BOSCH
Invented for life

XS_CAN

Slim CAN XL IP module

Reception Handling Cut Through Mode (CTM)

Application Note XS_CAN_AN003

Document Revision 1.0
05.08.2026

Robert Bosch GmbH
Mobility Electronics

LEGAL NOTICE

Revision History

Version	Date	Remark
1.0	16.06.2026	Initial version for XS_CAN 1.1.0

Conventions

The following conventions are used within this document:

Register names	`RXFQ0_SA, RXFQ_STS`
Names of files and directories	`directoryname/filename`
Source code/function names	`xs_can_mh_rx_fifo_queue_dequeue_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 Module 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 001 RX 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
FEC	Filter Element Configuration
FMM	Full Message Mode
HOST	This is the CPU which is hosting the X_CAN
IP	Intellectual property
IRC	Interrupt Controller
LMEM	Local Memory

Term	Meaning
MH	Message Handler
PRT	Protocol Controller
PWM	Pulse Width Modulation
RAM	Random Access Memory
REFM	Reference Pair Mask
REFP	Reference Value-Mask Pair
REFV	Reference Pair Value
RX	Receive
RXFQ	Receive FIFO Queue
SMEM	System Memory
TEFQ	Transmit Event FIFO Queue
TS	Timestamp
TX	Transmit
TXEF	Transmit Event FIFO
TXPQ	Transmit Priority Queue
TXFQ	Transmit FIFO Queues
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:

- RX Message dequeue for reception (RXFQ0, RXFQ1)

Note: The configuration and use of RX Filtering have 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.

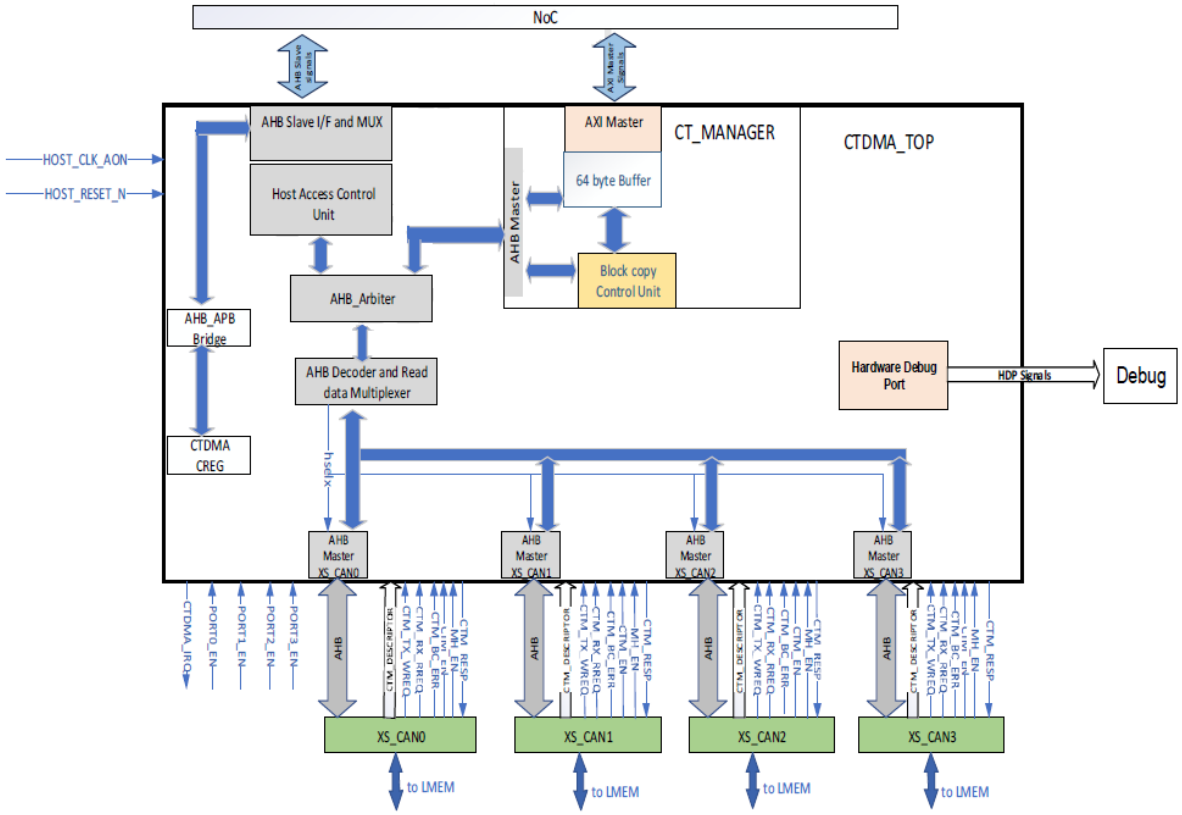

Figure 1: CTMDA Block diagram

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

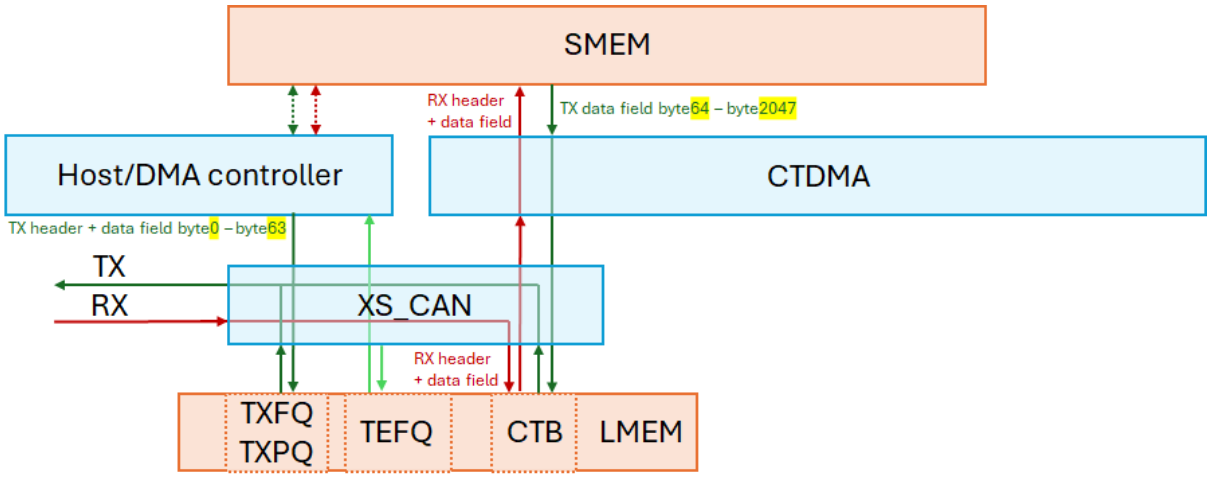

Figure 2: TX and RX relevant data flow path in CTM

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.

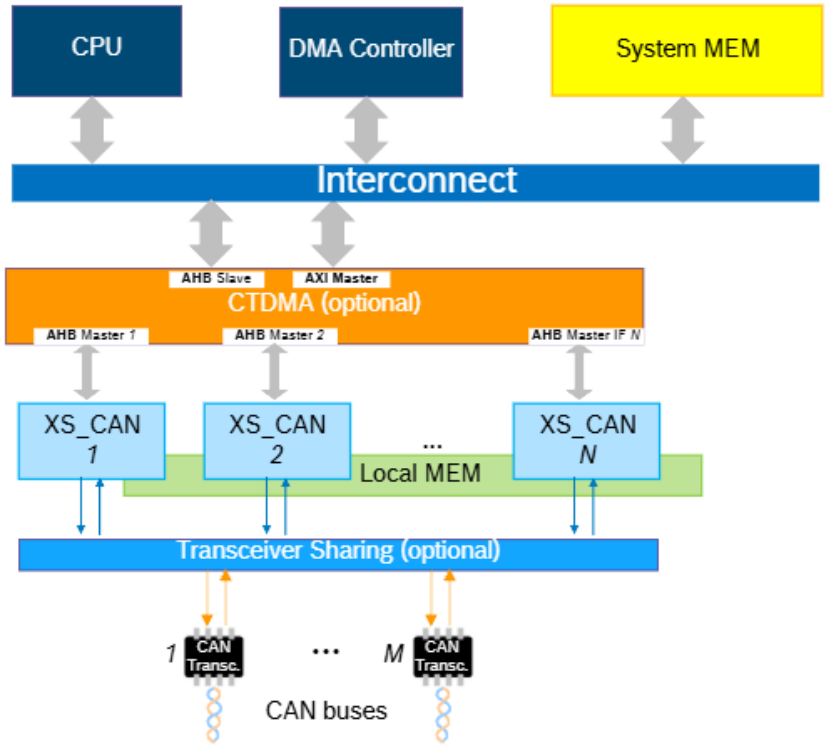

Figure 3: XS_CAN with CTDMA

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.

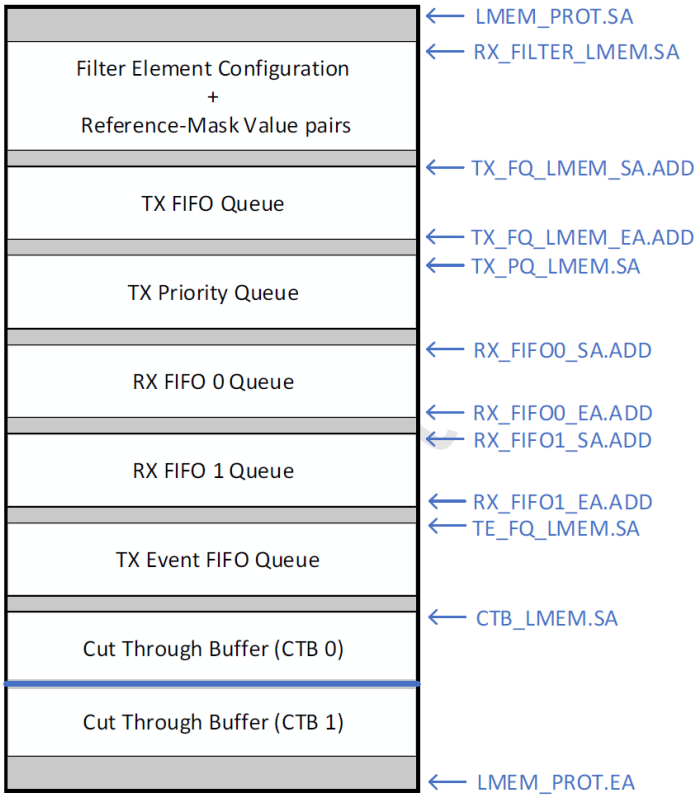

Figure 4: LMEM configuration example

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.

The XS_CAN supports **two RXFQs** with up to 255 messages per RXFQ.
However, in CTM, **no LMEM** is required for RXFQs. Instead, the received messages are first stored in the CTB and then copied into the SMEM by the CTDMA. The application only needs to configure the start address on the LMEM for the CTB.

Received Messages are stored in the **RXFQ0** and/or the **RXFQ1** depending on the use case of the application.

RX Messages are stored in **SMEM** and consist of a Message Header, a Message Payload and Message Timestamp. [1] describes the RX Message Header's structure and how it is stored in the RXFQ. The Message Header's data structure depends on the CAN Frame Format (CAN CC, CAN FD, CAN XL).

4.1 Queue memory space in LMEM configuration

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

4.1.1 RXFQ memory size

The size and location of RXFQ in SMEM can be configured via the following registers.

RXFQ0_SA.ADD	defines the RXFQ0's start addresses within SMEM.
RXFQ0_EA.ADD	defines the RXFQ0's end addresses within SMEM.
RXFQ1_SA.ADD	defines the RXFQ1's start addresses within SMEM.
RXFQ1_EA.ADD	defines the RXFQ1's end addresses within SMEM.

Their values must always be 64-byte aligned address; therefore, only the higher 26 bits (31:6) of the 32-bit address are considered.

Example: For getting 64byte sized RXFQ0/1, if the **RXFQ0/1_SA.ADD**=0x0100, the **RXFQ0/1_EA.ADD** must be 0x0140 and not 0x013C. The next queue, if configured, shall start from the next 64-byte aligned boundary which is 0x0180. The space from 0x0144 to 0x017C will remain unused in this example.

Hint:
The maximum size of RX message varies according to its CAN protocol type.

	RX Message						total size of one message	
	Header		max Payload		Timestamp			
	(byte)	(word)	(byte)	(word)	(byte)	(word)	(byte)	(word)
CC	8 (R0,R1)	2 (R0,R1)	8	2	8	2	24	6
FD	8 (R0,R1)	2 (R0,R1)	64	16	8	2	80	20
XL	12 (R0,R1,R2)	3 (R0,R1,R2)	2048	512	8	2	2068	517

Example configuration for RXFQ

In this example configuration the RXFQ0 Queue starts at address 0x10002000 (byte address) in the SMEM.

Number of messages in RXFQ	Maximum message length	Calculation and configuration **See following Note:** (CC, FD = 8Byte Header, XL= 12Byte Header) (CC, FD, XL = 8Byte Timestamp) (Required Size = Nr Message * (Header + payload + Timestamp) (RXFQ_LMEM_SA.ADD= SA / 64) (RXFQ_LMEM_EA.ADD= EA / 64)
10	Classical CAN 8-byte payload	required size = 240 bytes SA = 0x10002000 (byte address) EA = 0x100020EF (byte address) configurable size = 256 bytes (multiple of 64 bytes) SA = 0x10002000 (64-byte aligned address) EA = 0x10002100 (64-byte aligned address) **RXFQ0_SA.ADD = 0x400080** **RXFQ0_EA.ADD = 0x400084**
10	CAN FD 8-byte payload	required size = 240 bytes SA = 0x10002000 (byte address) EA = 0x100020EF (byte address) configurable size = 256 bytes (multiple of 64 bytes) SA = 0x10002000 (64-byte aligned address) EA = 0x10002100 (64-byte aligned address) **RXFQ0_SA.ADD = 0x400080** **RXFQ0_EA.ADD = 0x400084**
10	CAN FD 64-byte payload	required size = 800 bytes SA = 0x10002000 (byte address) EA = 0x1000231F (byte address) configurable size = 832 bytes (multiple of 64 bytes) SA = 0x10002000 (64-byte aligned address) EA = 0x10002340 (64-byte aligned address) **RXFQ0_SA.ADD = 0x400080** **RXFQ0_EA.ADD = 0x40008D**
10	CAN XL 512-byte payload	required size = 5320 bytes SA = 0x10002000 (byte address) EA = 0x100034C7 (byte address) configurable size = 5376 bytes (multiple of 64 bytes) SA = 0x10002000 (64-byte aligned address) EA = 0x10003500 (64-byte aligned address) **RXFQ0_SA.ADD = 0x400080** **RXFQ0_EA.ADD = 0x4000D4**
10	CAN XL 2048-byte payload	required size = 20680 bytes SA = 0x10002000 (byte address) EA = 0x100070C7 (byte address) configurable size = 20736 bytes (multiple of 64 bytes) SA = 0x10002000 (64-byte aligned address) EA = 0x10007100 (64-byte aligned address) **RXFQ0_SA.ADD = 0x400080** **RXFQ0_EA.ADD = 0x4001C4**

4.1.2 CTB (Cut Through Buffer)

The XS_CAN IP supports **2 Cut Through Buffers (CTB0 and CTB1)**.

In CTM, all the received messages are stored into the two cut through buffers. RXFQs are located in system memory (SMEM) and RXFQs in the LMEM are disabled in CTM. The messages buffered in CTBs are transferred to SMEM RXFQ in blocks of 64bytes. The target RXFQ in SMEM must be known before one CTB block gets full. i.e before one 64byte block is ready for copy.

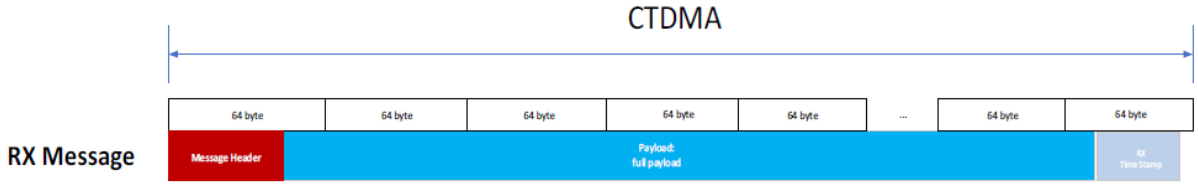

Figure 5: RX Message in CTM

MH starts to receive words from PRT at RX_MSG interface and are stored into the first available CTB. One full CTB must be free to start storing. If no CTB is available when the first word is received at the RX_MSG interface, the message is dropped. When one CTB is filled with data, VBM raises CTM_RX_RREQ for the CTDMA to start copying from CTB to SMEM. Cut Through Descriptors are also updated at the same time by RX Handler with the relevant information.

See more information in [1], chapter 1.5.7.2 “CTD updates in RX direction”.

In CTM, the host does not need to copy the data from SMEM to CTB, this is done by CTDMA. See more information in [1], chapter 1.5.6.7.3 CTM Description.

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). There are **CTB_LMEM.SA**(CTB0) and **CTB_LMEM.SA**+64bytes (CTB1).

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 stores the RX Filter Elements (Filter Element Configuration and Reference Value-Mask Pair) in the LMEM. Virtual buffers are not available for storing RX filter elements. Both read and write operations are allowed in this address range.

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 3: 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.

Note: 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. The Application 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 behaviour starts.

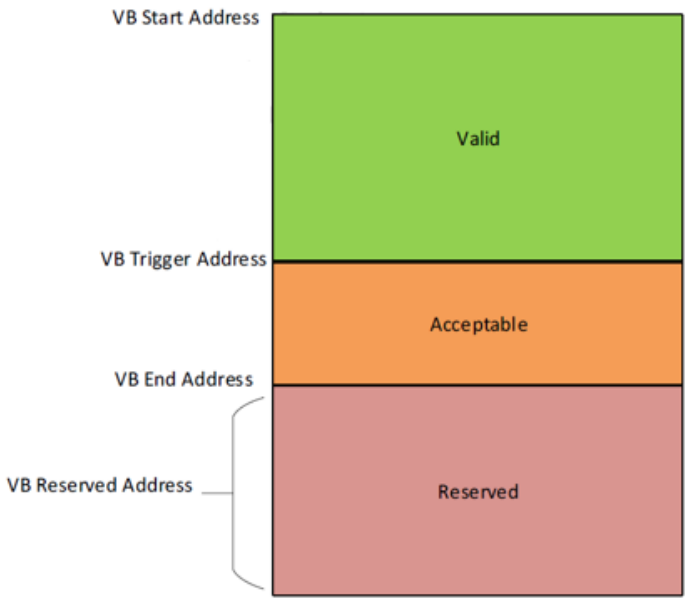

Figure 6: Adress space of a Virtual Buffer (VB)

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 RX Message Dequeue

The XS_CAN supports two RXFQs (RXFQ0 and RXFQ1) with up to 255 messages per RXFQ.RXFQ0/1 contains the RX messages which are received after RX filtering.

Before using the RXFQ0 and RXFQ1, **MH_CFG.RXFQ0E** and RXFQ1 **MH_CFG.RXFQ1E** must be enabled respectively.

To dequeue a message, the following steps are required:

- **Step1:** Ensure there is a message in the RXFQ available.

Important hint: the interpretation of the **RXFQ_STS.RXFQ0_FILL_LEVEL** and **RXFQ_STS.RXFQ1_FILL_LEVEL** in CTM differs from in FMM.

FMM: Indicates the number of messages in RXFQ0 pending to be dequeued. Possible values are 0 to 255.
CTM: Acts as a message count which gets reset at 255 and again starts to count up.

When a Host is used for dequeuing, the Host need to check the following FIFO Status signals before the dequeuing is started. **RXFQ_STS.RXFQ0_FILL_LEVEL** or **RXFQ_STS.RXFQ1_FILL_LEVEL** depending on which is relevant. These 2 flags indicate the total number of messages have been received in RXFQ0 and RXFQ1.

The Host may need to keep tracking how many messages the host has already dequeued the from the RXFQ, and how many messages are still in the RXFQ available for the dequeue.

Example:
Number of messages to be dequeued = **RXFQ_STS.RXFQ_FILL_LEVEL** – number of messages already dequeued.

Please note that after the **RXFQ_STS.RXFQ_FILL_LEVEL** wraps from 255 to 0, the correct calculation must be also taken care.

- **Step 2:** Start dequeuing

To start dequeuing a message from the RXFQ on the SMEM, read the message header **R0 (the first word)** at
1) When dequeuing the first message: Start address of the RXFQ.
2) When dequeuing further message: The next word address after the last dequeued address (the last dequeue address is the address of the most significant word of the timestamp of the last dequeued message)

After that, read subsequent words, beginning with R1, linearly at increments of 4-byte address. Until the last words which is the most significant word of the timestamp of the message.

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	...
n	n/a	n/a	n/a	n+1
...	n/a	n/a	n/a	...
2047	n/a	n/a	n/a	2048

Table 4: Message DLC and payload length

While dequeuing, the host may check if the dequeued address = **RXFQ0_WRAP_PTR.SMEM_ADD/RXFQ0_WRAP_PTR.SMEM_ADD** or not. If yes, the Host has to start dequeuing at the start address of the RXFQ again.

This **SMEM_ADD** indicates the address in SMEM at which RXFQ0/RXFQ1 is wrapped around to store the next full RX message. This register is updated by the IP. The value is word aligned. The **SMEM_ADD[1:0]** bits are always 0.

Tip: This wrap checking may only need to do once per message (e.g. before dequeuing the R0). Because the XS CAN checks if the entire message cannot fit into the bottom of the target RXFQ, the XS CAN will wrap and store the receiving message at the start address of the RXFQ again.

- **Step 3:** Updating RXFQ read pointer address.

After the Host dequeues the last word of the current dequeued message, the Host has to update the **RXFQ0_RPTR.ADD/RXFQ1_RPTR.ADD** with that last word address of the current dequeued message. This is to inform the IP up to which address the RXFQ0/RXFQ1 is dequeued.

This is a word aligned address. The **ADD[1:0]** bits are always 0 and not considered.

General Flowcharts

These two flow diagrams show the general flow of Message dequeuing for RXFQ0/1.

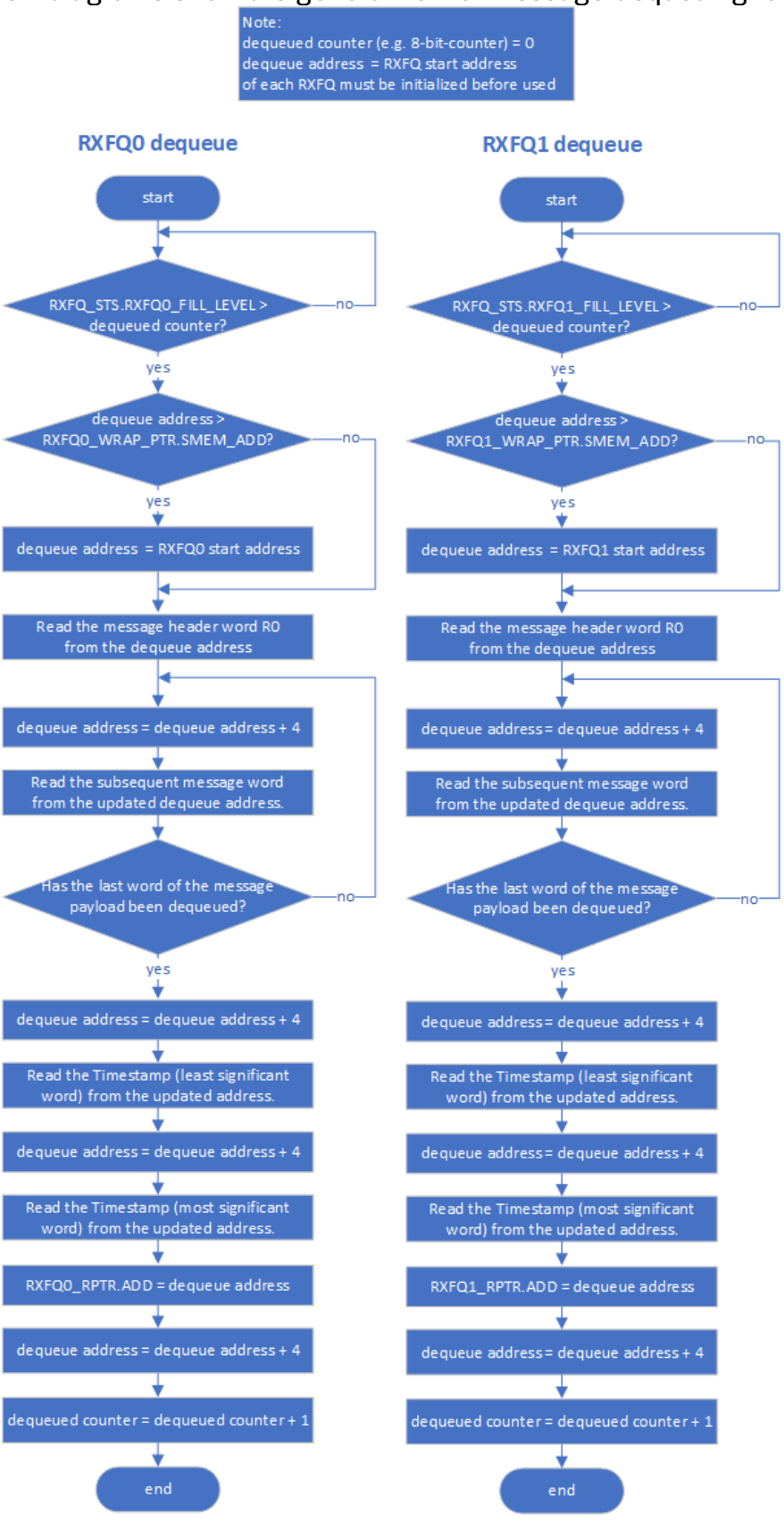

Figure 7 : Dequeue flow diagram for RXFQ0/1

Example: Dequeue messages from RXFQ0 and RXFQ1

RXFQ0:

Message 1:	A CAN FD frame with DLC=4,
Message 2:	A CAN XL frame with DLC=6,

RXPQ1:

Message 3:	A CAN FD frame with DLC=4,
Message 4:	A CAN XL frame with DLC=6,

The following steps detail how to dequeue **msg1** and **msg2** from RXFQ0 and **msg3** and **msg4** from RXFQ1.

Note: Dequeued counter (e.g. 8-bit-counter) = 0, dequeue address = RXFQ, start address of each RXFQ must be initialized before used.

Step	Description	Dequeue from RXFQ0	Dequeue from RXFQ1
1	Ensure there is a message available for dequeue.	**RXFQ_STS. RXFQ0_FILL_LEVEL >** dequeued counter?	**RXFQ_STS. RXFQ1_FILL_LEVEL** > dequeued counter?
2	If message is available, proceed with dequeuing; otherwise, wait until message becomes available.		
3	Check if FIFO wrapped around or not.	dequeue address > **RXFQ0_WRAP_PTR. SMEM_ADD**?	dequeue address > **RXFQ1_WRAP_PTR. SMEM_ADD**?
4	If FIFO wrapped around, (if not, skip this step)	dequeue address = RXFQ0 Start Address	dequeue address = RXFQ1 Start Address
5	Start dequeuing **msg1/msg3**	Read R0 (header word 0) dequeue address	Read R0 (header word 0) from dequeue address
6	increment dequeue address	dequeue address = dequeue address + 4	dequeue address = dequeue address + 4
7	Continue dequeuing **msg1/msg3** the subsequent word	Read R1 (header word 1) from dequeue address	Read R1 (header word 1) from dequeue address
8	increment dequeue address	dequeue address = dequeue address + 4	dequeue address = dequeue address + 4
9	Continue dequeuing **msg1/msg3** the last payload word	Read R2 (payload byte 0 – 3) from dequeue address	Read R2 (payload byte 0 – 3) from dequeue address

Step	Description	Dequeue from RXFQ0	Dequeue from RXFQ1
10	increment dequeue address	dequeue address = dequeue address + 4	dequeue address = dequeue address + 4
11	Continue dequeuing **msg1/msg3** the Timestamp	Read R3 (low significant word) from dequeue address	Read R3 (low significant word) from dequeue address
12	increment dequeue address	dequeue address = dequeue address + 4	dequeue address = dequeue address + 4
13	Continue dequeuing **msg1/msg3** the Timestamp	Read R4 (high significant word) from dequeue address	Read R4 (high significant word) from dequeue address
14	increment dequeue address	dequeue address = dequeue address + 4	dequeue address = dequeue address + 4
15	increment dequeued counter	dequeued counter = dequeued counter + 1	dequeued counter = dequeued counter + 1
16	Ensure there is a message available for dequeue.	**RXFQ_STS. RXFQ0_FILL_LEVEL >** dequeued counter?	**RXFQ_STS. RXFQ1_FILL_LEVEL** > dequeued counter?
17	If message is available, proceed with dequeuing; otherwise, wait until message becomes available.		
18	Check if FIFO wrapped around or not.	dequeue address > **RXFQ0_WRAP_PTR. SMEM_ADD**?	dequeue address > **RXFQ1_WRAP_PTR. SMEM_ADD**?
19	If FIFO wrapped around, (if not, skip this step)	dequeue address = RXFQ0 Start Address	dequeue address = RXFQ1 Start Address
20	Start dequeuing **msg2/msg4**	Read R0 (header word 0) dequeue address	Read R0 (header word 0) from dequeue address
21	increment dequeue address	dequeue address = dequeue address + 4	dequeue address = dequeue address + 4
22	Continue dequeuing **msg2/msg4** the subsequent word	Read R1 (header word 1) from dequeue address	Read R1 (header word 1) from dequeue address
23	increment dequeue address	dequeue address = dequeue address + 4	dequeue address = dequeue address + 4
24	Continue dequeuing **msg2/msg4** the subsequent word	Read R2 (acceptance field) from dequeue address	Read R2 (acceptance field) from dequeue address
25	increment dequeue address	dequeue address = dequeue address + 4	dequeue address = dequeue address + 4

Step	Description	Dequeue from RXFQ0	Dequeue from RXFQ1
26	Continue dequeuing **msg2/msg4** the subsequent word	Read R3 (payload byte 0 – 3) from dequeue address	Read R3 (payload byte 0 – 3) from dequeue address
27	increment dequeue address	dequeue address = dequeue address + 4	dequeue address = dequeue address + 4
28	Continue dequeuing **msg2/msg4** the last payload word	Read R4 (payload byte 4 – 7) from dequeue address note: since DLC = 6, byte 7 is read but not used.	Read R4 (payload byte 4 – 7) from dequeue address note: since DLC = 6, byte 7 is read but not used.
29	increment dequeue address	dequeue address = dequeue address + 4	dequeue address = dequeue address + 4
30	Continue dequeuing **msg2/msg4** the Timestamp	Read R5 (low significant word) from dequeue address	Read R5 (low significant word) from dequeue address
31	increment dequeue address	dequeue address = dequeue address + 4	dequeue address = dequeue address + 4
32	Continue dequeuing **msg2/msg4** the Timestamp	Read R6 (high significant word) from dequeue address	Read R6 (high significant word) from dequeue address
33	increment dequeue address	dequeue address = dequeue address + 4	dequeue address = dequeue address + 4
34	increment dequeued counter	dequeued counter = dequeued counter + 1	dequeued counter = dequeued counter + 1

Table 5: Dequeue messages from RXFQ0 and RXFQ1

7 Interrupt Flags

7.1 RX Functional Raw Event Status register (RX_FUNC_RAW)

See the following relevant flags to message reception and filtering.

Bit field/flag	Description
PRT_RX_EVT	A valid CAN message was received at PRT.
MH_RX_MSG_ALERT	A message alert is detected. (**FEC.ALERT** = ’1’)
MH_RX_FILTER_MATCH	A filter match is detected. (**FEC.IRQ** = ‘1’)
MH_CTB_RX_BC_REQ	MH interrupt that indicates a block copy transfer pending in RX direction
MH_RXFQ1_ENQ	A message is enqueued into RXFQ1.
MH_RXFQ0_ENQ	A message is enqueued into RXFQ0.

7.2 Error Raw Event Status register (ERR_STS_RAW)

See the following relevant flags to message reception.

Bit field/flag	Description
MH_RXFQ1_DROP	A new message to be stored into RXFQ1 is dropped.
MH_RXFQ0_DROP	A new message to be stored into RXFQ0 is dropped.

7.3 Safety Raw Event Status register (SAFETY_RAW)

See the following relevant flags to message reception and filtering.

Bit field/flag	Description
PRT_RX_DO	PRT detected overflow condition at RX_MSG sequence.
MH_RX_ABORT	MH has to abort an ongoing reception of a message from PRT.
MH_RX_FILTER_ERR	An error in filtering process like filtering not completed on time or target RXFQ not enabled.
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 section provides examples of the C 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_rx_fifo_queue_dequeue_msg()`
File:	`../xs_can/mh/xs_can_mh.c`
Description:	This functon demonstrates how to dequeue a message from the RXFQ.

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.