BOSCH
Invented for life

XS_CAN

Slim CAN XL IP module

Transmission Handling Cut Through Mode (CTM)

Application Note XS_CAN_AN004

Document Revision 1.0
23.07.2026

Robert Bosch GmbH
Mobility Electronics

LEGAL NOTICE

The content of this document is subject to continuous developments and improvements. All particulars and its use contained in this document are given by BOSCH in good faith.

NO WARRANTIES: TO THE MAXIMUM EXTENT PERMITTED BY LAW, NEITHER THE INTELLECTUAL PROPERTY OWNERS, COPYRIGHT HOLDERS AND CONTRIBUTORS, NOR ANY PERSON, EITHER EXPRESSLY OR IMPLICITLY, WARRANTS ANY ASPECT OF THIS SPECIFICATION, SOFTWARE RELATED THERETO, CODE AND/OR PROGRAM RELATED THERETO, INCLUDING ANY OUTPUT OR RESULTS OF THIS SPECIFICATION, SOFTWARE RELATED THERETO, CODE AND/OR PROGRAM RELATED THERETO UNLESS AGREED TO IN WRITING. THIS SPECIFICATION, SOFTWARE RELATED THERETO, CODE AND/OR PROGRAM RELATED THERETO IS BEING PROVIDED "AS IS", WITHOUT ANY WARRANTY OF ANY TYPE OR NATURE, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, AND ANY WARRANTY THAT THIS SPECIFICATION, SOFTWARE RELATED THERETO, CODE AND/OR PROGRAM RELATED THERETO IS FREE FROM DEFECTS.

ASSUMPTION OF RISK: THE RISK OF ANY AND ALL LOSS, DAMAGE, OR UNSATISFACTORY PERFORMANCE OF THIS SPECIFICATION (RESPECTIVELY THE PRODUCTS MAKING USE OF IT IN PART OR AS A WHOLE), SOFTWARE RELATED THERETO, CODE AND/OR PROGRAM RELATED THERETO RESTS WITH YOU AS THE USER. TO THE MAXIMUM EXTENT PERMITTED BY LAW, NEITHER THE INTELLECTUAL PROPERTY OWNERS, COPYRIGHT HOLDERS AND CONTRIBUTORS, NOR ANY PERSON EITHER EXPRESSLY OR IMPLICITLY, MAKES ANY REPRESENTATION OR WARRANTY REGARDING THE APPROPRIATENESS OF THE USE, OUTPUT, OR RESULTS OF THE USE OF THIS SPECIFICATION, SOFTWARE RELATED THERETO, CODE AND/OR PROGRAM RELATED THERETO IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, BEING CURRENT OR OTHERWISE. NOR DO THEY HAVE ANY OBLIGATION TO CORRECT ERRORS, MAKE CHANGES, SUPPORT THIS SPECIFICATION, SOFTWARE RELATED THERETO, CODE AND/OR PROGRAM RELATED THERETO, DISTRIBUTE UPDATES, OR PROVIDE NOTIFICATION OF ANY ERROR OR DEFECT, KNOWN OR UNKNOWN. IF YOU RELY UPON THIS SPECIFICATION, SOFTWARE RELATED THERETO, CODE AND/OR PROGRAM RELATED THERETO, YOU DO SO AT YOUR OWN RISK, AND YOU ASSUME THE RESPONSIBILITY FOR THE RESULTS. SHOULD THIS SPECIFICATION, SOFTWARE RELATED THERETO, CODE AND/OR PROGRAM RELATED THERETO PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL LOSSES, INCLUDING, BUT NOT LIMITED TO, ANY NECESSARY SERVICING, REPAIR OR CORRECTION OF ANY PROPERTY INVOLVED TO THE MAXIMUM EXTEND PERMITTED BY LAW.

Revision History

Version	Date	Remark
1.0	27.02.2026	Initial version for XS_CAN 1.1.0

Conventions

The following conventions are used within this document:

Register names	`TXFQ_LMEM_SA, TXFQ_CFG`
Names of files and directories	`directoryname/filename`
Source code/function names	`xs_can_mh_tx_fifo_queue_enqueue_msg()`

References

This document refers to the following documents:

Ref	Author	Title
[1]	MS/EEJ	XS_CAN User's Manual
[2]	MS/EEJ	XS_CAN System Integration Guide
[3]	MS/EEJ	CTDMA User's Manual
[4]	MS/EEJ	CTDMA System Integration Guide
[5]	MS/EEJ	calc_min_host_clk_freq_for_XS_CAN.xlsx
[6]	MS/EEJ	XS_CAN_Local_Memory_size_calculator.xlsx
[7]	ME-IC/PAI	XS_CAN Application Note 002 TX Handling FMM

Terms and Abbreviations

This document uses the following terms and abbreviations:

Term	Meaning
CAN	Controller Area Network
CAN CC	Controller Area Network Classical CAN
CAN FD	Controller Area Network with Flexible Data Rate
CAN XL	Controller Area Network with Extended frame Length
CTB	Cut Through Buffer
CTDMA	Cut Through Direct Memory Access
CTM	Cut Through Mode
DLC	Data Length Code
FMM	Full Message Mode
HOST	This is the CPU which is hosting the X_CAN
IP	Intellectual property
IRC	Interrupt Controller
LMEM	Local Memory
MH	Message Handler

Term	Meaning
PRT	Protocol Controller
PWM	Pulse Width Modulation
RAM	Random Access Memory
RX	Receive
RXFQ	Receive FIFO Queue
SMEM	System Memory
TEFQ	Transmit Event FIFO Queue
TEQE	Transmit Event FIFO Queue Element
TS	Timestamp
TX	Transmit
TXEF	Transmit Event FIFO
TXPQ	Transmit Priority Queue
TXFQ	Transmit FIFO Queue
TXQE	Transmit FIFO Queue Element
VBM	Virtual Buffer Manager
XS_CAN	Slim CAN XL IP

Table of Contents

List of Figures:

List of Tables:

1 Target

This application note describes transmit message handling in the **XS_CAN versions 1.1.0 with CTM.**

The topics covered include:

- Local Memory configuration for Queues (TXFQ, TXPQ, CTB)
- TX Message enqueue for transmission (TXFQ, TXPQ)

Note: The configuration and use of TX Event FIFO Queue has no difference between FMM and CTM, please see more information [7]

Important Note:

The software examples delivered with this application note are for illustration purposes only. Use the examples at your own risk.

2 Overview of CTDMA and CTM data path

2.1 CTMA block diagram

CTDMA is an **Add On Module** to the XS_CAN controller IP which enables the XS_CAN IP to perform cut through block copy operations thereby offloading the task from system DMA controllers. 4 instances of XS_CAN IPs can be connected to one CTDMA module and the data transfer from each XS_CAN IP is redirected back and forth to the **system memory (SMEM)** during block copy transfer by CTDMA. Apart from block copy transfers, CTDMA also supports host accesses to the **LMEM space** of XS_CAN IP for enqueue, dequeue, CREG read write etc via the AHB slave interface in CTDMA.

Key features:
-AXI master interface for block copy transfer between SMEM and XS_CAN IPs
-AHB slave interface for host access to XS_CAN IP.
-AHB Master interface per XS_CAN IP
-Supports until 4 x XS_CAN IPs.
-Support burst transfers at all interfaces.
-One internal buffer of 64byte in CT_MANAGER.
-Internal arbiter to prioritize between host access and Cut through transfers
-AHB-APB bridge to convert CREG access at AHB slave to APB interface in CREG.

The following block diagram shows the CTDMA.

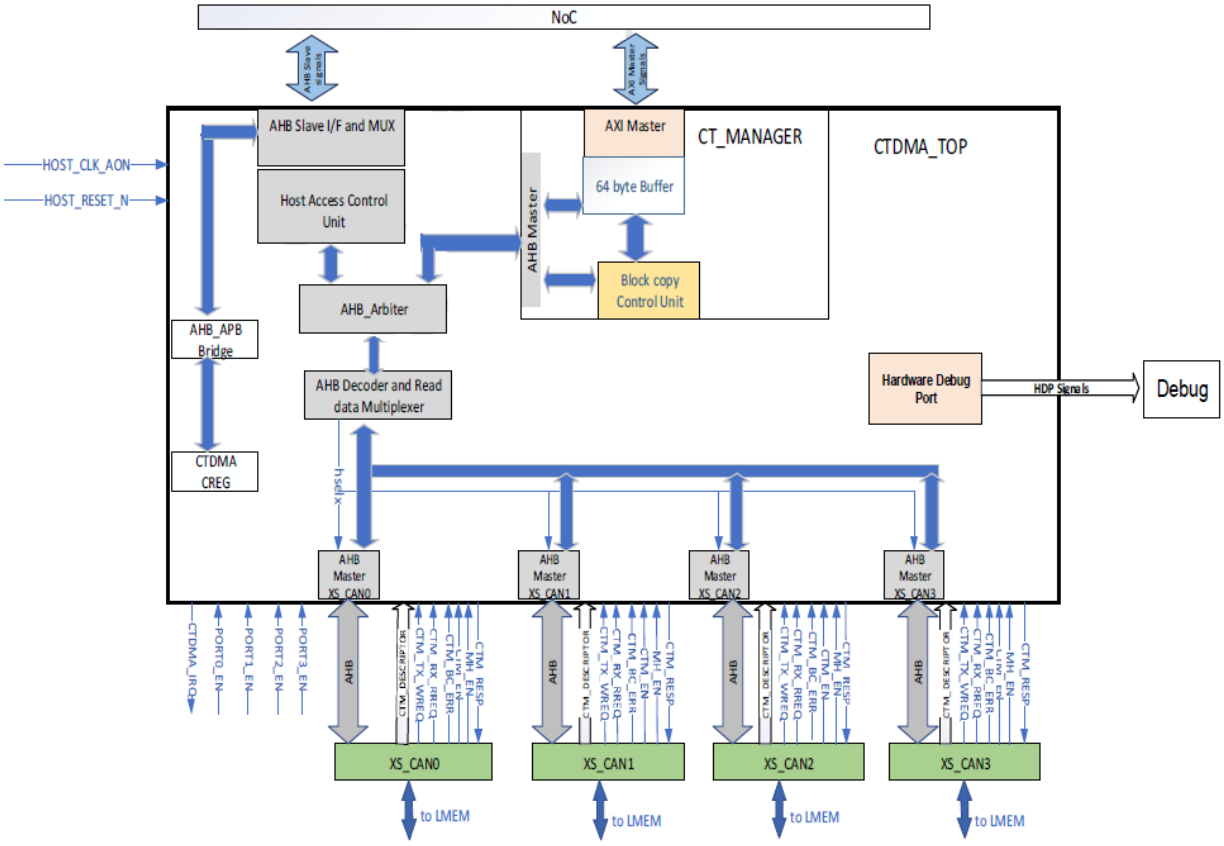

Figure 1: CTMDA Block Diagram

See more information about CTDMA functions in the CTDMA User manual [3], chapter 1.3.3 and 1.4.

2.2 CTDMA data flow paths

The following block diagram shows the data flow paths in CTM.

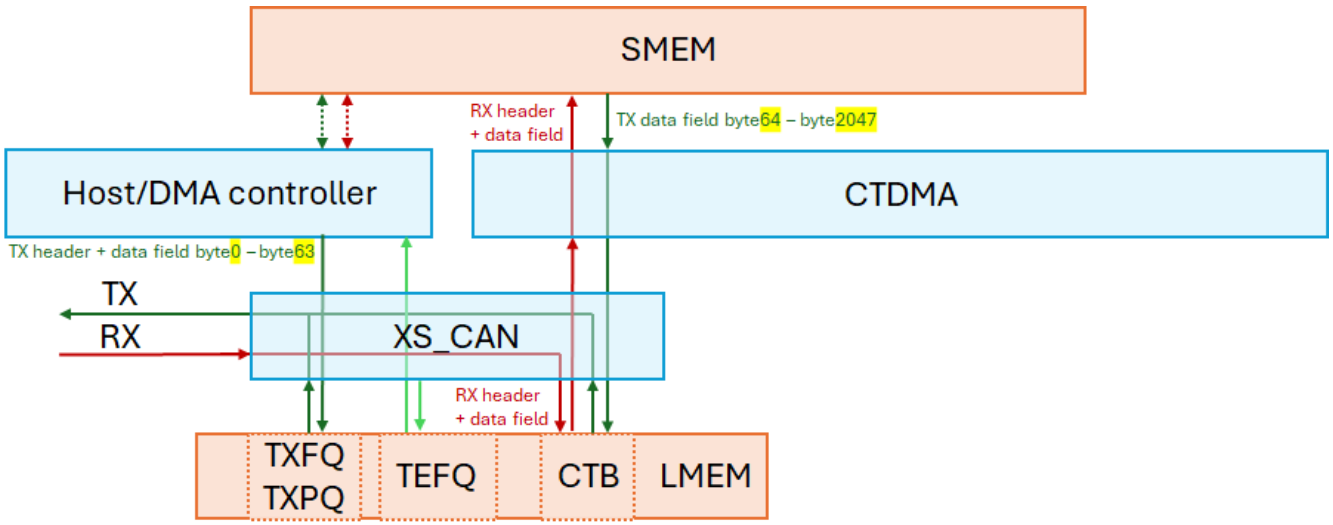

Figure 2: TX and RX relevant data flow path in CTM

2.3 System overview of XS_CAN with CTDMA

The following block diagram shows the XS_CAN with CTDMA. Maximum 4 XS_CAN´s cam be used with one CTDMA.

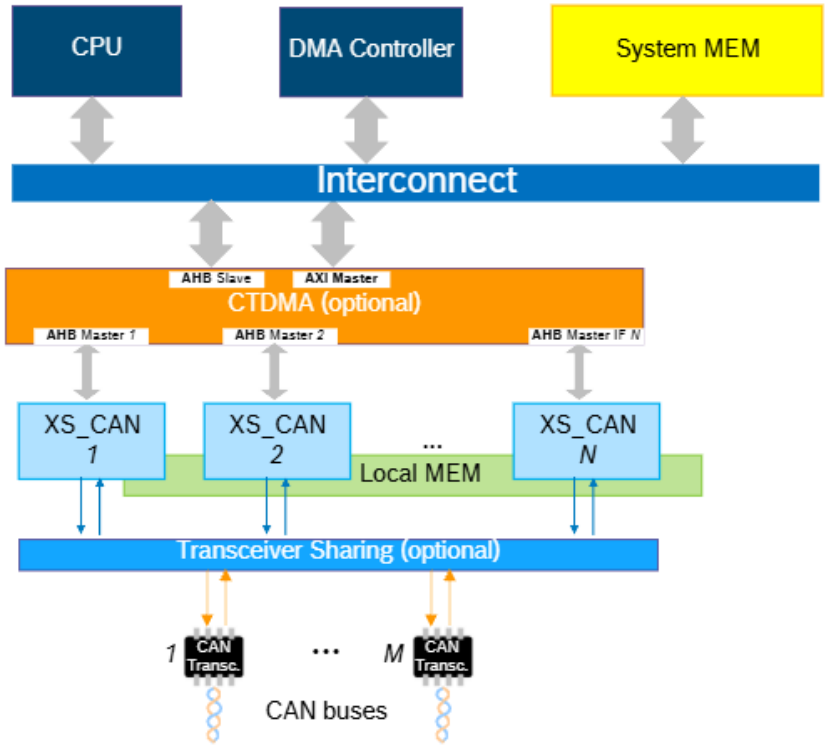

Figure 3: XS_CAN with CTDMA

3 Memory map overview

The XS_CAN instance has the following memory map.

Start address	End address	Symbol	Name
0x00000	0x008FC	MH reg	Message Handler register.
0x00900	0x009FC	PRT reg	Protocol Controller registers
0x00A00	0x00AFC	IRC reg	Interrupt Controller registers
0x00B00	0x00FFC	reserved	reserved
0x01000	0x01808	TXFQ	FIFO Queue
0x0180C	0x01FFC	reserved	reserved
0x02000	0x02808	TXPQ	TX Priority Queue
0x0280C	0x02FFC	reserved	reserved
0x03000	0x03810	RXFQ0	RX FIFO 0 Queue
0x03814	0x03FFC	reserved	reserved
0x04000	0x04810	RXFQ1	RX FIFO 1 Queue
0x04814	0x04FFC	reserved	reserved
0x05000	0x05010	TEFQ TX	TX Event FIFO Queue
0x05014	0x05FFC	reserved	reserved
0x06000	0x0603C	CTB	Cut Through Buffer
0x06040	0x3FFFC	reserved	reserved
0x40000	0x7FFFC	LMEM TA	LMEM Transparent Access

Table 1: XS_CAN memory map

The CTDMA memory map with 4 XS_CAN has the following memory map.

Start address	End address	Symbol	Name
0x000000	0x07FFFC	XSCAN 0	XSCAN 0 memory map
0x080000	0x0FFFFC	XSCAN 1	XSCAN 1 memory map
0x100000	0x17FFFC	XSCAN 2	XSCAN 2 memory map
0x180000	0x1FFFFC	XSCAN 3	XSCAN 3 memory map
0x200000	0x200B0C	CTDMA reg	Cut- Through DMA registers
0x200B10	0x3FFFFC	Reserved	Reserved address

Table 2: CTDMA memory map

See also as reference [3] for more info and related CTDMA registers.

4 LMEM configuration in CTM and interface

The XS_CAN in CTM requires **external local memory** (**LMEM**) and **SMEM** to store all messages (transmitted and received), TX Event information, and RX filter elements. The LMEM size is calculated with the **XS_CAN LMEM size calculator Excel sheet** [6]. The LMEM size for CTM is much lower as the LMEM size for FMM. You need to calculate the needed LMEM size with the XS_CAN LMEM size calculator Excel sheet [6]. In case for the CTM you need to configure the with CTM selector in the Excel sheet [6]. There are already some example configurations for FMM and CTM.

Up to 256KB of LMEM is accessible by a single XS_CAN IP instance via the LMEM interface. The address width is 18 bits, and the data width is one 32-bit word. Data is stored into LMEM as words (32 bits). The following Figure 4 shows a LMEM configuration example.
.

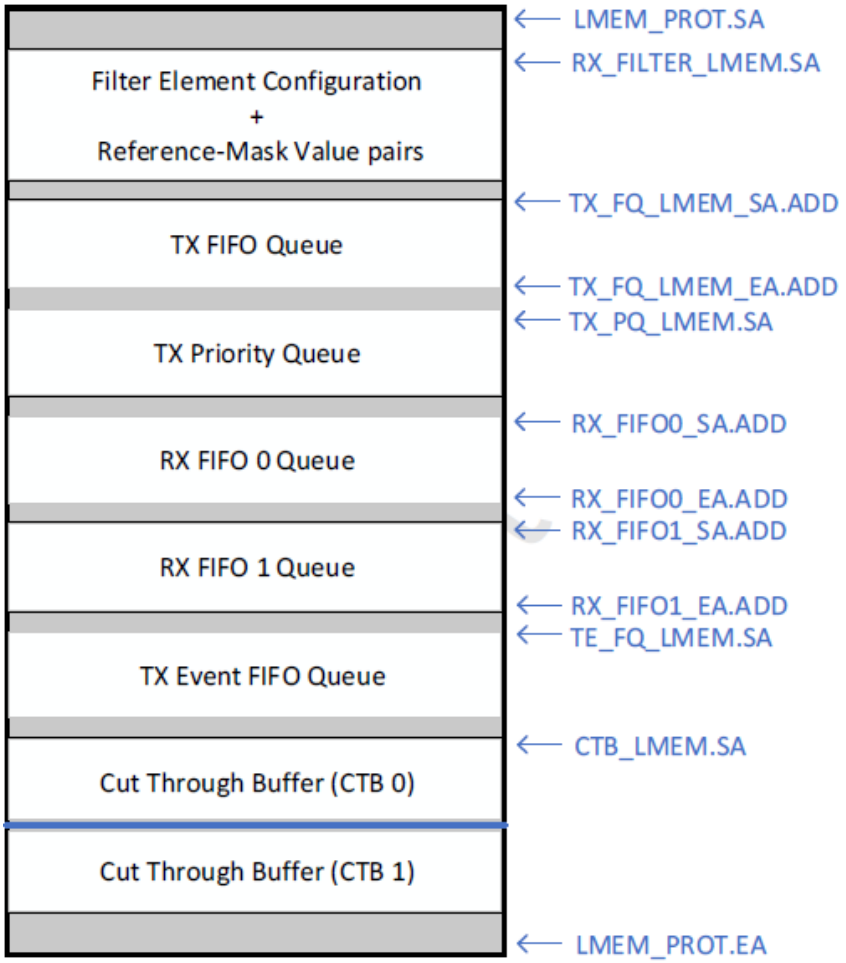

Figure 4: LMEM configuration diagram example

The XS_CAN provides a **“LMEM protection configuration”** (**LMEMPC**) - feature to configure LMEM protection for all accesses to the LMEM. XS_CAN allows accesses to LMEM only in the range of start and end addresses.

The **start address** (**LMEMPC_START_ADDR**) and the **end address** (**LMEMPC_END_ADDR**) are 18-bit width and must be configured in **64-byte** granularity. These are static inputs and must be provided before configuring and starting the XS_CAN.

Important hint: The start address of every Queue must be **64-byte aligned**. Therefore, the lower 6 bits (5:0) of the 18-bit address are not used and are treated as '0'.
XS_CAN allows accesses to LMEM only in the range of start and end addresses. Both addresses are included in this range. Since the start and end addresses are in 64-byte granularity an access is granted in the following address range: **LMEMPC_START_ADDR to LMEMPC_END_ADDR**. This granularity introduces limitation where the entire LMEM space may not be fully utilized. For example, for a 4KB memory with start address 0x000, the usable range would be from 0x00 to 0xFC0 (included). Any access to an address greater than 0xFC0 will result in the setting of an **error event flag** and **triggering of an interrupt**. Depending on the source of access (host, TX Handler, RX handler, Transparent access), behaviour of the IP is different. Refer Section 1.5.8 Error and Exception handling in [1] for details.

4.1 Storage concepts

The table show the different Storage concepts for FMM and CTM mode.

	FMM	CTM
TXFQ/TXPQ	LMEM	LMEM (Messages partly) SMEM (Messages full)
TEFQ	LMEM	LMEM
RXFQ	LMEM	SMEM

Table 3: Storage concept comparison

The following Figure 5 shows the FMM storage configuration. In FMM the SMEM is not used by the XS_CAN.

Full Message Mode (FMM)

- **LMEM contains: Full** message
- **SMEM contains:** <not used>
- **SMEM ⇔ LMEM data transfer:** done by host for enqueue/dequeue
- **Use Case: CC, FD, XL** with **short** messages (≤ 500-byte data field)

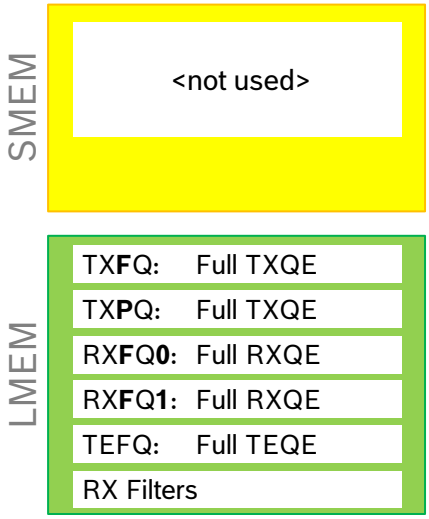

Figure 5: Storage concept FMM

The following figure shows the **CTM storage configuration**. In CTM-mode the SMEM is used for all full CAN Messages and LMEM for CAN Messages partly, CTM Buffers, TX-Event FIFO and RX-Filters.

Cut-Through Mode (CTM)

- **LMEM contains:** TX Header + **first 64 byte** of data field
- **SMEM contains: Full** messages
- **SMEM ⇔ LMEM data transfer** done by **CTDMA** during TX & RX
- **Use Case: XL** with **long** messages (> 500-byte data field)

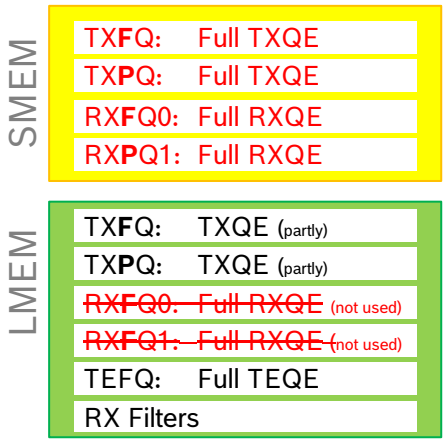

Figure 6: Storage concept CTM

4.2 CTM

All Queues must be located within this CTM memory space. See LMEM Access Protection chapter in [1] for more information.

In **Cut Through Mode (CTM),** there are 2 scenarios possible

1. **Scenario:** The TX message with payload (data) ≤ 64 bytes.
   - Classic CAN

- CAN FD
- CAN XL, payload ≤ 64 bytes

The TX Message Header + the payload are stored in/located into the **TXFQ** and/or the **TXPQ** depending on the use case of the application.

2. **Scenario:** The TX message with payload (data) > 64 bytes
   - CAN XL, payload > 64 bytes

   The TX Message Header + the first 64 bytes of Message Payload are stored in and located into the **TXFQ** and/or the **TXPQ** depending on the use case of the application.

   The remainder of the TX-Message Payload **(>64 bytes)** is automatically copied from **System Memory (SMEM)** to the **Cut Through Buffer** (CTB) during message transmission by the **Cut Through Direct Memory Access** (CTDMA) module. This copy is always 64 bytes at a time.

XS_CAN User manual [1], chapter 1.5.6.1 TX Message Header Definition, describes the TX Message Header's structure and how it must be stored in the TXFQ/TXPQ. The Message Header's data structure depends on the CAN Frame Format (Classical CAN, CAN FD and CAN XL) and the mode of operation (FMM or CTM).

Note:

- For **Classical CAN and CAN FD**, the TX Message Header's structure of the have **no difference** between FMM and CTM.
- For **CAN XL**, the **TX Message Header word3** (**T3**) is added in CTM. T3 is SMEM 32-bit address pointer, which points to **the first payload word** of the TX message on the SMEM

The XS_CAN supports single **TXFQ with up to 255 messages** and up **to 32 TXPQ slots**.

The size of the CTB on the LMEM has a fixed **size of 128 bytes**, consisting of 2 contiguous buffers with 64 bytes.

4.3 Queue memory space in LMEM configuration

The size and location of Queues in LMEM can be configured via their respective configuration registers, depending on the Queue type. See [1] for more information.

Hint:
The maximum size of TX message varies according to its CAN frame format.

See the following Table 4 with the overview of TX message frame format and size, and Table 5: Overview of TX message frame format, size and storage location for their storage location

	TX Message				total size of one message	
	Header		maximum Payload			
	(byte)	(word)	(byte)	(word)	(byte)	(word)
CC	8 (T0,T1)	2 (T0,T1)	8	2	16	4
FD	8 (T0,T1)	2 (T0,T1)	64	16	72	18
XL	16 (T0,T1,T2,T3)	4 (T0,T1,T2,T3)	2048	512	2064	516

Table 4: Overview of TX message frame format and size

	TX Message				total size of one message		storage location
	Header		maximum Payload				
	(byte)	(word)	(byte)	(word)	(byte)	(word)	
CC	8 (T0,T1)	2 (T0,T1)	8	2	16	4	TXFQ/ TXPQ
FD	8 (T0,T1)	2 (T0,T1)	64	16	72	18	
XL	16 (T0,T1,T2,T3)	4 (T0,T1,T2,T3)	64	16	80	20	
			31x64	31x16	31x64	31x16	SMEM ➡CTB

Table 5: Overview of TX message frame format, size and storage location

4.4 TXFQ (TX FIFO Queue)

The XS_CAN supports one **TX FIFO Queue (TXFQ)** defined in LMEM. A TX FIFO Queue holds TX messages to be sent in order to the PRT. Each message inside TXFQ is called a TX Queue Element (TXQE). Each TXQE consists of header and a payload. See more information in XS_CAN User manual [1], chapter TX Message Header Definition.

The TXFQ is enabled when **MH_CFG.TXFQE** is set to 1. The XS_CAN provides register **TXFQ_LMEM_SA.ADD** for configuring the start address and **TXFQ_LMEM_EA.ADD** for configuring the end address of TXFQ in LMEM.

See the following parameter overview:

TXFQ_LMEM_SA.ADD	defines the TXFQ's start addresses within LMEM.
TXFQ_LMEM_EA.ADD	defines the TXFQ's end addresses within LMEM.

Their values must always be **64-byte aligned** address; therefore, only the **higher 12 bits (17:6)** of the 18-bit address are considered.

Example: For a TXFQ size of 64 bytes, if TXFQ_LMEM_SA.ADD = 0x0000, then **TXFQ_LMEM_EA.ADD** shall be programmed as 0x0040 (64byte aligned boundary) and not 0x003C. The next queue, if configured, shall start from the next 64-byte aligned boundary which is 0x0080. The memory space from 0x0044 to 0x007C will remain unused in this example.

TXFQ_CFG.MAX_ELEM_SIZE[5:0] is the TX FIFO Configuration register which defines the maximum size of 1 TXQE which includes the header and payload. The total size of TXFQ should be greater than or equal to **TXFQ_CFG.MAX_ELEM_SIZE** else it is not considered as a valid configuration. Also, XS_CAN IP does not generate TXFQ_WREQ if MAX_ELEM_SIZE=0.

The status registers for TXFQ are given as:

- **TXFQ_STS.FIFO_FULL**: When set to 1, indicates TXFQ is full. Empty conditions can be detected by reading the fill level register TXFQ_STS.FILL_LEVEL
- **TXFQ_STS.FILL_LEVEL:** It is a read-only register which stores the fill level of TXFQ.
- **TXFQ_WPTR.WPTR_ADD:** The TX FIFO Queue Write Pointer register TXFQ_WPTR.WPTR_ADD points to the location where the last word of TXQE is enqueued and the next TXQE starts at address wptr+4.
  TXFQ_RPTR.RPTR_ADD: Points to the location corresponding to the last word of TXQE is read by TX Handler for transmission to PRT.

See more info about TXFQ in [1], chapter 1.5.6.4.2 “TXFQ”.

Example configuration for TXFQ

In this configuration example, the TXFQ starts at address 0x02000 (byte address) in the LMEM.

Number of messages in TXFQ	**Maximum message length**	**Calculation and configuration** **See following Note:** (CC, FD = 8-byte Header, XL= 16-byte Header) (Required Size = Nr Message * (Header + Payload) **when ≤ 64-byte Payload** (Required Size = Nr Message * (Header + (Payload = 64 byte)) **when > 64-byte Payload** (**TXFQ_LMEM_SA.ADD**= SA / 64) (**TXFQ_LMEM_EA.ADD**= EA / 64)
10	Classical CAN 8-byte payload	required size = 160 bytes SA = 0x02000 (byte address) EA = 0x0209F (byte address) (-> 0...159) byte configurable size = 192 bytes (multiple of 64 bytes) *SA = 0x02000 (64-byte aligned address)* *EA = 0x020C0 (64-byte aligned address)* **TXFQ_LMEM_SA.ADD = 0x0080** **TXFQ_LMEM_EA.ADD = 0x0083**
10	CAN FD 8-byte payload	required size = 160 bytes SA = 0x02000 (byte address) EA = 0x0209F (byte address) (-> 0...159) byte configurable size = 192 bytes (multiple of 64 bytes) *SA = 0x02000 (64-byte aligned address)* *EA = 0x020C0 (64-byte aligned address)* **TXFQ_LMEM_SA.ADD = 0x0080** **TXFQ_LMEM_EA.ADD = 0x0083**
10	CAN FD 64-byte payload	required size = 720 bytes SA = 0x02000 (byte address) EA = 0x022CF (byte address) (-> 0...719) byte configurable size = 768 bytes (multiple of 64 bytes) *SA = 0x02000 (64-byte aligned address)* *EA = 0x02300 (64-byte aligned address)* **TXFQ_LMEM_SA.ADD = 0x0080** **TXFQ_LMEM_EA.ADD = 0x008C**
10	CAN XL 32-byte payload	required size = 480 bytes SA = 0x02000 (byte address) EA = 0x021DF (byte address) (-> 0...479) byte configurable size = 512 bytes (multiple of 64 bytes) *SA = 0x02000 (64-byte aligned address)* *EA = 0x02200 (64-byte aligned address)* **TXFQ_LMEM_SA.ADD = 0x0080** **TXFQ_LMEM_EA.ADD = 0x0088**
10	CAN XL 512-byte payload	required size = 800 bytes (only Message Headers + first 64 bytes of Payload) SA = 0x02000 (byte address) EA = 0x0231F (byte address) configurable size = 832 bytes (multiple of 64 bytes) *SA = 0x02000 (64-byte aligned address)* *EA = 0x02340 (64-byte aligned address)* **TXFQ_LMEM_SA.ADD = 0x0080** **TXFQ_LMEM_EA.ADD = 0x008D**

Number of messages in TXFQ	Maximum message length	Calculation and configuration **See following Note:** (CC, FD = 8-byte Header, XL= 16-byte Header) (Required Size = Nr Message * (Header + Payload) **when ≤ 64-byte Payload** (Required Size = Nr Message * (Header + (Payload = 64 byte)) **when > 64-byte Payload** (**TXFQ_LMEM_SA.ADD**= SA / 64) (**TXFQ_LMEM_EA.ADD**= EA / 64)
10	CAN XL 2048-byte payload	required size = 800 bytes (only Message Headers + first 64 bytes of Payload) SA = 0x02000 (byte address) EA = 0x0231F (byte address) configurable size = 832 bytes (multiple of 64 bytes) *SA = 0x02000 (64-byte aligned address)* *EA = 0x02340 (64-byte aligned address)* **TXFQ_LMEM_SA.ADD = 0x0080** **TXFQ_LMEM_EA.ADD = 0x008D**

Table 6: TXFQ example configuration

4.5 TXPQ (TX Priority Queue)

The XS_CAN IP supports **one TX Priority Queue (TXPQ).** It supports up to 32 priority queue slots, and the number of slots is user configurable. See more information in XS_CAN User manual [1].

The XS_CAN provides a register **TXPQ_LMEM.SA** for configuring the TXPQ start address in the LMEM. The address value is always 64-byte aligned.

The TXPQ Configuration register provides **TXPQ_CFG.SLOT_NUM** for configuring the number of slots in TXPQ. The user can store up to 32, and **TXPQ_CFG.SLOT_SIZE** to define the size of a TXPQ slot.

See the following parameter overview:

TXPQ_LMEM.SA	defines the TXPQ's start addresses within LMEM.
TXPQ_CFG. SLOT_SIZE	defines the size of a TXPQ slot in 32-bit word.
TXPQ_CFG.SLOT_NUM	defines the number of TXPQ slots, up to 32.

The size of the TXPQ in LMEM is = SLOT_NUM x SLOT_SIZE

The start address of **Slot n** where n∈{1,2,3,...,32} in TXPQ is calculated as:
*(TXPQ_LMEM.SA) + (n * TXPQ_CFG.SLOT_SIZE)*

End address of each slot (n) is Start address of slot (n+1) - 4.

Note: In CTM, the maximum SLOT_SIZE shall be 20 words, see Table 5 for details.

See following static registers for TXPQ, which are relevant:

- **TXPQ_STS0.SLOT_OCC:** When SLOT_OCC[n] = 1, the TXPQ slot n is busy which means that the TX message in slot n is being loaded in LMEM and

considered by TX-SCAN. The message attached to the slot n has not been sent yet as long as this bit remains high.

- **TXPQ_STS1.TXPQ_FULL:** When set to 1, indicates all slots are occupied and cannot store any new message. Empty condition of TXPQ can be detected by reading the fill level register TXPQ_STS1.FILL_LEVEL.
- **TXPQ_STS1.FILL_LEVEL:** It defines the total number of messages in the TXPQ pending for transmission. The maximum allowed fill level is same as the number of slots configured by the user. Once a message is successfully enqueued, TXPQ_STS1.FILL_LEVEL is incremented by 1 and decremented by 1 when a message is successfully dequeued. Once maximum fill level is attained, no more elements are enqueued. Status register fields are updated one clock after the interrupt TX functional interrupt TXPQ_ENQ gets generated from MH

- **TXPQ_WPTR.WPTR_ADD:** This is TXPQ Write Pointer register which indicates the LMEM address corresponding to the last word of the previously enqueued TXQE.

Example configuration for TXPQ

In this configuration example, the TXPQ starts at address 0x03000 (byte address) in the LMEM.

Number of slots in TXPQ	**Maximum message length**	**Calculation and configuration** **See following note:** (CC, FD = 8-byte Header, XL= 16-byte Header) (Required Slot Size = Header + Payload) **when ≤ 64-byte Payload** (Required Slot Size = Header + (Payload = 64byte)) **when > 64-byte Payload** (**TXPQ_CFG. SLOT_SIZE**= Required Slot Size / 4) (TXPQ size= Slot_num * Slot_size)
10	Classical CAN 8-byte payload	required slot size = 16 bytes configurable slot size = 4 words *SA = 0x03000 (64-byte aligned address)* **TXPQ_LMEM.SA = 0x00C0** **TXPQ_CFG. SLOT_SIZE = 4** **TXPQ_CFG.SLOT_NUM = 10** total TXPQ size = 40 words
10	CAN FD 64-byte payload	required slot size = 72 bytes configurable slot size = 18 words *SA = 0x03000 (64-byte aligned address)* **TXPQ_LMEM.SA = 0x00C0** **TXPQ_CFG. SLOT_SIZE = 18** **TXPQ_CFG.SLOT_NUM = 10** total TXPQ size = 180 words
10	CAN XL 32-byte payload	required slot size = 48 bytes configurable slot size = 12 words *SA = 0x03000 (64-byte aligned address)* **TXPQ_LMEM.SA = 0x00C0** **TXPQ_CFG. SLOT_SIZE = 12** **TXPQ_CFG.SLOT_NUM = 10** total TXPQ size = 120 words

Number of slots in TXPQ	Maximum message length	Calculation and configuration **See following note:** (CC, FD = 8-byte Header, XL= 16-byte Header) (Required Slot Size = Header + Payload) **when ≤ 64-byte Payload** (Required Slot Size = Header + (Payload = 64byte)) **when > 64-byte Payload** (**TXPQ_CFG. SLOT_SIZE**= Required Slot Size / 4) (TXPQ size= Slot_num * Slot_size)
10	CAN XL 512-byte payload	required slot size = 80 bytes (Note: only Message Headers + first 64 bytes Payload) configurable slot size = 20 words *SA = 0x03000 (64-byte aligned address)* **TXPQ_LMEM.SA = 0x00C0** **TXPQ_CFG. SLOT_SIZE = 20** **TXPQ_CFG.SLOT_NUM = 10** total TXPQ size = 200 words
10	CAN XL 2048-byte payload	required slot size = 80 bytes (Note: only Message Headers + first 64 bytes Payload) configurable slot size = 20 words *SA = 0x03000 (64-byte aligned address)* **TXPQ_LMEM.SA = 0x00C0** **TXPQ_CFG. SLOT_SIZE = 20** **TXPQ_CFG.SLOT_NUM = 10** total TXPQ size = 200 words

Table 7: TXPQ example configuration

4.6 CTB (Cut Through Buffer)

The XS_CAN IP supports **2 Cut Through Buffers (CTB0 and CTB1).** In TX direction, one CTB contains **64 bytes** of the CAN XL payload beyond 64 bytes (byte128 – byte191, ..., byte1984 – byte2047). This size of transfer is always in 64-byte blocks and is called **block copy transfer** in TX direction. For TX, the block copy is from SMEM to LMEM. See following figure with the TX Message in CTM.

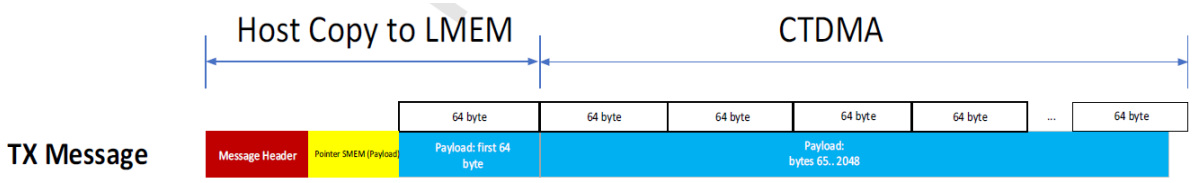

Figure 7: TX Message in CTM

In CTM, the host does not need to copy the data from SMEM to CTB, this is done by CTDMA.

The XS_CAN provides a register **CTB_LMEM.SA** for configuring the **CTB start address** where the CTB are defined in LMEM. The address is always 64-byte aligned.

The size of CTB on the LMEM is always **128 bytes** (2 x CTBs x 64 bytes)

See the following parameter overview:=======

CTB_LMEM.SA	defines the CTB's start addresses within LMEM.

Example configuration for CTB

In this configuration example, the CTB starts at address 0x07000 (byte address) in the LMEM.

CTB_LMEM.SA = 0x01C0 (0x7000/64)

5 Virtual Buffer Manager

The **Virtual Buffer Manager (VBM)** in MH handles the conversion of virtual addresses to physical LMEM addresses. It manages the enqueuing of messages into **TXFQ**, **TXPQ** slots and **CTB**, as well as the dequeuing of messages from **CTB** and the **TEFQ**.

The host can access the full 256KB LMEM address space using **LMEM Transparent Access**, **LMEM TA**, the address range 0x40000 to 0x7FFFC (the byte address bit 18 (the 19th bit) is '1').

However, the actual memory space which is allocated to an XS_CAN IP is indicated in **LMEM_PROT.SA** (Start Address of LMEM space) and **LMEM_PROT.EA** (End Address of LMEM space). Transparent accesses outside this range of start and end address will generate **SAFETY_RAW.MH_LMEM_PROT_ERR** interrupt. See LMEM Access Protection chapter [1] for more information.

The host interacts with LMEM via the VBM interface. To simplify the LMEM access for the host, virtual buffers are set up for each Queue type: **TXFQ, TXPQ, TEFQ and CTB**. The address range for these virtual buffers is fixed, matching the size of the largest message that can be stored in any given Queue.

All virtual addresses are **64-byte aligned**. Each virtual buffer has its own **start, trigger, and end addresses**. The XS_CAN-CTM's virtual buffer address map is shown in the following table.

Start Location Address	End Location Address	Symbol	Name
0x01000	0x01808	TXFQ	TX FIFO Queue
0x0180C	0x01FFC	reserved	reserved
0x02000	0x02808	TXPQ	TX Priority Queue
0x0280C	0x02FFC	reserved	reserved
0x03000	0x03810	RXFQ0	RX FIFO 0 Queue (reserved in CTM)
0x03814	0x03FFC	reserved	reserved
0x04000	0x04810	RXFQ1	RX FIFO 1 Queue (reserved in CTM)
0x04814	0x04FFC	reserved	reserved
0x05000	0x05010	TEFQ	TX Event FIFO Queue
0x05014	0x05FFC	reserved	reserved
0x06000	0x0603C	CTB	Cut Through Buffer
0x06040	0x3FFFC	reserved	reserved
0x40000	0x7FFFC	LMEM TA	LMEM Transparent Access

Table 8: VBM Virtual Address Map (CTM)

The VBM processes transactions to virtual buffers only when the **MH_CTRL.ENABLE** bit is set. However, transparent LMEM access remains possible even without setting the **MH_CTRL.ENABLE** bit.

In CTM, the RXFQ0 and RXFQ1 are not used, instead, the CTB is used for message reception, and for message transmission in case that the payload of the transmit message is larger than 64 bytes. However, with the CTDMA module, the access to CTB via VBM on the XS_CAN is done/managed by the CTDMA module. This is done by CTDMA module itself, the application SW needs to do nothing.

5.1 Virtual buffer access order

The host issues address in linear incrementing order (4-byte alignment), starting from the start address until the trigger address.

Any other order of addresses issued is considered as a nonlinear access and the Error- and Exception Handling of XS_CAN behaviour starts.

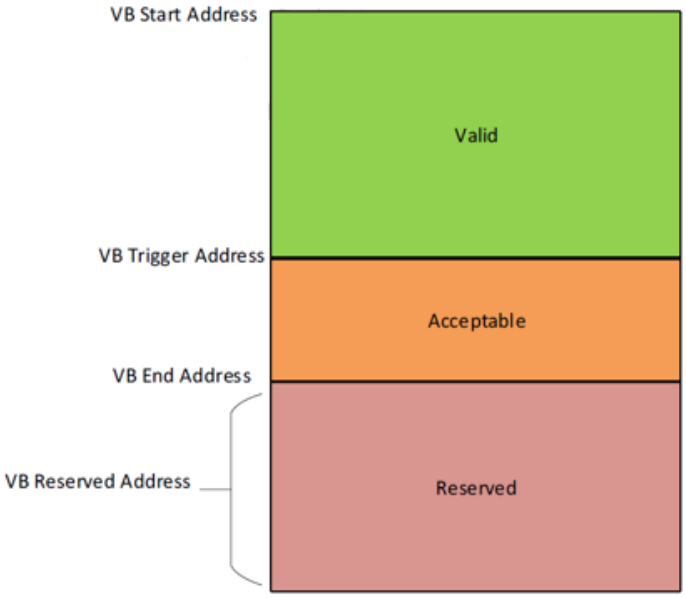

Figure 8: Adress space of a Virtual Buffer (VB)

To start the enqueue and dequeue of a message, the host issues the start address of the VB. Trigger address corresponds to the last word of the message. Trigger address and end address can be same if the message has the maximum payload possible which is a CAN XL with 2KB payload. See more information in [1].

6 Message Enqueue for transmission

The XS_CAN supports a single TXFQ with up to 255 messages, and up to 32 TXPQ slots.

Before using the TXFQ and TXPQ, they must be enabled by setting **MH_CFG.TXFQE= '1'** and **MH_CFG.TXPQE= '1'**.

And to use XS_CAN in CTM, it must be enabled by setting the **MH_CFG.CTME= '1'**.

To transmit a message, the following steps are required:

- **Step 1:** Ensure sufficient memory space for Message Enqueueing

When a Host is used for enqueuing, check the following FIFO Status signals before the enqueuing is started.

- **TXFQ:** check **TXFQ_STS.FULL** to ensure the TXFQ is not full.
- **TXPQ:** check **TXPQ_STS1.TXPQ_FULL** to ensure not all TXPQ slots are occupied.

The enqueue operation will only proceed if the **TXFQ is not full** or the **TXPQ slot is available**.

When an external DMA controller is used for enqueuing instead of the host interface, the following DMA request signals shall be used for checking status changes. Monitor the following Transfer Request Signals to ensure that message write is accepted by VBM.

- **TXFQ: TXFQ_WREQ** (TX FIFO Queue write request from VBM) signal.
- **TXPQ: TXPQ_WREQ** (TX Priority Queue write request from VBM) signal.

See more information in [1], chapter Transfer Request Signals.

- **Step 2:** Start Enqueuing of TXFQ/TXPQ

To start Enqueuing a new message into either the **TXFQ** or **TXPQ**, write the message header **T0** (the first word) to **VB start address**. This address is **0x01000** for TXFQ or **0x02000** for TXPQ. This action triggers the VBM to initiate the enqueue process for the respective Queue.

The TXFQ and TXPQ handling module within the VBM converts the virtual buffer address to the actual LMEM address where the message will be enqueued.

After that, write subsequent words, beginning with T1, linearly at increments of +4 from the start address of TXFQ/TXPQ.

The VBM calculates a trigger address based on the frame type (determined by the FDF and XLF bits in T0), and the RTR and DLC bits in T1.

In CTM,

- The trigger address corresponds to the last word of the enqueued message when the payload of the enqueued message shorter than or equal to 64 bytes.
  - Classic CAN
  - CAN FD
  - CAN XL, **Payload ≤ 64 bytes**

- The trigger address corresponds to the 16th word of the enqueued message's payload when the payload of the enqueued message longer than 64 bytes. (address **0x0104C** for TXFQ or **0x0204C** for TXPQ)
  - CAN XL, **Payload > 64 bytes**

After the trigger address is accessed, the next expected address is the start address of the VB again. In case if this is violated, status flags are updated. Refer the registers **MH_STS0, MH_STS1 or INTERRUPTS**, see more information in [1].
The following diagram shows the enqueue process with the block copy for payload ≤ 64 byte and > 64byte.

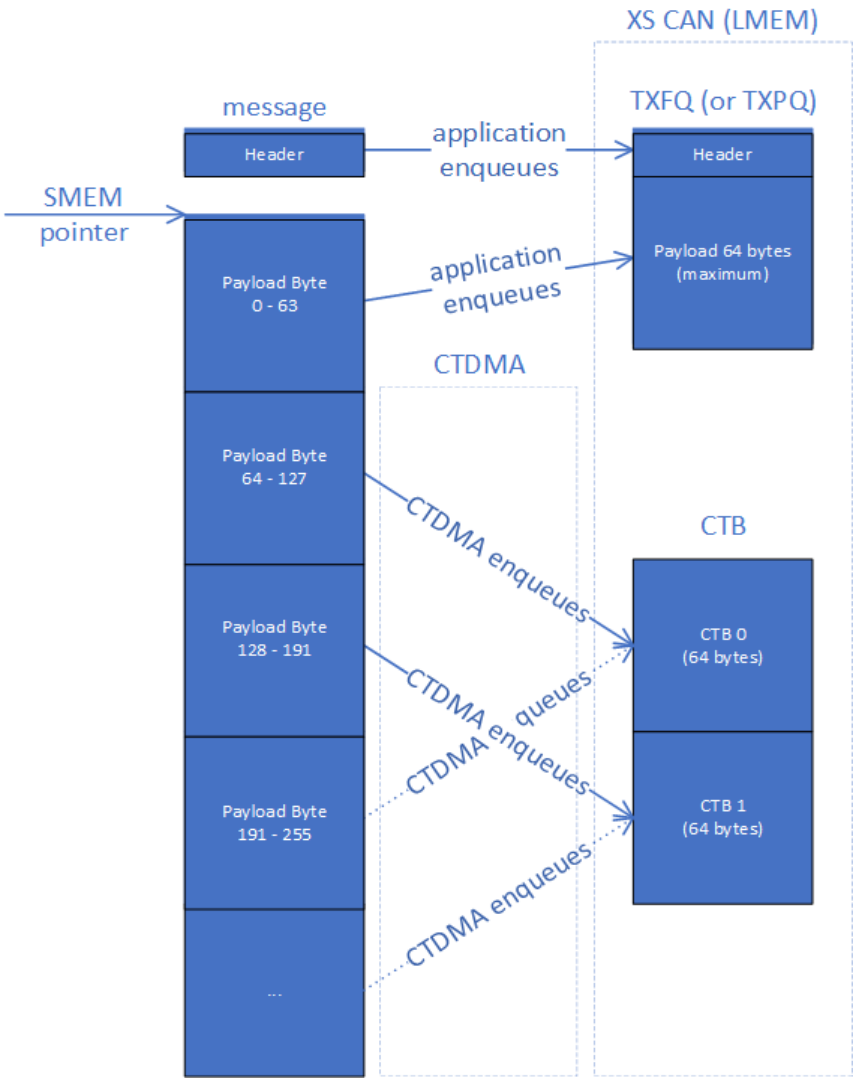

Figure 9: Enqueue and Block Copy overview

- **Step 3:** End of Enqueuing

After enqueuing the following registers are updated by VBM:

For TXFQ:
1) Write pointer register **TXFQ_WPTR.WPTR_ADD** with the actual LMEM address corresponding to the last word of TXQE that is enqueued.
2) The fill level of the TXFQ is incremented by one and is updated into the register **TXFQ_STS.FILL_LEVEL**.

For TXPQ:
1) Write pointer register **TXPQ_WPTR.WPTR_ADD** with the actual LMEM address corresponding to the last word of TXQE that is enqueued.
2) The slot occupied status into the register **TXPQ_STS0.SLOT_OCC[n]**.
3) The fill level of the TXPQ is incremented by one and is update into the register **TXPQ_STS1.FILL_LEVEL**

The Application SW can use these registers to monitor the Enqueue process.

See the following table with the message DLC and payload length.

DLC	data field in bytes			
	classical CAN		CAN FD	CAN XL
	remote frame	data frame		
0	0	0	0	1
1	0	1	1	2
2	0	2	2	3
3	0	3	3	4
4	0	4	4	5
5	0	5	5	6
6	0	6	6	7
7	0	7	7	8
8	0	8	8	9
9	0	8	12	10
10	0	8	16	11
11	0	8	20	12
12	0	8	24	13
13	0	8	32	14
14	0	8	48	15
15	0	8	64	16
16	n/a	n/a	n/a	17
...	n/a	n/a	n/a	...
62	n/a	n/a	n/a	63
63	n/a	n/a	n/a	64
64	n/a	n/a	n/a	65
...	n/a	n/a	n/a	...
n	n/a	n/a	n/a	n+1
...	n/a	n/a	n/a	...
2047	n/a	n/a	n/a	2048

⬅maximum trigger address

Table 9 : Message DLC and payload length

In some applications, message transfers with dynamic lengths, controlled by the message DLC are not always applicable. In such cases, the DMA controller may transfer (Enqueue) with the size of the largest possible TX message (maximum 2060 bytes), regardless of the actual message length. The VBM tolerates this by simply discarding these accesses. The status flags **MH_STS0.TXFQ_VB_NO_SA** or **MH_STS0.TXPQ_VB_NO_SA** will be updated, when this occurs.

6.1 General flowcharts for TXFQ and TXPQ

These two flow diagrams show the general flow of Message Enqueueing for TXFQ and TXPQ.

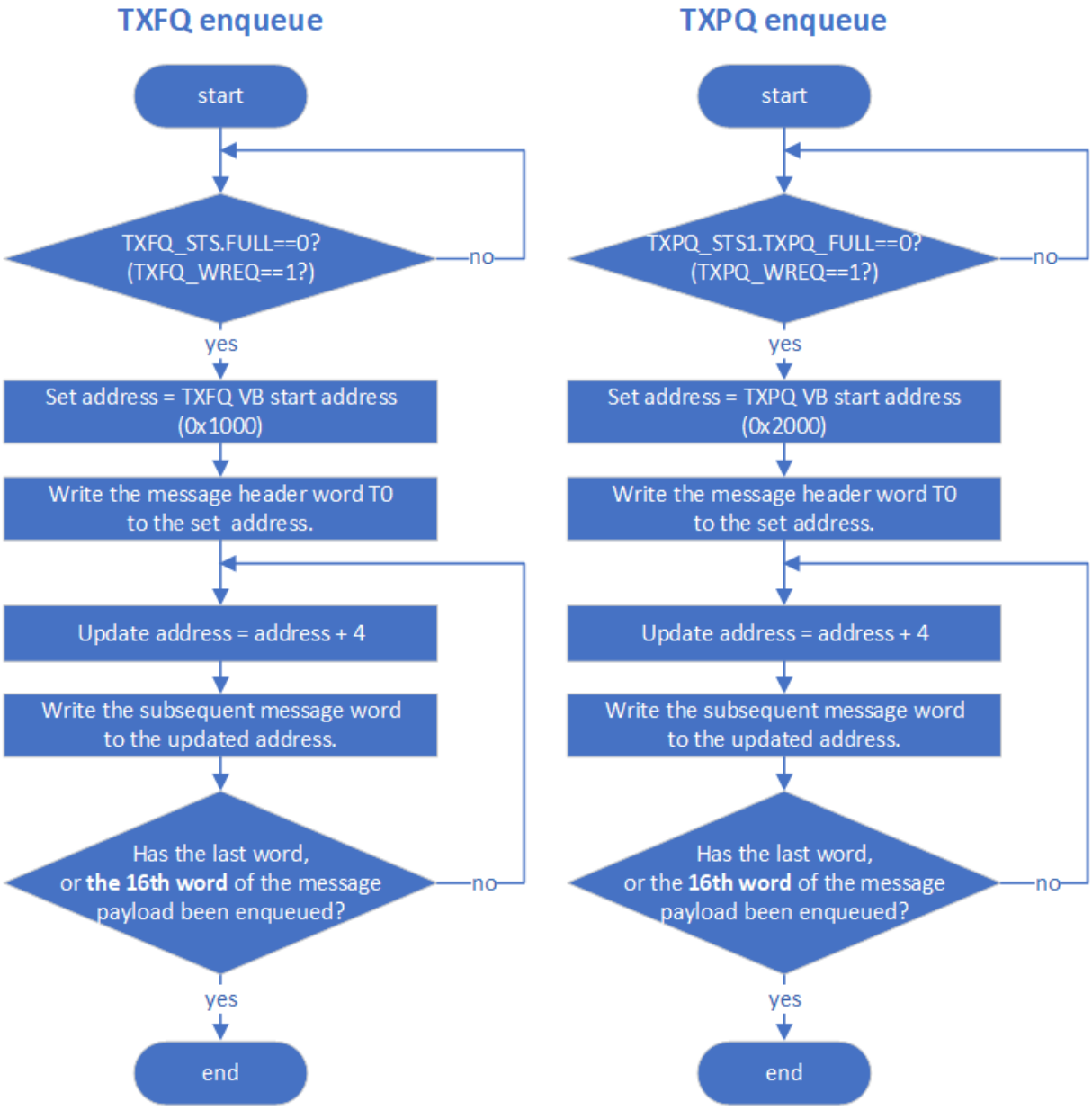

Figure 10 : Enqueue flow diagram for TXFQ and TXPQ.

6.2 Example of Messages Enqueue for TXFQ and TXPQ

The following examples show the Enqueue of TX Messages in detailed steps for TXFQ and TXPQ.

TXFQ example configuration:

Message 1:	A CAN FD frame with DLC=4, trigger address will be at word T2(0x1008)
Message 2:	A CAN XL frame with DLC=31, trigger address will be at word T11(0x102C)
Message 3:	A CAN XL frame with DLC=2047, trigger address will be at word T19(0x104C)

TXPQ example configuration:

Message 4:	A CAN FD frame with DLC=4, trigger address will be at word T2(0x2008)
Message 5:	A CAN XL frame with DLC=31, trigger address will be at word T11(0x202C)
Message 6:	A CAN XL frame with DLC=2047, trigger address will be at word T19(0x204C)

The following steps detail how to enqueue **msg1, msg2** and **msg3** into TXFQ, and **msg4**, **msg5** and **msg6** into TXPQ.

Step	Description	Enqueue to TXFQ	Enqueue to TXPQ
1	Ensure sufficient memory space for enqueue	**TXFQ_STS.FULL**==0? (TXFQ_WREQ==1?)	**TXPQ_STS1.TXPQ_FULL** ==0? (TXPQ_WREQ==1?)
2	If space is available, proceed with enqueuing; otherwise, wait until space becomes available.		
3	Begin enqueuing **msg1/msg4** at the VB start address.	Write T0 (header word 0) to address **0x1000**	Write T0 (header word 0) to address **0x2000**
4	Continue enqueuing **msg1/msg4** the subsequent word	Write T1 (header word 1) to address **0x1004**	Write T1 (header word 1) to address **0x2004**
5	Continue enqueuing **msg1/msg4** the last word	Write T2 (payload byte 0 – 3) to address **0x1008** = trigger address	Write T2 (payload byte 0 – 3) to address **0x2008** = trigger address

Step	Description	Enqueue to TXFQ	Enqueue to TXPQ
6	Ensure sufficient space for enqueue	**TXFQ_STS.FULL**==0? (TXFQ_WREQ==1?)	**TXPQ_STS1.TXPQ_FULL** ==0? (TXPQ_WREQ==1?)
7	If memory space is available, proceed with enqueuing; otherwise, wait until space becomes available.		
8	Begin enqueuing **msg2/msg5** at the VB start address.	Write T0 (header word 0) to address **0x1000**	Write T0 (header word 0) to address **0x2000**
9	Continue enqueuing **msg2/msg5** the subsequent word	Write T1 (header word 1) to address **0x1004**	Write T1 (header word 1) to address **0x2004**
10	Continue enqueuing **msg2/msg5** the subsequent word	Write T2 (acceptance field) to address **0x1008**	Write T2 (acceptance field) to address **0x2008**
11	Continue enqueuing **msg2/msg5** the subsequent word	Write T3 (SMEM pointer) to address **0x100C** Note: since payload ≤ 64, the value of SMEM pointer is not really use.	Write T3 (SMEM pointer) to address **0x200C** Note: since payload ≤ 64, the value of SMEM pointer is not really use.
12	Continue enqueuing **msg2/msg5** the subsequent word	Write T4 (payload byte 0 – 3) to address **0x1010**	Write T4 (payload byte 0 – 3) to address **0x2010**
13	Continue enqueuing **msg2/msg5** the subsequent word	Write T5 (payload byte 4 – 7) to address **0x1014** = trigger address	Write T5 (payload byte 4 – 7) to address **0x2014** = trigger address
…	…	…	…
19	Continue enqueuing **msg2/msg5** the last word	Write T11 (payload byte 28 – 31) to address **0x102C** = trigger address	Write T11 (payload byte 28 – 31) to address **0x202C** = trigger address
20	Ensure sufficient space for enqueue	**TXFQ_STS.FULL**==0? (TXFQ_WREQ==1?)	**TXPQ_STS1.TXPQ_FULL** ==0? (TXPQ_WREQ==1?)

Step	Description	Enqueue to TXFQ	Enqueue to TXPQ
21	If memory space is available, proceed with enqueuing; otherwise, wait until space becomes available.		
22	Begin enqueuing **msg3/msg6** at the VB start address.	Write T0 (header word 0) to address **0x1000**	Write T0 (header word 0) to address **0x2000**
23	Continue enqueuing **msg3/msg6** the subsequent word	Write T1 (header word 1) to address **0x1004**	Write T1 (header word 1) to address **0x2004**
24	Continue enqueuing **msg3/msg6** the subsequent word	Write T2 (acceptance field) to address **0x1008**	Write T2 (acceptance field) to address **0x2008**
25	Continue enqueuing **msg3/msg6** the subsequent word	Write T3 (SMEM pointer) to address **0x100C**	Write T3 (SMEM pointer) to address **0x200C**
26	Continue enqueuing **msg3/msg6** the subsequent word	Write T4 (payload byte 0 – 3) to address **0x1010**	Write T4 (payload byte 0 – 3) to address **0x2010**
27	Continue enqueuing **msg3/msg6** the subsequent word	Write T5 (payload byte 4 – 7) to address **0x1014**	Write T5 (payload byte 4 – 7) to address **0x2014**
…	…	…	…
41	Continue enqueuing **msg3/msg6** the payload word 16	Write T19 (payload byte 60 – 63) to address **0x104C** = trigger address Note: only enqueue until payload byte 63 (64 bytes) when payload > 64 bytes	Write T19 (payload byte 60 – 63) to address **0x204C** = trigger address Note: only enqueue until payload byte 63 (64 bytes) when payload > 64 bytes

Table 10: Detailed steps of Message Enqueue for TXFQ and TXPQ

See following CAN XL TX Queue element (TXFQ) Header (CTM) definition.

Tn	Bits	Name	Description/Constraints
T0	[31]	FDF	FD Format
	[30]	XLF	XL Format
	[29]	XTD	Extended Identifier
	[28:18]	Priority ID [28:18]	Priority ID (11 bit identifier for frames in CAN XL Frame Format (XLFF))
	[17]	RRS	Remote Request Substitution
	[16]	SEC	Simple Extended Content
	[15:8]	VCID [7:0]	Virtual CAN Network ID
	[7:0]	SDT [7:0]	SDU Type
T1	[31]	Reserved	Not Applicable
	[30]	FIR	Fault Injection Request
	[29]	EFC (HOST)	Event FIFO Control: When set to 0, TX Event FIFO is not stored for the TX message. When set to 1, TX Event FIFO is stored for the TX message
	[28:27]	Reserved	Not Applicable
	[26:16]	DLC-XL [10:0]	Data Length Code with CAN XL encoding
	[15:0]	MM (HOST)	Message Marker (HOST)
T2	[31:0]	AF [31:0]	Acceptance Field
T3	[31:0]	TX_PAYLOAD_SMEM_PTR	TX payload SMEM pointer. This SMEM address points to the first payload word of the message.

Note: CAN XL frames (XLFF) require **T0.FDF** = 1, **T0.XLF** = 1 and **T0.XTD** = 0. The header consists of T0, T1,T2 and T3.

Table 11: CAN XL TX Queue Element (TXFQ) Header (CTM)

There are also Header Definition for CC and CAN FD. See more information in [1], chap. 1.5.6.1 TX Message Header Definition.

6.3 Internal Prioritization, and transmission after the enqueue

Inside the XS_CAN, the internal **TX Scan** module identifies which enqueued message in LMEM to be transmitted first. The internal arbitration process selects the message with the **lowest ID**, which has the **highest priority**. Only the first message header element T0 from enqueued message is used to identify the priority during the TX Scan. See more information in [1] chapter TX Scan.

- **TXFQ (only)**: The oldest message (first enqueued) in the TXFQ is transmitted first. (A)
- **TXPQ (only)**: Among the messages currently in the TXPQ slots, the message with the lowest ID is transmitted first. (B)

  Note:

  If **TXPQ_CFG.TX_MSG_SEQE** (Transmit Messages Sequence Enable) is set to '1', messages in the TXPQ that have the same priority are transmitted in a user-defined sequence. This sequence is defined by the user via the **MM** (Message Marker) in the TX Message Header T1. The message with the lowest **MM** is transmitted first.

- **TXFQ and TXPQ (combined):** When both Queues (TXFQ/TXPQ) contain messages, then the message with the lower ID between the **oldest TXFQ message (A)** and **the lowest ID TXPQ message (B)** is transmitted first.

If messages have the same priority, they are transmitted in the following order, first TXPQ slot elements from 0 to 31 followed by TXFQ oldest element.

Hint: The Remote frame and the Data frame of the Classical CAN frame have the same priority.

7 Interrupt Flags

7.1 TX Functional Raw Event Status register (TX_FUNC_RAW)

See the following relevant flags to message transmission.

Bit field/flag	Description
PRT_TX_EVT	A valid CAN message was transmitted by PRT.
MH_CTB_TX_BC_REQ	A block copy transfer pending in TX direction
MH_TEFQ_DEQ	A message is dequeued from TEFQ.
MH_TEFQ_ENQ	A message is enqueued into TEFQ.
MH_TXPQ_ENQ	A message is enqueued into TXPQ.
MH_TXFQ_ENQ	A message is enqueued into TXFQ.

7.2 Error Raw Event Status register (ERR_STS_RAW)

See the following relevant flags to message transmission.

Bit field/flag	Description
PRT_IFF	An invalid Frame Format at TX_MSG detected by PRT.
MH_TEFQ_DROP	A new TEQE is dropped
MH_TXPQ_DEQ_NO_TX	A message has been dequeued from TXPQ but is not transmitted.
MH_TXFQ_DEQ_NO_TX	A message has been dequeued from TXFQ but is not transmitted.

7.3 Safety Raw Event Status register (SAFETY_RAW)

See the following relevant flags to message transmission.

Bit field/flag	Description
PRT_TX_DU	PRT detected underrun condition at TX_MSG sequence.
PRT_USOS	PRT detected unexpected Start of Sequence during TX_MSG sequence.
PRT_TX_ABORTED	PRT detected stop of TX_MSG sequence by TX_MSG_WUSER code ABORT.
MH_TX_ABORT	MH has to abort an ongoing transmission of a message to PRT.
MH_CTB_BC_ERR	Ongoing block copy is cancelled due to error response received at CTM_RESP input
MH_CTB_BC_TO	Time out for block copy request .i.e. block copy not finished in time before data underrun or overflow at CTB.

See more information in [1], chapter 1.8 IRC - Interrupt Controller

These flags are typically processed by the host SW driver interrupt routine.

8 Software Examples

The following table provides examples of the 'C' SW-functions that are used as demonstrated in this application note.

Name:	`xs_can_app_note_tx_rx_ctdma()`
File:	`../xs_can/app_notes/app_note_tx_rx.c`
Description:	The example demonstrates how to use the TXFQ and the RXFQ to transmit and receive Classical CAN, CAN FD and CAN XL messages in CTM with CTDMA.
Name:	`xs_can_app_note_txpq_ctdma()`
File:	`../xs_can/app_notes/app_note_txpq.c`
Description:	The example demonstrates how to use the TXPQ and the RXFQ to transmit and receive Classical CAN, CAN FD and CAN XL messages in CTM with CTDMA.
Name:	`xs_can_mh_set_config()`
File:	`../xs_can/mh/xs_can_mh.c`

Description:	This function demonstrates how to configure all of the relevant MH configuration registers.
	`MH_CFG`
	`RX_FILTER_CFG`
	`RX_FILTER_LMEM`
	`TXFQ_LMEM_SA`
	`TXFQ_LMEM_EA`
	`TXFQ_CFG`
	`TXPQ_CFG`
	`TXPQ_LMEM`
	`TEFQ_LMEM`
	`TEFQ_CFG`
	`CTB_LMEM`
	`RXFQ0_SA`
	`RXFQ0_EA`
	`RXFQ1_SA`
	`RXFQ1_EA`

Name:	`xs_can_mh_tx_fifo_queue_enqueue_msg()`
File:	`../xs_can/mh/xs_can_mh.c`
Description:	This function demonstrates how to enqueue a message into the TXFQ.
Name:	`xs_can_mh_tx_priority_queue_enqueue_msg()`
File:	`../xs_can/mh/xs_can_mh.c`
Description:	This function demonstrates how to enqueue a message into the TXPQ.

Table 12: List of SW example functions for TX message handling

This application note contains all the C-source files that are necessary to compile the examples. The file `_info.txt` contains a short description of each provided source file.