	~~Continue the Reception~~	-	-	-	-	-	-	-	-	-	-								x
~~TpRxGetParameterGroupIdentification~~	~~Get Identification of Parameter Group~~	-	-	-	-	-	-	-	-	-	-				x				
ApplTpRxGetBuffer	Assign a Buffer to a Channel	ISR	x	x	x	x	x	-	x	x	x								
ApplTpRxIndication	Reception Closed	ISR	x	x	x	x	x	-	x	x	x								
ApplTpRxErrorIndication	Reception Error	ISR/tpTask	x	x	x	x	x	-	x	x	x								
ApplTpRxSF	Reception of Single Frame	ISR	x	x	x	x	x	-	x	x	x								
ApplTpRxFF	Reception of First Frame	ISR	x	x	x	x	x	-	x	x	x								
ApplTpRxCF	Reception of Consecutive Frame	ISR	x	x	x	x	x	-	x	x	x								
ApplTpRxCanMessageReceived	Reception of CAN-Frame	ISR	x	x	x	x	x	-	x	x	x								
ApplTpRxCopyFromCAN	Copy Function of Application	ISR	x	x	x	x	x	-	x	x	x								
ApplTpRxGetTxID	Get Transmit Id for the FlowControl	ISR					x	-											
ApplTpPrecopy		ISR			x			-	x	x	x								
Transmission																			
TpTransmit	Sending a Message	-	x	x	x	x	x	-	x	x	x								
TpTxResetChannel	Free Rx-Channel	background/TP-hook	x	x	x	x	x	-	x	x	x								
TpTxGetFreeChannel	Assign Channel to Connection	-					x	-	x		x								
TpTxGetDataBuffer	Get the Corresponding Data Buffer	-	x	x	x	x	x	-	x	x	x								
TpTxGetDataIndex	Get the Corresponding Data Index	-	x	x	x	x	x	-	x	x	x								
TpTxSetResponse	Assemble a Response	-					x	-	x	x	x								
TpTxLockChannel	Do not Release the Locked Channel after Transmission	-					x	-	x	x	x								
TpTxUnlockChannel	Unlock Tx-Channel	-					x	-	x	x	x								
TpTxGetConnectionNumber	Get the Corresponding Connection-Number	-					x	-	x	x	x								
TpTxGetConnectionStatus	Returns an assigned tpChannel to connection	-					x	-	x	x	x								
TpTxWithoutFC	Set withoutFC support	-					x	-	x	x	x			x					
TpTxWithFC	Reset withoutFC support	-					x	-	x	x	x			x					
TpTxSetCanChannel	Physical CAN Channel	-					x	-	x	x	x					x			
TpTxSetEcuNumber	Set ECU Number	-						-	x	x	x						x		
TpTxSetTargetAddress	Set Target Address	-						-	x	x	x								
TpTxSetBaseAddress	Set BaseAddress	-						-	x										
TpTxSetChannelID	Set Transmit- and Receive-ID	-					x	-											
TpTxSetAddressingFormat	Set Addressingformat	-						-			x								
~~TpTxSetParameterGroupIdentification~~	~~Set Identification of Parameter Group~~	-	-	-	-	-	-	-	-	-	-				x				
~~TpTxSetPriority~~	~~Set Priority of the CAN-Frame~~	-	-	-	-	-	-	-	-	-	-				x				
ApplTpTxConfirmation	Sending Closed	ISR	x	x	x	x	x	-	x	x	x								
ApplTpTxErrorIndication	Transmit Error	ISR/tpTask	x	x	x	x	x	-	x	x	x								
ApplTpTxCanMessageTransmitted	CAN-Message Transmitted	ISR	x	x	x	x	x	-	x	x	x								
ApplTpTxNotification	of CAN-Frame	ISR	x	x	x	x	x	-	x	x	x								
ApplTpTxCopyToCAN	Copy Function of the Application	ISR/tpTask	x	x	x	x	x	-	x	x	x								

Figure 5-1 Transmission attributes and callback functions

6 Integration of CANbedded Components into a Customer Project

6.1 Requirements to the Customer System Environment

A customer system environment from the CANbedded component point of view is the environment (system architecture) where the component together with other CANbedded components, an operation system, startup code, system control software and the application is running.

To full fill the different requirements to the component architecture like small ROM and RAM footprint, short API runtime and short interrupt lock times (global and only CAN/LIN/others bus interrupts) during the API execution, some requirements to the customer's system environment and the component usage in that system has to be given to and kept by the user.

The requirements and needs to use CANbedded components in a customer specific project are listed in this chapter. It is necessary to check the requirements, preconditions and needs carefully to guaranteed the correct and consistent usage of the software in the resulting system and to prevent malfunction and data consistency problems during the system execution (in the vehicle in the field).

6.2 Component Integration to the Customer Project

6.2.1 Requirements to the Component Initialization in a Customer Project

The correct sequence for all CANbedded component initialization calls (e.g. CAN Driver, network management, interaction layer …) depends on the needs for the whole, vehicle manufacturer specific integration package. Therefore the correct call location in the context to the other (CANbedded) power up initialization calls for this component is just a example.

The following rules are valid for each use case of a CANbedded component in a customer project and must be guaranteed to prevent faulty situations:

1) The component must be initialized after the primary CAN Driver initialization via CanInitPowerOn().
2) The component must be initialized during the global interrupt is locked, to prevent any interrupt occurrences during the initialization sequence of this and ALL other CANbedded modules. Therefore the requirement is to make sure the global interrupt is disabled during the whole initialization sequence of all CANbedded components (driver, IL, NM, TP, diagnostics …).
3) Please note, that the usage of CanDisableInterrupt and CanRestoreInterrupt is incorrect to lock the global interrupt during the CANbedded initialization sequence. A customer project specific global interrupt lock and unlock is necessary.
4) The customer system architecture must guarantee that all CANbedded modules are initialized before the first usage of any API or variable access in the customer's application software is performed.
5) The call to the component initialization function TpInitPowerOn() will reset the component state to the initial state. Therefore it is NOT recommended to call the component initialization function during the system runtime to e.g. terminate

something. Please check carefully, if the call to this API is valid (and helpful) in the planned application context.

6) Please note, that the call to the component initialization function may be runtime consuming, especially if there are additional callbacks to the application are performed and that the global interrupts are locked during that time, too.
7) If an OSEK/OS is used, the basic initialization sequence has to be performed in the startup-hook or, alternatively in an task used to initialized the whole system. Please check, that the global interrupt is locked during the startup hook execution to ensure the required data consistency. This is true for all osCAN OSEK but not for each OSEK/OS on the market. If the initialization is performed in a task, the interrupt must be locked by the user for each OSEK/OS implementation.

6.2.2 Requirements to Component API Usage in a Customer Project

1) The CANbedded component needs a first initialization of all internal variables and states via the call of the initialization API function TpInitPowerOn(). It is not allowed to use any API or data structure of the component before the primary initialization has been performed. See chapter 6.2.1 Requirements to the Component Initialization in a Customer Project for details to the component needs according to the initialization sequence.
2) The cyclic function(s) (e.g. TpRxTask()/TpTxTask()) of a component must not be called on interrupt level (e.g. the timer interrupt). It is strictly forbidden, that the cyclic called component API interrupts the component’s API functions running in the (CAN/LIN) interrupt context or an other component API’s. See chapter 6.2.3.1 Common Requirements for details.
3) It is not allowed to call any CANbedded API function in the context of an interrupt, if this is not explicitly allowed or required in this documentation.
4) Please refer to chapter 6.2.3 Requirements to the Customer Project Operating System for the component requirements to the operating system.

6.2.3 Requirements to the Customer Project Operating System

The operating system used in the customer project has to fulfill the rules listed in chapter 6.2.3.1 Common Requirements to guarantee data consistency of the internal and external component states and values.

6.2.3.1 Common Requirements

The component offers different API functions and global variable/state access to the application program. Some of these API functions are necessary to fulfill the basic functionality of the component. This is e.g. the initialization and the cyclic called function to realize the internal time base and the state handling.

The cyclic called API function TpRxTask()/TpTxTask() is also called TASK in the context of this chapter. Due to the need for fast (1 - 10ms) cyclic calls, this tasks are often called erroneously by calling this API function in an timer interrupt context. This is STRICTLY forbidden.

The list below describes the common rules for all component API calls. The documentation of the API functions and the component callback functions describes the deviations from this rules if, e.g. the API is allowed to be called during the TASK is running.

Please check carefully, if this restrictions are valid in your system:

- API functions must not interrupt the (CAN/LIN) RX/TX interrupt service functions
- API functions must not interrupt the TASK functions
- API functions must not interrupt other API functions of the same component
- TASK functions must not interrupt API functions of the same component
- If there are multiple TASK functions for a component: TASK function must not interrupt other TASK functions of the same component
- TASK functions must not interrupt the (CAN/LIN) RX/TX interrupt service functions

Info

- API and TASK functions are protected against interruption by the (CAN/LIN) RX/TX interrupt service functions
- There are no limitations for interruptions of the component API's with other, independent interrupt service functions (e.g. A/D converter, SIO lines, ...)

6.2.3.2 Round-Robin-Scheduler and Comparable OS Approaches

If the used operating system works like a round-robin scheduler or comparable and there is only one common call level for application and CANbedded APIs with additional, small interrupt handlers, the preconditions as described in chapter 6.2.3.4 should be valid.

6.2.3.3 Usage of OSEK/OS

The component can be used together with an OSEK operating system. The component itself is operating system independent and can therefore be used together with an OSEK/OS, if the rules listed in chapter 6.2.3.1 are fulfilled.

OSEK/OS can be configured to 4 different setups (BCC1 to ECC2). Depending on the selected setup, OSEK/OS is non-preemptive or (full-)preemptive. The preemptive setups are able to run non-preemptive and preemptive tasks. Please refer to the chapters 6.2.3.4 and 6.2.3.5 for further details.

If an OSEK/OS is used, the basic initialization sequence has to be performed in the startup-hook or, alternatively in an task used to initialized the whole system. Please check, that the global interrupt is locked during the startup hook execution to ensure the required data consistency. This is true for all osCAN OSEK but not for each OSEK/OS on the market. If the initialization is performed in a task, the interrupt must be locked by the user for each OSEK/OS implementation.

6.2.3.4 Non-Preemptive Operating System

If an non-preemptive OS is used, there are no limitations to the usage of CANbedded component API's on task/main level due to an task change is started by an OS-API call or by exiting a function called directly by the OS scheduler. Due to this there is no situation with possible dangerous interruptions of component API executions in this environment.

Non-preemptive approaches are using also interrupt handlers for e.g. CAN, LIN, A/D and D/A conversion and other things. Until the requirements listed in chapter 6.2.3.1 are fulfilled, no critical situation according to data consistency and the CANbedded component usage occurs. The CANbedded component itself is able to cope with the interruption via the internal connection to the CAN/LIN driver.

6.2.3.5 Preemptive Operating System

If the CANbedded component has to be used in a full-preemptive environment, some additional restrictions have to be kept in mind. If this is not explicitly allowed, please check carefully, that the restrictions listed in chapter 6.2.3.1 are fulfilled by the system setup.

Possible solutions for a save usage of the CANbedded component may be calling the cyclic functions and API's in non-preemptive tasks or to lock task changes during the execution of the cyclic function calls and the component APIs.

It is not recommended to solve the restrictions via a special task priority setup due to possible maintenance issues when changing and extending the software system in the future.

7 Advanced usage

7.1 Separation of TimerTask and TransmissionTask (StateTask)

Until TPMC version 2.35 there is a combination of a timer observation and the handling of transmission requests in one task function. By the demand of faster TP transmission the most popular possibility is to separate the transmission mechanism from the timer task. Since TPMC version 2.35 TimerTask and TransmissionTask are separated.

The 'TimerTask' includes the time observation. The 'StateTask' includes the transmission handling of the CAN-frames. Especially the retry of the transmission while CanTransmit() cannot accept the message, because the (all) TX registers are currently in use.
Like the former 'Task' function (TpXxTask()) the current 'Task' function (TpXxTask()) includes the call of both tasks to have a full compatibility. So it must be called further on periodically. The 'StateTask' can be called out of a fixed time periods in addition.

⚠	**Caution** It is not necessary to call the 'StateTask', if the CAN Driver queue is enabled.

void TpTxTask(void)

- static void TpTxTimerTask(void) (not visible for the application)
- void TpTxStateTaskAllChannels(void)

void TpRxTask(void)

- void TpRxTimerTask(void) (not visible for the application)
- void TpRxStateTaskAllChannels(void)

The 'StateTaskAllChannels' iterates over all tpChannels. To speed up only one connection. a 'StateTask' is provided, which is handles the transmission of this connection.
void TpTxStateTask(vuint8 tpChannel)
void TpRxStateTask(vuint8 tpChannel)

7.2 Fast transmission of ConsecutiveFrames

Available since TPMC version 2.35.

The TP-layer calculates the STmin time based on the CallCycle of the TpTimerTask().To guarantee that a under run of the STmin is not possible, one CallCycle is added. This conservative way of calculation do not fit the demand of a fast transmission.

The added feature includes a possibility to transmit a TP-frame as quick as possible. Typically this feature can be used for a fast re-programming of ECU's through Gateways or Testers.

The feature can’t be enabled through the GenTools. A user-config file has to be used, including following define:

#define TP_USE_FAST_TX_TRANSMISSION kTpOn

7.2.1 Usage

The TP provides a special API function which assembles and transmits the next CF-frame by skipping the internal timer for the minimum sending distance (STmin). This means the application has the possibility to transmit the next CF frame faster than the calculated minimum sending distance of the TP module allows.
Normally the timer will be reloaded with the value of the minimum sending distance and is observed in the TpTxTimerTask(). By calling the function TpTxPrepareSendImmediate() the timer of the TP is stopped. If the preparation returns a ‘kTpSuccess’ the application gets the responsibility of transmitting the next ConsecutiveFrame. The application can reload an (application) alarm-timer with the STmin value of the FlowControl-frame by calling the function TpTxGetSTminInFrame(). If the alarm occurs (timer is decremented to zero) the application can transmit the ConsecutiveFrame by calling the function TpTxSendImmediate(), which prepares the CF-frame and calls the TpTxStateTask() to transmit the frame immediately.

7.2.2 Application example

For non-zero STmins:

```
void ApplTpTxFC(canuint8 channel)
{
  if(kTpSuccess == TpTxPrepareSendImmediate(channel))
  {
    TpTxSendImmediate(channel);
  }
}
void ApplTpTxCanMessageTransmitted(canuint8 channel)
{
  canuint8 stminTime;

  if(kTpSuccess == TpTxPrepareSendImmediate(channel))
  {
   stminTime = TpTxGetSTminInFrame(channel);

    /* load an OSEK-OS alarm (in ms) */
    SetRelAlarm(TpSepAlarm, MSEC(stminTime),0);

    /* after alarm time expires: TpTxSendImmediate(channel); */
  }
}
```

For zero STmins (fast as possible):

Attention: Due to the current priority rules it could be possible that no real parallel transmission is possible. All other channels are not handled anymore while another transmission is running.

```
void ApplTpTxFC(canuint8 channel)
{
  if(kTpSuccess == TpTxPrepareSendImmediate(channel))
  {
    TpTxSendImmediate(channel);
  }
}
void ApplTpTxCanMessageTransmitted(canuint8 channel)
{
  if(kTpSuccess == TpTxPrepareSendImmediate(channel))
  {
    TpTxSendImmediate(channel);
  }
}
```

7.3 Normal Fixed Addressing

7.3.1 Multiple ECU’s

Multiple ECU’ s are control units which are assembled several times within the CAN network with the same software (example: seat in the front on the left hand side and on the right hand side). In this case, the application has to decide at run-time, which ECU is actually installed and has to set-up these parameters dynamically.

7.3.1.1 Using the CANgen configuration tool

The configuration tool does not apply the ECU information but it provides all possible values for the application as constants in the generated code.

E.g.: In the generated tp_cfg.h file you will find constants for all existing ECU numbers:

```
#define kTpEcuNumber0      0x10
#define kTpEcuNumber1      0x11
#define kTpEcuNumber2      0x12
#define kTpEcuNumber3      0x13
…
```

In case of using the CANgen configuration tool the application must accomplish two things now at Power On time:

a) The actual ECU number must be set using the ComSetCurrentECU() API.

b) The actual ECU number must be provided to the TPMC.

Code example:

```
extern canuint8 tpEcuNumber;

canuint8 tpEcuNumber;

void main(void)
{
  CanInitPowerOn();
  ComSetCurrentECU(currentECU);
  ...
  if ( FirstECUis selected) {
    tpEcuNumber = kTpEcuNumber0;
  }
  else if (SecondECU is selcted) {
    tpEcuNumber = kTpEcuNumber1;
 }
  TpInitPowerOn();  /* For some configuration it could be also
    DiagInitPowerOn()  with implicit TPMC initialization       */
  ...
  <EnableCAN_ISR>
}
```

7.3.1.2 Using the GENy configuration tool

The configuration tool does not apply the ECU information completely but it provides all possible values for the application as constants in the generated code.

E.g.: In the generated tp_par.c file a kTpEcuNumber_field[] is provided for all existing ECU numbers:

```
vuint8 kTpEcuNumber_field [4] = {
      0x10,
      0x11,
      0x12,
      0x13
}
```

In case of using the GENy configuration tool there is left one thing now the application must accomplish at Power On time:

a) The actual ECU number must be set using the ComSetCurrentECU() API.

Code example:

```
void main(void)
{
  CanInitPowerOn();
  ComSetCurrentECU(currentECU);
  ...
  TpInitPowerOn();  /* For some configuration it could be also
    DiagInitPowerOn()  with implicit TPMC initialization       */
  ...
  <EnableCAN_ISR>
}
```

7.4 Extended- and Normal Fixed Addressing

7.4.1 Virtual ECU's / 'Multiple EcuNumber' feature

'Virtual ECU's' are control units which include the logic of more than one ECU. In the network they have to react for more than one ECU number. The application has to decide which ECU number should be received and which not.

For versions < 2.73.00:

All TargetAddresses (except the functional TargetAddress 0xFF) will be received through the Transport Layer. Following the reception of a TP-frame the application callback ApplTpPrecopy() is called by the Transport Layer. In this function the application has to decide which TargetAddress should be received and which not. In this function the application gets the received TargetAddress and has to return the TargetAddress itself to receive TransportFrames. To not receive the following TransportFrames the return value has to be 'kTpNoChannel' (0xff).

If the received TargetAddress e.g. is a part of a functional range, the application can modify the received TargetAddress by returning another TargetAddress in the ApplTpPrecopy function. If the returned value is unequal to the received the Transport Layer will receive the TransportFrames with this TargetAddress and not with the received (the responded FlowControl is also modified).

```
canuint8 ApplTpCheckTA(vuint8 targetAddress)
{
  vuint8 result;
  switch(targetAddress)
  {
    case TargetAddress_0:
    case TargetAddress_1:
            ...
    case TargetAddress_n:
      result = targetAddress;
      break;
    default:
      result = kTpNoChannel;
  }
  return result;
}
```

For versions >= 2.73.00:

All TargetAddresses are received through the Transport Layer. Following the reception of a TP-frame the application callback ApplTpPrecopy() is called by the Transport Layer. In this function the application has to decide which TargetAddress should be received and which not. The application gets the received TargetAddress and has to return either ‘kTpPhysical’ or ‘kTpFunctional’. To not receive any subsequent TP Frames the application returns ‘kTpNone’.

```
t_ta_type ApplTpCheckTA(vuint8 targetAddress)
{
  t_ta_type result;
  if(targetAddress == MY_ECU_NUMBER)
            {
    result = kTpPhysical;
  }
  else if((targetAddress >= TP_LOWEST_FUNCTIONAL_ADDRESS ) &&
          (targetAddress <= TP_HIGHEST_FUNCTIONAL_ADDRESS))
    result = kTpFunctional;
  }
  else
  {
    result = kTpNone;
  }
  return result;
}
```

7.5 Using different CAN-Identifiers

For some purposes different CAN-Ids, as well 11-Bit standard as also 29-Bit extended identifiers shall be used for the Normal Addressing type. If so, the TPMC provides two configuration opportunities to handle this requirement either statically at configuration time or dynamically at runtime.

7.5.1 Statically configured CAN-Ids

By default 11-Bit standard Ids are used with Normal Addressing. If 29-Bit extended Ids are requested by the user and thus also entered as Addressing Information in the GENy generation tool, then the preprocessor switch TP_USE_EXT_IDS_FOR_NORMAL is generated with the value kTpOn. The code is now applicable to be used with 29-Bit CAN-Ids.

7.5.2 Dynamically configured CAN-Ids

If the user has the necessity to handle both kinds of CAN-Ids during runtime, then in the GENy generation tool different CAN-Ids can be entered for different Addressing Informations. Now the preprocessor switch TP_USE_MIXED_IDS_FOR_NORMAL is generated with the value kTpOn in addition and the code is now applicable to be used simultaneously with 11- and 29- Bit CAN-Ids.

7.5.3 Additional API functions

If both kinds of CAN-Ids are used then the additional API function

`canuint8 TpRxGetChannelIDType(canuint8 tpChannel)` is provided.

This function either returns kTpCanIdTypeStd for 11-Bit or kTpCanIdTypeExt for 29-Bit identifiers.

7.6 Transmissions without Flow Control frames

For some purposes the usage of FC frames might be omitted. Please note that this feature is not supported for single connection TP.

If using a dynamic Tp Class then the provided API functions TpRxWithoutFC resp. TpTxWithoutFC can be used (see 0, 4.2.3.28) to control the FC usage.

If using a static Tp Class then a channel specific FC control information must be provided at compile time for the TP containing the information if FC frames shall be used or not for a specific channel either on the Rx- and/or on the Tx- side.

The definition and usage of the FC control array must be as described below:

```
vuint8 TpRxFlowControl[kTpRxChannelCount];
vuint8 TpTxFlowControl[kTpTxChannelCount];
```

In the default case, if the usage of FC frames is required, then the FC control array contains a value of “1” for the belonging Rx- or Tx- channel. If FC frames shall be suppressed, then the FC control array contains a value of “0” for the belonging Rx- or Tx- channel.

Example:

```
vuint8 TpRxFlowControl[3] =
{ 1,    // use FC frames
  1,    // use FC frames
  0     // use no FC frames
};  //
vuint8 TpTxFlowControl[3] =
{ 1,    // use FC frames
  1,    // use FC frames
  0     // use no FC frames
};
```

8 Example for the user

8.1 Administrative usage

The Transport Protocol has to be initialized before all other functions were called. This initialization has to be done after initializing the CAN-driver (**CanInitPowerOn()**), possibly if the interrupts are still locked. The Transport Layer is ready for reception after calling **TpInitPowerOn().**

To perform the state machine the functions **TpRxTask()** and **TpTxTask()** have to be called periodically.

If the application wants to have access to the API of the TPMC-component it has to include the “tpmc.h” file after including of the “can_inc.h” file.

8.2 How to Transmit a Tp-Frame?

8.2.1 Static Normal Addressing

First you need an own buffer with your data which should be transmitted. To start the transmission simply call **TpTransmit().**

```
if (TpTransmit(tpChannel, appl-buffer, appl-data-length) != kTpSuccess)
{
  /* Error case – transmission was not successful */
}
```

A confirmation function is called after the complete transmission. It can be used to release buffers...

```
void ApplTpTxConfirmation(vuint8 tpChannel, vuint8 state)
{
```

If you want an own copy mechanism to move the data from your buffer into CAN buffer you have to use the function **ApplTpTxCopyToCan()** (This can be configured in the Generation Tool).

8.2.2 Dynamic Addressing

(Normal- / Normal Fixed- / Extended- / Multiple-Addressing)

Before the application can call **TpTransmit()** (refer 8.2 How to Transmit a Tp-Frame?) a transport channel has to be requested. The function **TpTxGetFreeChannel()** returns a free transport channel or – if no channel is available at the moment – **kTpNoChannel**. After a channel is assigned, the channel has to parameterized by the application. In the example below, the application will set the Transmit ID and Receive ID (Dynamic Normal Addressing) before sending the data.

Important: replace the cursive words by your own

```
tpChannel = TpTxGetFreeChannel(connection-number);
if(tpChannel != kTpNoChannel)
{
    /* normal addressing */
    TpTxSetChannelID(tpChannel, TransmitID, ReceiveID);

    if (TpTransmit(tpChannel, appl-buffer, appl-data-length) != kTpSuccess)
    {
      /* Error case – transmission was not successful */
    }
}
```

The callback functions provide only the tpChannel as a parameter. To get the unique connection-number out of this tpChannel the function **TpTxGetConnectionNumber(tpChannel)** is provided

```
void ApplTpTxConfirmation(vuint8 tpChannel, vuint8 state)
{
  switch(TpTxGetConnectionNumber(tpChannel))
   .
   .
```

8.3 How to Receive a Tp-Frame

It is only possible to get an Indication by a function callback. The reception progress is completed by the Transport Layer.

Important: The Transport Layer blocks the receive tpChannel as long as the application desires. To free the receive channel call **TpRxResetChannel()**.

```
void ApplTpRxIndication ( vuint8 tpChannel, vuint16 dataLength)
{
   ...
   ...
   TpRxResetChannel(tpChannel);
   ...
   ...
}
```

The Transport Layer supports only buffer-management by the application. If data will be received, it is important to the Transport Layer to get a buffer into which the data can be moved.

```
vuint8 * ApplTpRxGetBuffer (vuint8 tpChannel, vuint16 length)
{
  if (Is_ReceiveDataBuffer_free)
  {
    Set_ReceiveDataBuffer_Used;

    if (length <= MaxLength)
    {
         /* return a valid data buffer */
      return ReceiveDataBuffer;
    } else {
        /* length is too big for the ReceiveDataBuffer - do not receive the data */
      return NULL;
  } else {
        /* ReceiveDataBuffer is not free - do not receive the data */
     return NULL;
  }
}
```

8.4 How to Send a Response on a Received Transport-Frame

Normally the application has to set transmission attributes like TargetAddress, TargetIdentifier or physical CanChannel (depending on the addressing mode and configuration). So if the application want to send a response to the sender of a received transport-frame it has to set these transmission attributes. For this case it can do it easily by using the function TpTxSetResponse(). The Preconditions are only the Rx-Channel - which is still blocked - from the sender and a free Tx-Channel for the transmission.

```
if ( (txTpChannel = TpTxGetFreeChannel(user_connection)) != kTpNoChannel )
{
  TpTxSetResponse(rxTpChannel, txTpChannel);
  TpRxResetChannel(rxTpChannel);
  TpTransmit(txTpChannel, ...);
}
```

8.5 How to serve Different Connections (only dynamic channels)

The dynamic TP classes does not support connection specific callback functions.
Therefore the application needs an easy handling between the different connections with less resource requirements. Especially the diagnostic-layer must be handled

8.5.1 How to serve the diagnostic connection

This is also an example to serve different connections in your own application! I.e. you can derive from the diagnosis example to your own.

Reception part:

Within the ‘ApplTpRxGetBuffer()’ the application is responsible to distinguish between the different connections. If the right connection is found a connection-number can be set to have in the later callbacks a faster decision.

(Dynamic Normal Addressing) The received CAN-ID (for the diagnosis) is unique (get it with: TpRxGetChannelID(tpChannel))

Transmission part:

At the transmission the connection-number is unique. The diagnosis uses the connection-numbers “kDiagConnection” and “kDiagAddConnection”.

```
unsigned char* ApplTpRxGetBuffer(vuint8 tpChannel, vuint16 tpRxDataLength)
{
  switch(TpRxGetChannelID(tpChannel))
  {
  case DIAG_RECEIVE_ID:
    TpRxSetConnectionNumber(tpChannel, kDiagConnection);
    return DiagTpGetRxBuffer(tpChannel, tpRxDataLength);
  case APPL_RECEIVE_ID:
    TpRxSetConnectionNumber(tpChannel, CONNECTION_0);
    /* Check for an valid application buffer */
    return APPLICATION_BUFFER;
  default:
    return NULL;
    break;
  }
}

void ApplTpRxIndication(vuint8 tpChannel, vuint16 tpRxDataLength)
{
  switch(TpRxGetConnectionNumber(tpChannel))
  {
  case kDiagConnection:
    DiagPhysReception(tpChannel, tpRxDataLength);
    break;
  case CONNECTION_0:
    UserTpRxIndication(tpRxDataLength);
    break;
  default:
    break;
  }
}

void ApplTpRxErrorIndication(vuint8 tpChannel, vuint8 status)
{
  switch(TpRxGetConnectionNumber(tpChannel))
  {
  case kDiagConnection:
    DiagRxErrorIndication(tpChannel, status);
  case CONNECTION_0:
    UserTpRxErrorIndication(status);
  default:
    break;
  }
}

void ApplTpRxFF(vuint8 tpChannel)
{
  if (TpRxGetConnectionNumber( tpChannel ) == kDiagConnection )
  {
    DiagRestartS1TimerInternal( tpChannel );
  }
}

void ApplTpRxCF(vuint8 tpChannel)
{
  if (TpRxGetConnectionNumber( tpChannel ) == kDiagConnection )
  {
    DiagRestartS1TimerInternal( tpChannel );
  }
}


void ApplTpTxConfirmation(vuint8 tpChannel, vuint8 state)
{
  switch(TpTxGetConnectionNumber(tpChannel))
  {
  case kDiagConnection:
    DiagConfirmation( tpChannel, state);
  case CONNECTION_0:
    UserTpConfirmation(status);
  default:
    break;
  }
}
```

```
vuint8 ApplTxErrorIndication(vuint8 tpChannel, vuint8 status)
{
  switch(TpTxGetConnectionNumber(tpChannel))
  {
  case kDiagConnection:
    return DiagTxErrorIndication(tpChannel, status);
  case CONNECTION_0:
    UserTpTxErrorIndication(status);
  default:
    return kTpFreeChannel;
  }
}

vuint8 ApplCopyToCAN(TpCopyToCanInfoStructPtr infoStruct)
{
  switch(TpTxGetConnectionNumber(infoStruct->Channel))
  {
  case kDiagConnection:
    return DiagCopyToCAN(infoStruct->Channel, kSFDataPos, tpTxDataLength);
  default:
    (void)memcpy( infoStruct->pDestination, infoStruct->pSource, infoStruct->Length);
    break;
  }
  return 0;
}

void ApplTpTxNotification(vuint8 tpChannel, vuint8 DataLength)
{
  switch(TpTxGetConnectionNumber(tpChannel))
  {
  case kDiagConnection:
    DiagTpMsgTxReady(tpChannel, DataLength);
    break;
  default:
    break;
  }
}
```

8.6 How to Lock a Tx-Channel and Why? (only dynamic channels)

Normally the application get a resource – use the resource – and release the resource. In the current version the resource Transmit-tpChannel will be released by the Transport Layer automatically after a transmission (for code optimization). If an application will use the same channel more than one time (i.e. a periodically transmission) it has to lock the channel.

```
...
...
TpTxLockChannel(channel);
TpTransmit(...)
...
TpTransmit(...)

...
TpTransmit(...)
```

The application has two possibilities to release the channel:

> unlock the channel using ‘**TpTxUnlockChannel ()**’: i.e. only one transmission without a release should be done...

```
TpTxLockChannel(user_channel);
TpTransmit(user_channel, ...)
...
>wait until confirmation occured<
TpTxUnlockChannel(user_channel);
TpTransmit(user_channel, ...)
/* After this transmission the channel will be released */
...
```

> release the channel using ‘**TpTxResetChannel()**’: Lock the resource for many transfers as long as used

```
...
...
TpTxLockChannel(channel);
TpTransmit(...)
...
TpTransmit(...)
...
TpTxResetChannel(channel);
...
```

8.7 How to transmit a ConsecutiveFrame as quick as possible

Typically this requirement is used for a fast re-programming of ECU’s through Gateways or Testers.
How to do that, please refer to chapter 7.2 Fast transmission of ConsecutiveFrames.

9 Contact

Visit our website for more information on

> News
> Products
> Demo software
> Support
> Training data
> Addresses

www.vector.com