BOSCH
Invented for life

XS_CAN

Slim CAN XL IP module

Reception Handling Full Message Mode (FMM)

Application Note XS_CAN_AN001

Document Revision 1.0
31.03.2026

Robert Bosch GmbH
Mobility Electronics

LEGAL NOTICE

Revision History

Version	Date	Remark
1.0	31.03.2026	Initial version for XS_CAN 1.0.0 – 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	calc_min_host_clk_freq_for_XS_CAN.xlsx
[4]	MS/EEJ	XS_CAN_Local_Memory_size_calculator.xlsx

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
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
MH	Message Handler
PRT	Protocol Controller
PWM	Pulse Width Modulation
RAM	Random Access Memory
REFM	Reference Pair Mask

Term	Meaning
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 Prioriy Queue
TXFQ	Transmit FIFO Queue
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 1.0.0 through 1.1.0**

The topics covered include:

- Local Memory configuration for Queues (RXFQ0, RXFQ1) and RX Filter
- RX Message dequeue for reception (RXFQ0, RXFQ1)
- RX Filtering configuration

Important Note:

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

2 LMEM configuration and interface

The XS_CAN requires external local memory to store all messages (transmitted and received),TX Event information, and RX filter elements. This external memory is called Local Memory (LMEM).

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 1 shows a LMEM configuration example.

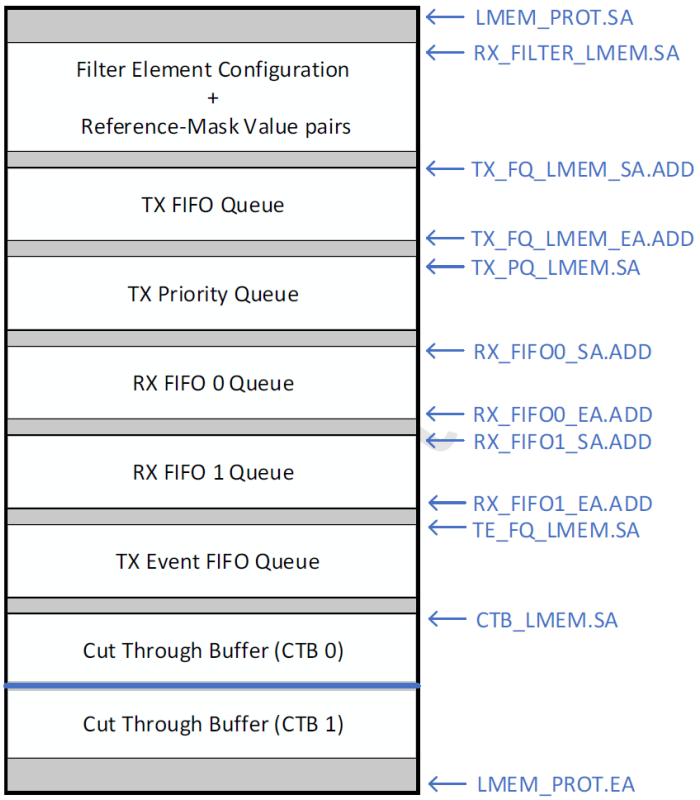

Figure 1: LMEM configuration example

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 end address (LMEMPC_END_ADDR) both of 18 bit width are 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 considered and are treated as '0'.

The **LMEM_PROT.SA** (Start Address of LMEM space) and **LMEM_PROT.EA** (End Address of LMEM space) define the LMEM access protection for an XS_CAN instance.

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

The XS_CAN supports **two RXFQs** with up to 255 messages per RXFQ.

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

RX Messages are stored in LMEM 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).

RX Filter are stored in LMEM consist of Filter Elements and Filter Reference-Mask Pairs. [1] describes how the Filter Elements and the Reference-Mask Pairs are stored in the LMEM.

The XS_CAN supports up to **127 RX Filter** Elements and up to **254 Reference-Mask Pairs.**

2.1 LMEM size

For the LMEM size calculation we provide with the XS_CAN IP a LMEM size calculator [4]. For the LMEM size calculation the LMEM size calculator [4]should be used. This LMEM size calculator provide already different memory example configurations for FMM or CTM mode

2.2 Host Clock frequency

For the Host clock calculation, we provide with the XS_CAN IP an Host clock calculator [3]. This Host clock calculation should be used to calculate Host clock frequency. This [3] is to check if a specific combination of bit rates (nominal bit rate, FD Data bit rate, XL Data bit rate) and SMEM/LMEM latencies is functional under specific system parameters.

2.3 Queue memory space in LMEM configuration

2.3.1 RXFQ memory size

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

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

The maximum size of RX message varies according to its 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 starts at address 0x02000 (byte address) in the LMEM.

Number of messages in RXFQ	Maximum message length	Calculation and configuration
10	Classical CAN 8-byte payload	required size = 240 bytes SA = 0x02000 (byte address) EA = 0x020EF (byte address) configurable size = 256 bytes (multiple of 64 bytes) SA = 0x02000 (64-byte aligned address) EA = 0x020C0 (64-byte aligned address) **RXFQ0_SA.ADD = 0x0080** **RXFQ0_EA.ADD = 0x0083**
10	CAN FD 8-byte payload	required size = 240 bytes SA = 0x02000 (byte address) EA = 0x020EF (byte address) configurable size = 256 bytes (multiple of 64 bytes) SA = 0x02000 (64-byte aligned address) EA = 0x020C0 (64-byte aligned address) **RXFQ0_SA.ADD = 0x0080** **RXFQ0_EA.ADD = 0x0083**
10	CAN FD 64-byte payload	required size = 800 bytes SA = 0x02000 (byte address) EA = 0x0231F (byte address) configurable size = 832 bytes (multiple of 64 bytes) SA = 0x02000 (64-byte aligned address) EA = 0x02300 (64-byte aligned address) **RXFQ0_SA.ADD = 0x0080** **RXFQ0_EA.ADD = 0x008C**
10	CAN XL 512-byte payload	required size = 5320 bytes SA = 0x02000 (byte address) EA = 0x034C7 (byte address) configurable size = 5376 bytes (multiple of 64 bytes) SA = 0x02000 (64-byte aligned address) EA = 0x034C0 (64-byte aligned address) **RXFQ0_SA.ADD = 0x0080** **RXFQ0_EA.ADD = 0x00D3**
10	CAN XL 2048-byte payload	required size = 20680 bytes SA = 0x02000 (byte address) EA = 0x070C7 (byte address) configurable size = 20736 bytes (multiple of 64 bytes) SA = 0x02000 (64-byte aligned address) EA = 0x070C0 (64-byte aligned address) **RXFQ0_SA.ADD = 0x0080** **RXFQ0_EA.ADD = 0x01C3**

2.3.2 RX Filter memory size

RX_FILTER_LMEM.SA	defines the RX Filter Elements start address within LMEM.
RX_FILTER_CFG.FE_NUM	defines the number of RX Filter Elements, up to 127.

The size of the RX FILTER in LMEM
= size of FEC (in 32-bit word) + size of REFP (in 32-bit word)
$= \left(\left\lceil\frac{n+m}{4}\right\rceil\right) + 2(n+2m)$ in the unit of 32-bit word

n = number of Filter Element with 1 comparison
m = number of Filter Element with 2 comparisons

Example configuration

For the following configuration of 10 Filter Elements
n = 5 and m = 5 requires 33 x 32-bit word.

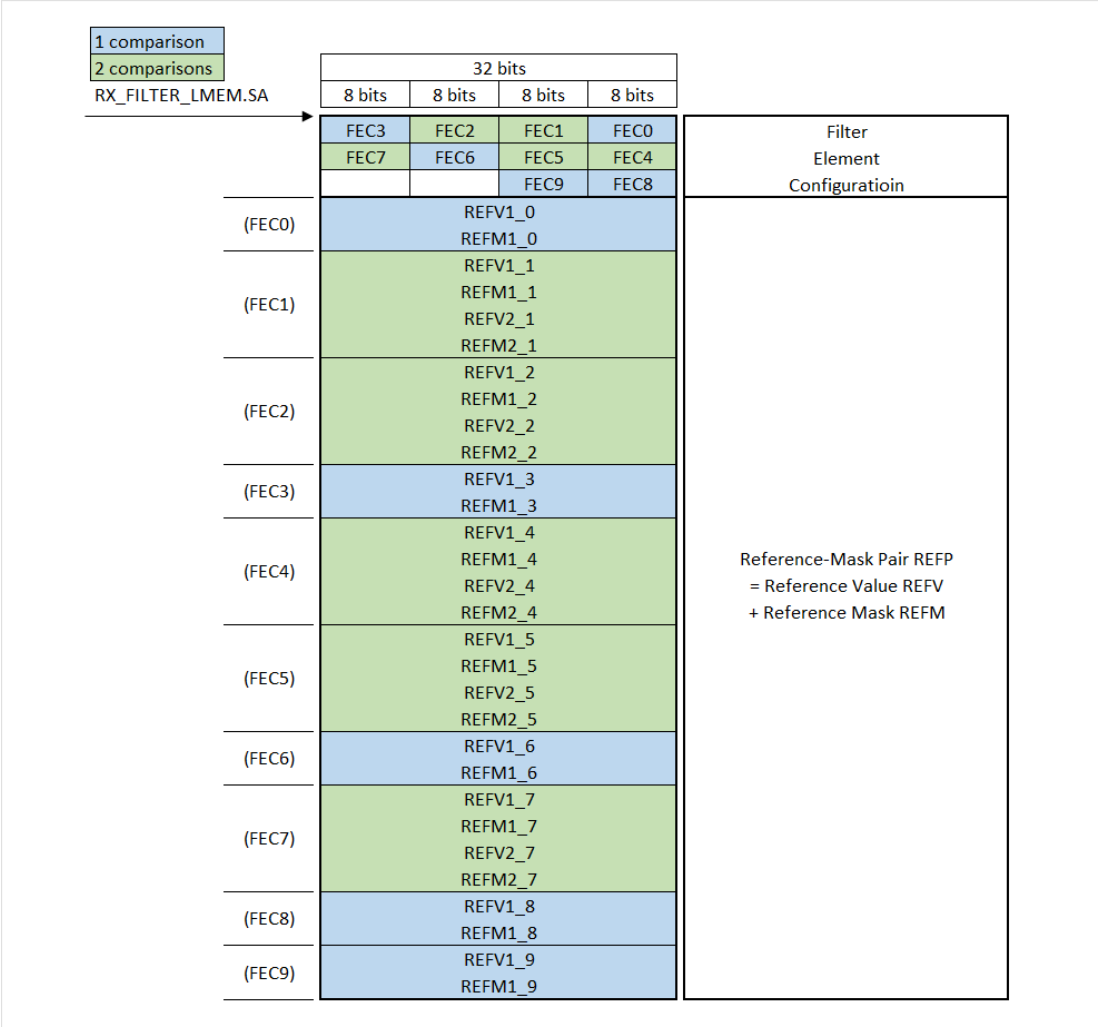

Figure 2: Example Filter configuration diagram

The first Reference Pair begins at the next 32-bit word after the last FEC. Each FEC has its own Reference Pair.

3 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, and TXPQ slots, as well as the dequeuing of messages from RXFQs and the TEFQ.

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, RXFQ0, RXFQ1, and 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-FMM's virtual buffer address map is shown in the following table.

Start location Address	End locationAddress	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
0x03814	0x03FFC	reserved	reserved
0x04000	0x04810	RXFQ1	RX FIFO 1 Queue
0x04814	0x04FFC	reserved	reserved
0x05000	0x05010	TEFQ	TX Event FIFO Queue
0x05014	0x3FFFC	reserved	reserved
0x40000	0x7FFFC	LMEM TA	LMEM Transparent Access

Table 1: VBM Virtual Address Map (FMM)

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.

4 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.**

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

The dequeue operation shall only proceed if the **RXFQ fill level** is greater than zero.

Alternatively, the interrupt **RX_FUNC_RAW.MH_RXFQ0_ENQ** or **RX_FUNC_RAW.MH_RXFQ1_ENQ** can also be used. They indicate a new message was enqueued into the RXFQ0/1.

When an external DMA controller is used for dequeue instead of the host interface, the DMA request signals **RXFQ0_RREQ** (RX FIFO Queue 0 read request from VBM) or **RXFQ1_RREQ** (RX FIFO Queue 1 read request from VBM) shall be used for monitoring status changes. The RXFQ0/1_RREQ are raised by VBM if the RXFQ0/1 is not empty and there is valid data to be read by the host. RXFQ0/1_RREQ is raised only in FMM.

See more info in [1], chapter Transfer Request Signals

- **Step 2: Start dequeuing.**

To start dequeuing a message from the RXFQ, read the message header **R0 (the first word)** from **VB start address**. This address is **0x03000** for **RXFQ0** or **0x04000** for **RXFQ1**. This action triggers the VBM to initiate the dequeue process.

The RXFQ handling module within the VBM converts the virtual buffer address to the actual LMEM address where the message will be dequeued.

After that, read subsequent words, beginning with R1, linearly at increments of +4 from the start address.

The VBM calculates a trigger address based on the frame type (determined by the FDF and XLF bits in R0), and the RTR and DLC bits in R1. This trigger address corresponds to the last word of the dequeued message.

After the trigger address is accessed, the next expected address is the start address of the VB again.

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 2: 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 (Dequeue) with the size of the largest possible RX message (maximum 2068 bytes), regardless of the actual message length. The VBM tolerates this by simply discarding these accesses. The status flags **MH_STS0.RXFQ0_VB_NO_SA** or **MH_STS0.RXFQ1_VB_NO_SA** will be updated, when this occurs.

General Flowcharts

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

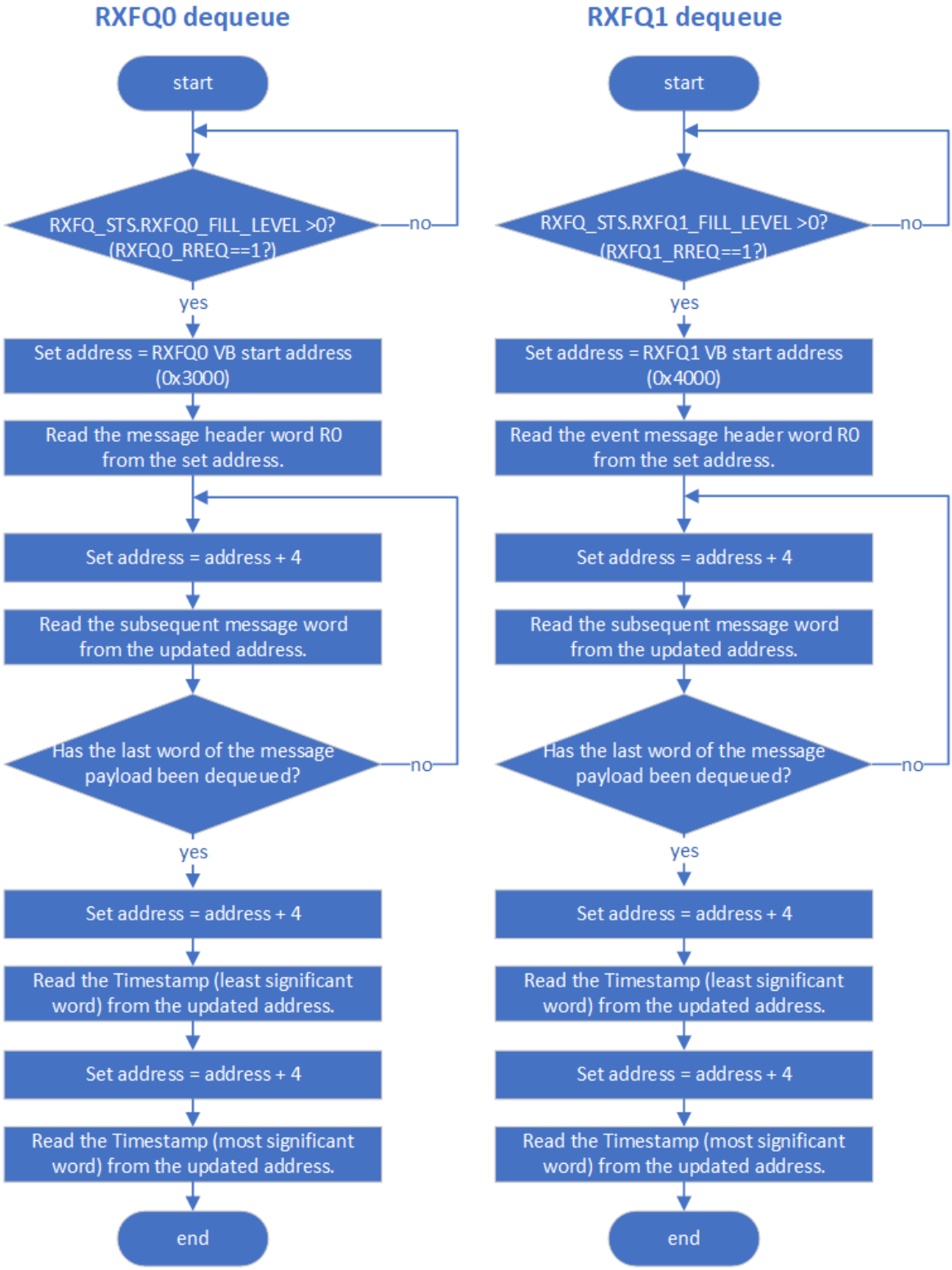

Figure 3 ：Dequeue flow diagram for RXFQ0 and RXFQ1

Example: Dequeue messages from RXFQ0 and RXFQ1

RXFQ0:

Message 1:	A CAN FD frame with DLC=4, trigger address will be at word R4(0x3010).
Message 2:	A CAN XL frame with DLC=6, trigger address will be at word R6(0x3018).

RXPQ1:

Message 3:	A CAN FD frame with DLC=4, trigger address will be at word R4(0x4010).
Message 4:	A CAN XL frame with DLC=6, trigger address will be at word R6(0x4018).

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

Step	Description	Dequeue from RXFQ0	Dequeue from RXFQ1
1	Ensure there is a message available for dequeue.	**RXFQ_STS. RXFQ0_FILL_LEVEL >** 0? (RXFQ0_RREQ==1?)	**RXFQ_STS. RXFQ1_FILL_LEVEL** > 0? (RXFQ1_RREQ==1?)
2	If message is available, proceed with dequeuing; otherwise, wait until message becomes available.		
3	Start dequeuing **msg1/msg3** at the VB start address.	Read R0 (header word 0) from address **0x3000**	Read R0 (header word 0) from address **0x4000**
4	Continue dequeuing **msg1/msg3** the subsequent word	Read R1 (header word 1) from address **0x3004**	Read R1 (header word 1) from address **0x4004**
5	Continue dequeuing **msg1/msg3** the last payload word	Read R2 (payload byte 0 – 3) from address **0x3008**	Read R2 (payload byte 0 – 3) from address **0x4008**
6	Continue dequeuing **msg1/msg3** the Timestamp	Read R3 (low significant word) from address **0x300C**	Read R3 (low significant word) from address **0x400C**
7	Continueue dequeuing **msg1/msg3** the Timestamp	Read R4 (high significant word) from address **0x3010** = trigger address	Read R4 (high significant word) from address **0x4010** = trigger address
8	Ensure there is a message available for dequeue.	**RXFQ_STS. RXFQ0_FILL_LEVEL** ==0? (RXFQ0_RREQ==1?)	**RXFQ_STS. RXFQ1_FILL_LEVEL** ==0? (RXFQ1_RREQ==1?)

Step	Description	Dequeue from RXFQ0	Dequeue from RXFQ1
9	If message is available, proceed with dequeuing; otherwise, wait until message becomes available.		
10	Start dequeuing **msg2/msg4** at the VB start address.	Read R0 (header word 0) from address **0x3000**	Write R0 (header word 0) from address **0x4000**
11	Continue dequeuing **msg2/msg4** the subsequent word	Read R1 (header word 1) from address **0x3004**	Read R1 (header word 1) from address **0x4004**
12	Continue dequeuing **msg2/msg4** the subsequent word	Read R2 (acceptance field) from address **0x3008**	Read R2 (acceptance field) from address **0x4008**
13	Continueue dequeuing **msg2/msg4** the subsequent word	Read R2 (payload byte 0 – 3) from address **0x300C**	Read R2 (payload byte 0 – 3) from address **0x400C**
14	Continue dequeuing **msg2/msg4** the last payload word	Read R3 (payload byte 4 – 7) from address **0x3010** note: since DLC = 6, byte 7 is read but not used.	Read R3 (payload byte 4 – 7) from address **0x4010** note: since DLC = 6, byte 7 is read but not used.
15	Continue dequeuing **msg2/msg4** the Timestamp	Read R5 (low significant word) from address **0x3014**	Read R5 (low significant word) from address **0x4014**
16	Continue dequeuing **msg2/msg4** the Timestamp	Read R6 (high significant word) from address **0x3018** = trigger address	Read R6 (high significant word) from address **0x4018** = trigger address

Table 3: Dequeue messages from RXFQ0 and RXFQ1

5 RX Filtering Configuration

RX filtering screens all incoming messages based on their content. This content can include identifiers, CAN frame types (Classical CAN, CAN FD, or CAN XL), the Acceptance Field for CAN XL, or the first payload word for Classical CAN and CAN FD.

RX Filters can be configured to accept or reject messages, thereby reducing the number of received messages that need to be processed by the software.

One Filter Element consists of a Filter Element Configuration (FEC) and its associated one or two Reference Pairs (REFP). Each Reference Pair consists of a 32-bit Reference Value (REFV) and a 32-bit Reference Mask (REFM).

The XS_CAN supports up to **127 RX Filter** Elements and up to **254 Reference-Mask Pairs.**

5.1 Global RX Filter configuration register (RX_FILTER_CFG)

Bit field and Bit position	**Description**
FE_NUM [6..0]	**Filter Element Number** defines the number of RX Filter Elements, with a valid range of 0 to 127. If set to '0', no Filter Elements are used. In this scenario: • If the **ANMF** bit is set to '1', all received message will be put into the **DEFAULT_FIFO**. • Otherwise (if the **ANMF** bit is set to '0'), all received messages will be dropped.
ANMF [7]	**Accept Non-Matching Frames** • If set to ‘1’, non matching messages are accepted and stored into the **DEFAULT_FIFO**. • If set to ‘0’, non matching messages are rejected.
AFAB [8]	**Accept Filter Abort** (The filtering was aborted before finding a result. The filtering was not completed on time.) • If set to ‘1’, messages for which filtering was aborted before finding a result, are accepted and stored into default Rx FIFO • If set to ‘0’, messages for which filtering was aborted before finding a result, are rejected.
DEFAULT_FIFO [9]	defines the **default FIFO** 0 = **RXFQ0** 1 = **RXFQ1**

Table 4: Global RX Filter configuration register

Summary of RX Filtering behaviour as configured in RX_FILTER_CFG

Configuration				RX Filtering behavior		
FE_NUM	**ANMF**	**AFAB**	**DEFAULT_FIFO**	**All** RX messages		
0	0	X	X	➡ dropped (default)		
0	1	X	0	➡ RXFQ0		
0	1	X	1	➡ RXFQ1		
Configuration				**RX Filtering behaviour**		
FE_NUM	**ANMF**	**AFAB**	**DEFAULT_FIFO**	**filter match**	**filter not match**	**filtering not complete on time (abort)**
>0	0	0	X	➡ RXFQ according to FEC	➡ dropped	➡ dropped
>0	0	1	0	➡ RXFQ according to FEC	➡ dropped	➡ RXFQ0
>0	0	1	1	➡ RXFQ according to FEC	➡ dropped	➡ RXFQ1
>0	1	0	0	➡ RXFQ according to FEC	➡ RXFQ0	➡ dropped
>0	1	0	1	➡ RXFQ according to FEC	➡ RXFQ1	➡ dropped
>0	1	1	0	➡ RXFQ according to FEC	➡ RXFQ0	➡ RXFQ0
>0	1	1	1	➡ RXFQ according to FEC	➡ RXFQ1	➡ RXFQ1

Table 5: Rx Filtering Behavior

5.2 Filter Element Configuration

RX Messages consist of a Message Header and a Message Payload [1] describes the RX Message Header's structure. The Message Header's data structure depends on the CAN Frame Format (CAN CC, CAN FD, CAN XL).

The RX Message Header Word0 (R0), the RX Message Header Word2 (R2) or the RX Message Payload Word0 (RD0) can be selected for RX filtering.

8-bit **Filter Element Configuration** (FEC) consists of

Bit field and Bit position	**Description**
EN [0]	Filter Element Configuration enable • If set to '1', the comparison of this FEC is performed. • If set to '0', this FEC is skipped, and the next FEC is fetched.
FIFO [1]	The RXFQ for storing accepted RX messages on match • If set to '0', ➡ RXFQ0 • If set to '1', ➡ RXFQ1
IRQ [2]	Interrupt Request on match • If set to '1', Interrupt generated • If set to '0', Interrupt not generated
ALERT [3]	Alert tag on match • If set to '1', R1.ALERT bit of the stored RX message is set to '1'. • If set to '0', R1.ALERT bit of the stored RX message is not set to '1'.
AR [4]	Accept or Reject on match • If set to '0', ➡ Accept • If set to '1', ➡ Reject

NC [5]	Number of Comparison
	• If set to '0', ➡ 1 comparison
	• If set to '1', ➡ 2 comparisons
	  • If **WI0** ≠ **WI1**, both of the comparisons much match.
	  • If **WI0** = **WI1**, one of the comparisons match is enough.

WI0 [6]	Word Index 0 • If set to '0', the R0 is used for the 1st comparison. • If set to '1', the R2/RD0 is used for the 1st comparison.
WI1 [7]	Word Index 1 • If set to '0', the R0 is used for the 2nd comparison. • If set to '1', the R2/RD0 is used for the 2nd comparison.

Table 6: Filter Element Configuration

Summary of the action on match as configured in FEC

1 Comparison							
Configuration				RX Filtering behavior			
NC	**AR**	**WI0**	**WI1**	**REFP compared with**			**Action on match**
0	0	0		R0			accept
0	1	0		R0			reject
0	0	1		R2/RD0			accept
0	1	1		R2/RD0			reject
2 Comparisons							
Configuration				RX Filtering behavior			
AR	**AR**	**WI0**	**WI1**	**1st REFP compared with**	**2nd REFP compared with**	**Condition**	**Action on match**
1	0	0	1	R0	R2/RD0	**both** of the comparisons must match	accept
1	0	1	0	R2/RD0	R0		accept
1	1	0	1	R0	R2/RD0		reject
1	1	1	0	R2/RD0	R0		reject
1	0	0	0	R0	R0	**one** of the comparisons matches	accept
1	0	1	1	R2/RD0	R2/RD0		accept
1	1	0	0	R0	R0		reject
1	1	1	1	R2/RD0	R2/RD0		reject

Table 7: Summary of action filter match

5.3 Comparison with Reference Value-Mask Pair

The RX-filter match condition is based on the following formula.
If (($\boldsymbol{R_i}$ XOR $\boldsymbol{REFP.value}$) AND $\boldsymbol{REFP.mask}$) = 0, then the filter element matches.

Details:

1) The 32 bits of R_i "**bitwise exclusive or**" with the 32 bits of the Reference Pair Value
2) The result of 1) "**bitwise and**" with the 32 bits of the Reference Pair Mask
3) The result of 2) determines the match condition.
   - if the result = '0' ➡ match
   - if the result = '1' ➡ not match

Note: $\boldsymbol{R_i}$ can be **R0** or **R2/RD0** based on the FEC.

This means, the bit positions with **zeros** in the **Reference Pair Mask** are ignored, and the bit positions with **ones** in the **Reference Pair Mask** are considered for comparison with **Reference Pair Value**, and these bit positions must have the same value as they are in **Reference Pair Value**.

As short summary,
if Mask = '0' ➡ it is considered as don't care,
if Mask = '1' ➡ it´s considered as must match.

RX Filtering process is complete if there is an outcome. It can be either a match occurred or no match occurred after going through the entire list.

If the filtering process is not completed on time with a result, then its a filtering abort condition. If the filtering is aborted without an end result, then decisions to store the message received or to discard it depends on the configuration of **AFAB bit** in the register **RX_FILTER_CFG**.

5.4 RX Filtering Path

See the following flow diagram of the RX-Filtering.

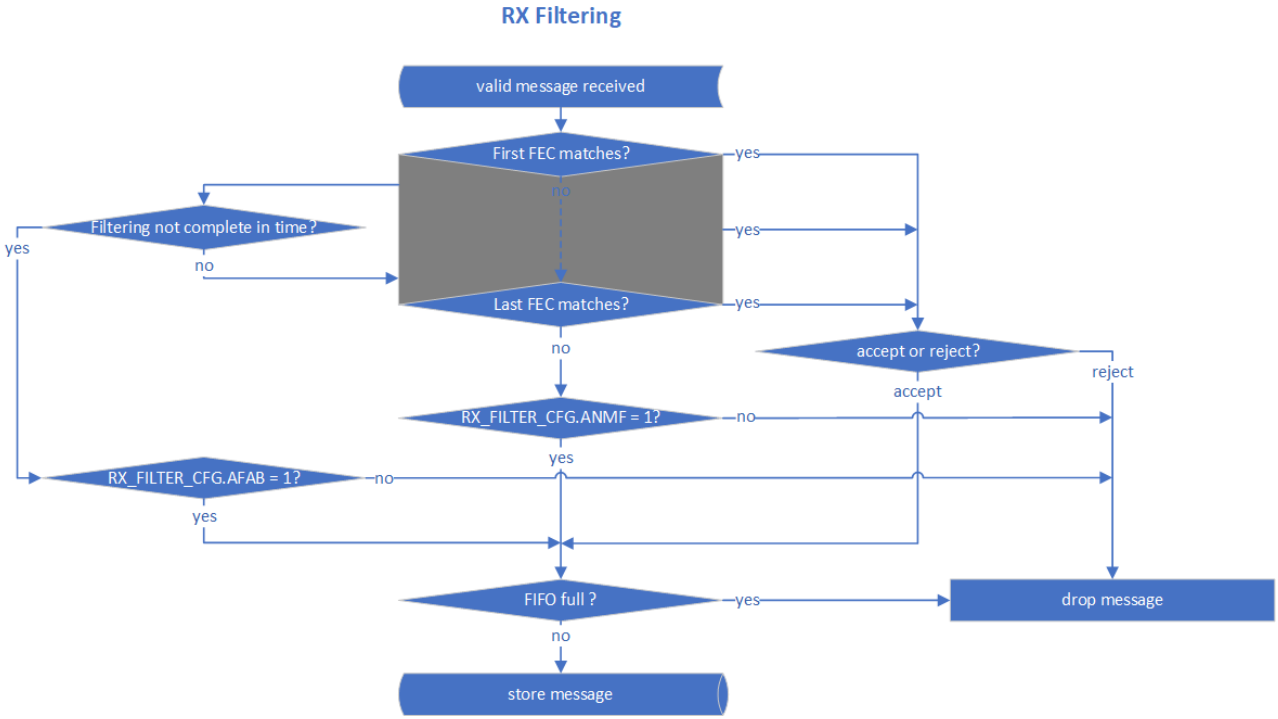

Figure 4: Flow diagram RX- filtering

5.5 RX Filter rule recommendations

The maximum number of RX Filter Elements directly impacts overall MH performance. More elements lead to longer processing times. To estimate the minimum required Host clock frequency, please refer to the Minimum Host Clock Calculator provided as Excel-Sheet [3]. See also chap. RX-Filter Processing Time in [1].

It is recommended to use the following rules for sorting the RX Filter element configurations. This provides two advantages.

- Reduced filtering time for critical cases.
- Increased number of supported RX Filter configurations.

Rule order	Rule type	Rule description
1	Sort by Data Field Size (payload)	short data field before large data field
2	Sort by Frame Format	first CAN FD, then CAN XL and then Classical CAN
3	Sort by the number of Comparison	1 comparison before 2 comparisons

Table 8: RX Filter rules recommendations

Example for a mixed network with FD and XL messages:

1. CAN FD messages with short payload up to 8 byte with 1 comparison, sorted by data field size
2. CAN XL messages with short payload up to 8 byte with 1 comparison, sorted by data field size
3. CAN XL messages with short payload up to 8 byte with 2 comparisons, sorted by data field size
4. CAN FD messages with longer payload
5. CAN XL messages with longer payload

5.6 Writing RX-Filter into LMEM

To write the FECs and REFPs to the LMEM, the Host uses the **LMEM Transparent Address** (LMEM TA) range **0x40000 to 0x7FFFC**. This range maps directly to LMEM addresses **0x00000 to 0x3FFFC** within the 256KB LMEM. (See VBM memory map.) See also [1], chapter 1.5.6.4.1

See the following needs for the RX-Filter writing. It is important to consider the structure of the FECs and REFPs within the LMEM. See in the following Figure 5: Rx Filter Elements example in LMEM.

Important notes and needs:

- Data has to be written in 32-bit words, not 8-bit bytes.
- The FECs shall be **stored back to back** in continuous way in the LMEM starting at the RX Filter Elements **Start Address** configured in **RX_FILTER_LMEM.SA**.
- All the **REFPs (REFVs and REFMs)** are appended imediately after the FECs section in the LMEM.

- The list of REFPs has to start at the next 32-bit word directly after the last FEC.
- The first REFP (REFV and REFM) corresponds to the first comparison of the first FEC.
    - If the FEC contains 2 comparisons, the next REFP (REFV and REFM) corresponds to the second comparison of the same FEC. Subsequently, the next memory location is allocated to the REFP of the next FEC in the ordering.
    - If the FEC contains 1 comparison, only one REFP is stored into LMEM. The subsequent memory location is then allocated to the REFP of the next FEC in the ordering.
    - so on and so forth

Example: writing RX Filters into the LMEM.

10 RX Filter Elements
RX_FILTER_LMEM.SA = 0x00000

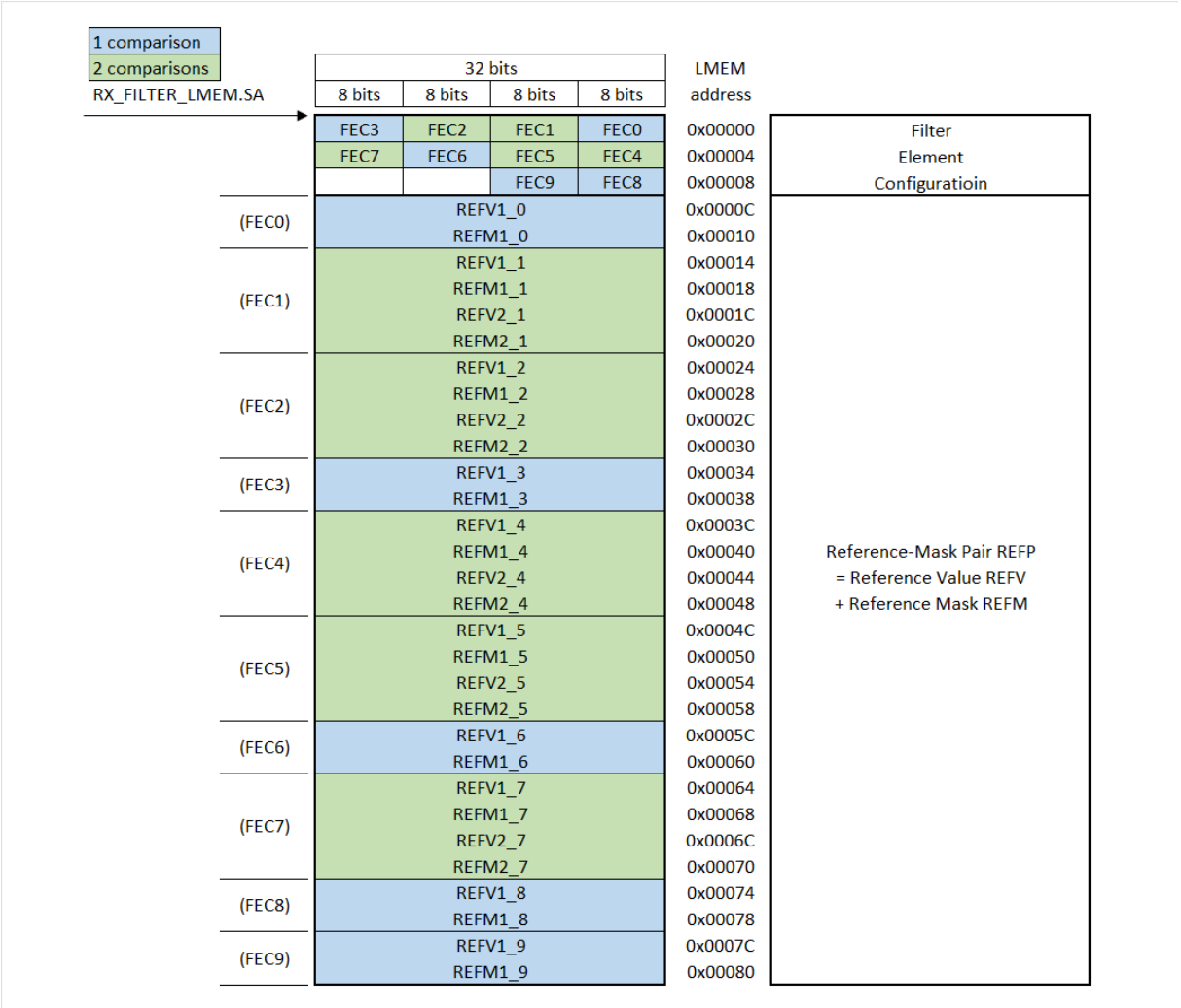

Figure 5: Rx Filter Elements example in LMEM

See the following memory map (Host address to LMEM address) of RX-Filter in LMEM.

Host write address	Host write data	actual address in LMEM
0x40000	FEC3 – FEC0	0x00000
0x40004	FEC7 – FEC4	0x00004
0x40008	FEC9 – FEC8	0x00008
0x4000C	FEC0.REFV1	0x0000C
0x40010	FEC0.REFM1	0x00010
0x40014	FEC1.REFV1	0x00014
0x40018	FEC1.REFM1	0x00018
0x4001C	FEC1.REFV2	0x0001C
0x40020	FEC1.REFM2	0x00020
0x40024	FEC2.REFV1	0x00024
0x40028	FEC2.REFM1	0x00028
0x4002C	FEC2.REFV2	0x0002C
0x40030	FEC2.REFM2	0x00030
0x40034	FEC3.REFV1	0x00034
0x40038	FEC3.REFM1	0x00038
0x4003C	FEC4.REFV1	0x0003C
0x40040	FEC4.REFM1	0x00040
0x40044	FEC4.REFV2	0x00044
0x40048	FEC4.REFM2	0x00048
0x4004C	FEC5.REFV1	0x0004C
0x40050	FEC5.REFM1	0x00050
0x40054	FEC5.REFV2	0x00054
0x40058	FEC5.REFM2	0x00058
0x4005C	FEC6.REFV1	0x0005C
0x40060	FEC6.REFM1	0x00060
0x40064	FEC7.REFV1	0x00064
0x40068	FEC7.REFM1	0x00068
0x4006C	FEC7.REFV2	0x0006C
0x40070	FEC7.REFM2	0x00070
0x40074	FEC8.REFV1	0x00074
0x40078	FEC8.REFM1	0x00078
0x4007C	FEC9.REFV1	0x0007C
0x40080	FEC9.REFM1	0x00080

Table 9: Adress memory map of RX-Filter in LMEM

Example: Detailed Rx-Filter configuration

RX Filter Configurations

RX_FILTER_CFG. **DEFAULT_FIFO** = 1 RXFQ1 is a default FIFO
AFAB = 1 filtering not completed in time, accepted to RXFQ1
ANMF = 0 non-matching frame rejected
FE_NUM = 8 8 Filter Elements in total

	FEC		REFP						behavior		
	WI1 WI0 NC AR AERT IRQ FIFO EN		CC/FD R0 / XL R0	FDF XLF XTD	BaseID[28:18]	ExtID[17:0]			condition of match	action	stored in
					PriorityID[28:18]	RRS SEC	VCID[7:0]	SDT[7:0]			
			CC/FD RD0 / XL R2		Data Word0 / Acceptance Field						
	7 6 5 4 3 2 1 0				31 30 29 28 27 26 25 24 23 22 21 20 19 18	17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0					
RX Filter 0	FEC0	REFV1 (R0)			0 0 0 0 0 1 0 0 0 1 0 1 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			[BaseID range 0x114 - 0x117]	reject	-
	0 0 0 1 0 0 0 1	REFM1 (R0)			0 0 1 1 1 1 1 1 1 1 1 1 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0					
RX Filter 1	FEC1	REFV1 (R0)			0 0 0 0 0 1 0 0 0 1 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			[BaseID range 0x110 - 0x11F]	accept	➡ RXFQ1
	0 0 0 0 0 0 1 1	REFM1 (R0)			0 0 1 1 1 1 1 1 1 1 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0					
RX Filter 2	FEC2	REFV1 (R0)			0 0 0 0 1 0 0 1 0 1 0 1 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			[BaseID = 0x255]	accept	➡ RXFQ1
	0 0 1 0 0 0 1 1	REFM1 (R0)			0 0 1 1 1 1 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			or		
		REFV2 (R0)			0 0 1 0 1 0 1 0 1 0 1 0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			[BaseID = 0x2AA]		
		REFM2 (R0)			0 0 1 1 1 1 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0					
RX Filter 3	FEC2	REFV1 (R0)			0 0 0 0 1 1 0 0 0 0 1 0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			[BaseID = 0x3xA]	accept	➡ RXFQ1
	0 0 1 0 0 0 1 1	REFM1 (R0)			0 0 1 1 1 1 0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			or		
		REFV2 (R0)			0 0 1 0 1 1 0 0 0 0 1 0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			[ExtID where its BaseID = 0x3xA]		
		REFM2 (R0)			0 0 1 1 1 1 0 0 0 0 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0					
RX Filter 4	FEC3	REFV1 (R0)			0 0 1 0 0 0 0 0 0 0 0 1 0 0	0 0 0 0 1 0 1 1 1 0 1 0 0 1 1 0			[ExtID = 0x10BA5E]	reject	-
	0 0 0 1 0 0 0 1	REFM1 (R0)			0 0 1 1 1 1 1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1					
RX Filter 5	FEC4	REFV1 (R0)			1 1 0 1 1 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			[Frame Format = XL] and [BaseID = 0x6xx] and [SDT = 4]	accept	➡ RXFQ0
	0 0 0 0 0 0 0 1	REFM1 (R0)			1 1 0 1 1 1 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1					
RX Filter 6	FEC5	REFV1 (R0)			1 1 0 1 1 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1			[Frame Format = XL] and [BaseID = 0x6xx] and [SDT = 5]	accept	➡ RXFQ0
	1 0 1 0 0 0 0 1	REFM1 (R0)			1 1 0 1 1 1 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1			and		
		REFV2 (R2)			1 0 1 1 1 1 1 0 1 1 1 0 1 1	1 1 0 0 1 0 1 0 1 1 1 1 1 1 1 0			[Acceptance Field = 0xBEEFCAFE]		
		REFM2 (R2)			1 1 1 1 1 1 1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1					
RX Filter 7	FEC6	REFV1 (R0)			0 0 0 1 1 1 0 1 1 1 0 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			[Frame Format = CC, FD] and [BaseID = 0x777]	accept	➡ RXFQ1
	1 0 1 0 0 0 1 1	REFM1 (R0)			0 1 1 1 1 1 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0			and		
		REFV2 (RD0)			0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 0 1 0 1 0 1 0 0 1 0 1 0 1			[Data Byte0 = 0x55, Byte1 = 0xAA]		
		REFM2 (RD0)			0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1 1 1 1 1 1 1					

Table 10: Example with a detailed Rx-Filter configuration

See the following table with the filtering results.

CAN Message Identifier	Frame Format	SDT	Acceptance Field	Payload Byte0	Payload Byte1	Payload Byte2	Payload Byte3	accept/reject	maching filter	stored in
0x110	CC	-	-	0xXX	0x22	0x33	0x44	accept	RX Filter 1	RXFQ1
0x111	FD	-	-	0x11	0x22	0x33	0x44	accept	RX Filter 1	RXFQ1
0x112	XL	4	0x00000000	0x11	0x22	0x33	0x44	accept	RX Filter 1	RXFQ1
0x113	CC	-	-	0x11	0x22	0x33	0x44	reject	RX Filter 0	-
0x114	FD	-	-	0x11	0x22	0x33	0x44	reject	RX Filter 0	-
0x115	XL			0x11	0x22	0x33	0x44	reject	RX Filter 0	-
0x116	CC	-	-	0x11	0x22	0x33	0x44	reject	RX Filter 0	-
0x117	FD	-	-	0x11	0x22	0x33	0x44	reject	RX Filter 0	-
0x118	XL			0x11	0x22	0x33	0x44	accept	RX Filter 1	RXFQ1
0x11A	CC	-	-	0x11	0x22	0x33	0x44	accept	RX Filter 1	RXFQ1
0x11C	FD	-	-	0x11	0x22	0x33	0x44	accept	RX Filter 1	RXFQ1
0x120	XL	5	0xBEEFCAFE	0x11	0x22	0x33	0x44	reject	-none-	-
0x255	CC	-	-	0x11	0x22	0x33	0x44	accept	RX Filter 2	RXFQ1
0x2AA	FD	-	-	0x11	0x22	0x33	0x44	accept	RX Filter 2	RXFQ1
0x2FF	XL	6	0xBEEFCAFE	0x11	0x22	0x33	0x44	reject	-none-	-
0x309	CC	-	-	0x11	0x22	0x33	0x44	reject	-none-	-
0x30A	FD	-	-	0x11	0x22	0x33	0x44	accept	RX Filter 3	RXFQ1
0x31B	XL	5	0xCAFECAFE	0x11	0x22	0x33	0x44	reject	-none-	-
0x3FA	CC	-	-	0x11	0x22	0x33	0x44	accept	RX Filter 3	RXFQ1
0x0C28FFFF	FD	-	-	0x11	0x22	0x33	0x44	accept	RX Filter 3	RXFQ1
0x0C380000	CC	-	-	0x11	0x22	0x33	0x44	reject	-none-	-
0x0010BA5E	FD	-	-	0x11	0x22	0x33	0x44	reject	RX Filter 4	-
0x600	CC	-	-	0x11	0x22	0x33	0x44	reject	-none-	-
0x610	FD	-	-	0x11	0x22	0x33	0x44	reject	-none-	-
0x630	XL	3	0xCAFECAFE	0x11	0x22	0x33	0x44	reject	-none-	-
0x645	XL	4	0xBEEFCAFE	0x11	0x22	0x33	0x44	accept	RX Filter 5	RXFQ0
0x666	XL	5	0xCAFECAFE	0x11	0x22	0x33	0x44	reject	-none-	-
0x6FF	XL	5	0xBEEFCAFE	0x11	0x22	0x33	0x44	accept	RX Filter 6	RXFQ0
0x777	CC	-	-	0x55	0xAA	0x55	0xAA	accept	RX Filter 7	RXFQ1
0x777	CC	-	-	0x11	0x22	0x33	0x44	reject	-none-	-
0x777	FD	-	-	0x55	0xAA	0x55	0xAA	accept	RX Filter 7	RXFQ1
0x777	XL	5	0xAA55AA55	0x55	0xAA	0x55	0xAA	reject	-none-	-

Table 11: Rx-Filter example results

6 Interrupt Flags

6.1 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_RXFQ1_DEQ	A message is dequeued from RXFQ1.
MH_RXFQ0_DEQ	A message is dequeued from RXFQ0.
MH_RXFQ1_ENQ	A message is enqueued into RXFQ1.
MH_RXFQ0_ENQ	A message is enqueued into RXFQ0.

6.2 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.

6.3 Safety Raw Event Status register (SAFETY_RAW)

See the following relevant flags to message reception and filtering.

Bit field/flag	Description
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.

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

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

7 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()`
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.
Name:	`xs_can_app_note_tx_rx_transceiver_mode_switch_on()`
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 CAN XL message when the **Transceiver Mode Switching** is enabled. In this mode, Pulse Width Mode (PWM) is applied on the TXD pin of the transceiver.
Name:	`xs_can_app_note_rx_filtering()`
File:	`../xs_can/app_notes/app_note_rx_filtering.c`
Description:	The example demonstrates how to configure the filtering for the messages reception.
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`
	`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.