Automotive Electronics

User Manual
X_CAN

Table of contents

page

1. X_CAN

The X_CAN is the new CAN Communication Controller IP supporting CAN XL protocol. It can be integrated as part of a SoC. It is described in VHDL on RTL level, prepared for synthesis. The X_CAN performs communication according to ISO11898-1:2015 and CiA610-1.

The X_CAN can be connected to a wide range of HOST CPUs via its 32bit interface. The clock domain concept allows the separation between the high precision CAN clock and the HOST clock, which may be generated by an FM-PLL.

1.1 Features

- Conform with ISO11898-1:2015 and CiA610-1
- CAN CC (CAN classic) with up to 8 data bytes and up to 1Mbit/s
- CAN FD (CAN flexible data rate) with up to 64 data bytes and up to 8Mbit/s
- CAN XL (CAN extended data length) with up to 2048 data bytes and up to 20Mbit/s
- 1 Priority Queue, up to 32 slots, priority based on the arbitration field of the CAN frame
- 8 TX FIFO queues, each with up to 1024 messages
- 8 RX FIFO queues, each with up to 1024 messages
- TX message filtering with up to 16 filter elements
- RX message filtering with up to 255 filter elements, while each can compare one 32bit word (The actual usable number of filter elements depends on CAN clock frequency, CAN bit rate, and Local Memory performance)

- Internal DMA engine, X_CAN is the DMA master for message handling
  - Message storage in system memory
  - Low CPU impact, any accesses to/from the system memory are done using the internal DMA engine (less interrupts needed)

- Requires only small local memory
  - Approx 4Kbytes for up to 255 RX filter elements
  - Multiple X_CAN can share the same Local Memory

- Maskable module interrupts with three categories: Functional, Functional Error and Safety
- Three clock domains (HOST, CAN, TIMEBASE clock domains)
- CAN Error Logging
- Fault Injection Module
- Programmable loop-back test mode
- Power-down support
- AXI4-Lite slave interface (*HOST_AXI*) (compliant to AMBA 4 ARM Ltd protocol, see [5])
- AXI4 master DMA interface (*DMA_AXI*) (compliant to AMBA 4 ARM Ltd protocol, see [5])
- AXI4 master Local Memory interface (*MEM_AXI*) (compliant to AMBA 4 ARM Ltd protocol, see [5])

1.2 Block Diagram

The following block diagram shows the internal structure of the X_CAN IP and the interconnection to the SoC. The chapter 'Functional Description' provides detailed information to this figure.

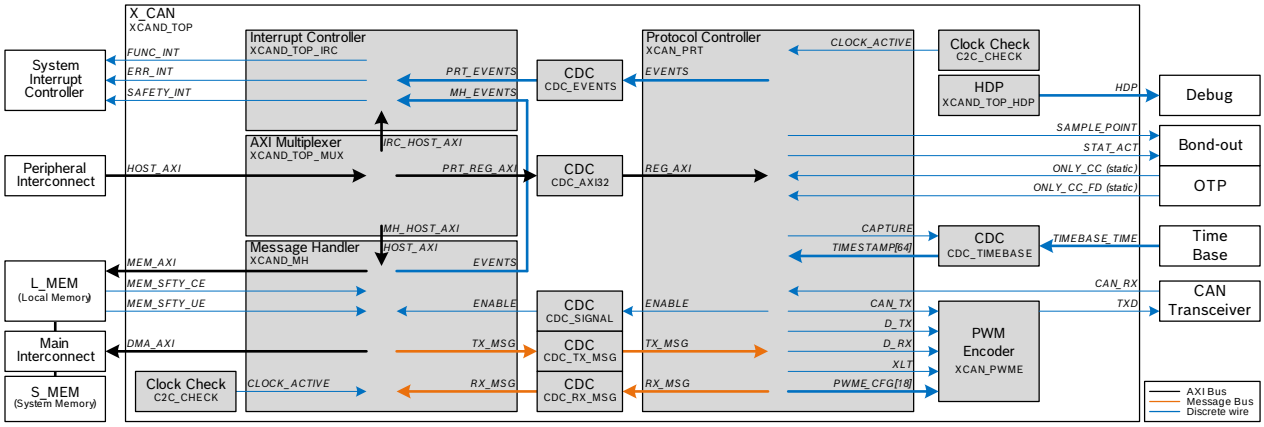

Figure: XCAND_TOP

1.3 TOP - Top Level

1.3.1 Software Interface

The registers banks of the modules are memory mapped by the AXI Multiplexer as depicted in the following figure.

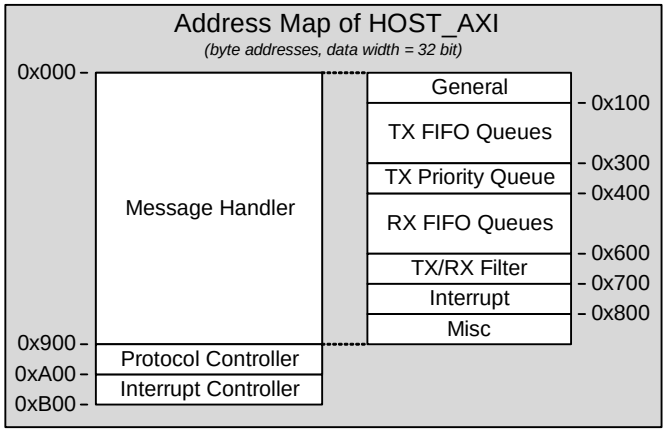

Figure: XCAND_TOP memory map

Detailed register description of Message Handler (MH), Protocol Controller (PRT) and Interrupt Controller (IRC) can be found in the 'Software Interface' section of the respective chapters.

1.3.2 Functional Description

The top level of the X_CAN IP embeds all digital blocks required for communication on one CAN bus. To start up the X_CAN IP, the Message Handler and the Protocol Controller must be configured beforehand. The Message Handler must be started first (writing a 1 to the **MH_CTRL.START** bit) and afterwards, the Protocol Controller must be started (writing a 1 to the **CTRL.STRT** bit). Detailed descriptions are provided by PRT and MH chapters later in this document.

The blocks of the top level are described in the following chapters.

1.3.2.1 AXI Multiplexer

The X_CAN embeds three register banks, containing configuration, control, status, and event information. They are in the modules Message Handler, Protocol Controller and Interrupt controller and are accessible via peripheral interconnect through *HOST_AXI* interface and IP internal AXI Multiplexer.

- When an access is performed to a non-mapped register in the address range, a SLVERR is provided as a response.
- When a read access to write-only registers or a write access to read-only registers is performed, a SLVERR is provided as a response.
- When a read or write access is performed outside the address range of the Interrupt Controller and the Message Handler and the Protocol Controller, a DECERR is provided as a response.
- When a write access is performed and write strobe signals are not set to 0b1111, a SLVERR is provided as a response. Only 32bit write access is allowed.

1.3.2.2 Message Handler

All functions concerning the storage and scheduling of CAN messages are implemented in the Message Handler (MH). The TX path supports the storage of CAN messages in 8 TX FIFO Queues and one TX Priority Queue. The RX path provides 8 RX FIFO Queues. FIFO data is physically stored in System Memory (S_MEM) and managed by descriptors. TX and RX Filters provide methods to accept or deny CAN Messages and (for RX only) to determine the target RX FIFO for data storage.

The MH will be configured and controlled by HOST CPU via *HOST_AXI* interface. CAN messages and descriptors are transported between System Memory and local memory autonomously by an internal DMA, which is connected to *DMA_AXI*. For fast access, the MH needs a Local Memory (L_MEM) which is connected via *MEM_AXI* interface. Depending on the chosen SoC integration, multiple X_CAN IPs can share the same local RAM.

Detailed description is provided by the MH section.

1.3.2.3 Protocol Controller

The Protocol Controller (PRT) performs CAN communication as specified in ISO 11898-1:2015 (Classical CAN and CAN FD) and in CiA610-1 (CAN XL). The bitrate can be configured to values up to 20MBit/s at a clock speed of 160MHz, depending on the used semiconductor technology. For the connection to the physical layer, additional transceiver hardware is required.

The PRT does not provide internal buffering of frames, so that data must be transferred by IP internal Message Busses in 32 bit slices in real-time while (de)-serializing them on the CAN Bus. Thus, single data transfers at the internal Message Busses are closely time-synchronized to the schedule at the CAN bus.

Detailed description is provided by PRT section.

1.3.2.4 PWME

The module PWME implements the PWM encoding as specified in [2]. When transceiver mode switching is enabled, the PWME encodes the *CAN_TX* input signal during a CAN XL frame's data phase and during ADH bit, to generate the PWM encoded output signal *TXD*.

Detailed description is provided by PWME section.

1.3.2.5 Hardware Debug Port

The X_CAN provides a 16 bit Hardware Debug Port (*HDP*), intended to be multiplexed to SoC output pins in a special X_CAN hardware debug mode. Internal signals can be multiplexed to this interface and be observed via a logic analyzer.

The use of this feature requires deep knowledge of internal behavior of the X_CAN and thus require support from the IP provider.

The internal signals are organized in pre-defined sets which are selected by **HDP.HDP_SEL**. The following tables describe the signal sets.

HDP [15:0]	HDP_SEL = 0 (MH debug port)	HDP_SEL = 1 (PRT interface signals)
15	*MH_HDP[15]*	*TX_DU*
14	*MH_HDP[14]*	*RX_DO*
13	*MH_HDP[13]*	*BUS_OFF*
12	*MH_HDP[12]*	*E_PASSIVE*
11	*MH_HDP[11]*	*E_ACTIVE*
10	*MH_HDP[10]*	*BUS_ERR*
9	*MH_HDP[9]*	*TX_EVT*
8	*MH_HDP[8]*	*RX_EVT*

HDP [15:0]	HDP_SEL = 0 (MH debug port)	HDP_SEL = 1 (PRT interface signals)
7	*MH_HDP[7]*	*STAT_ACT[1]*
6	*MH_HDP[6]*	*STAT_ACT[0]*
5	*MH_HDP[5]*	*XLT*
4	*MH_HDP[4]*	*D_RX*
3	*MH_HDP[3]*	*D_TX*
2	*MH_HDP[2]*	*SAMPLE_POINT*
1	*MH_HDP[1]*	*CAN_TX*
0	*MH_HDP[0]*	*CAN_CLK*

Detailed description of the MH Debug Port can be found in the 'Trace and Debug' section of the MH chapter.

1.3.2.6 Interrupt controller

The X_CAN IP is equipped with a central interrupt controller (IRC). It captures all events of the MH and PRT and can be configured for each event individually to interrupt the HOST CPU.

Detailed description is provided by IRC - Interrupt Controller section.

1.4 MH – Message Handler

1.4.1 Overview

The MH is located in between the main interconnect and the PRT.
It is designed to read TX CAN message data from System Memory (S_MEM) and to send them to the PRT.
On the other direction, it provides the RX CAN message data to the S_MEM when they are received by the PRT.
Status feedback is given to the SW for every CAN RX and TX messages directly in the S_MEM, avoiding register accesses.

All functions, concerning the storage and scheduling of CAN messages, are implemented in the Message Handler (MH). The TX path supports the storage of CAN messages in 8 TX FIFO Queues and one TX Priority Queue. The RX path provides 8 RX FIFO Queues. FIFO data is physically stored in S_MEM and managed by descriptors. TX and RX Filters provide methods to accept or deny CAN Messages and, for RX only, to determine the target RX FIFO for data storage.

The MH will be configured and controlled by HOST CPU via *HOST_AXI* interface. CAN messages and descriptors are transported between S_MEM and Local Memory (L_MEM) autonomously by an internal DMA, which is connected to *DMA_AXI* interface. The MH needs an L_MEM, which is connected via

MEM_AXI interface. Depending on the chosen SoC integration, multiple X_CAN can share the same L_MEM.

1.4.2 Features

- Functional and Error interrupts
- Safety interrupts
- Safety measures build-in:
    - Data path parity protection
    - Parity protection on address pointers
    - linked list descriptor protected by CRC
    - Register bank protected by CRC
    - Interface timeout protection (PRT and AXI master interfaces)
- TX message priority based on ID and IDE and SRR and RTR
- Up to 8 TX FIFO queues can be defined
- Up to 8 RX FIFO queues can be defined
- 1 Priority Queue with a programmable number of slots, limited to 32
- TX message filtering with up to 16 filter definitions
- RX message filtering with up to 255 filter definitions
- Classical CAN and CAN FD supported
- CAN XL supported
- Fully synchronous design
- Little Endian

1.4.3 Block Diagram

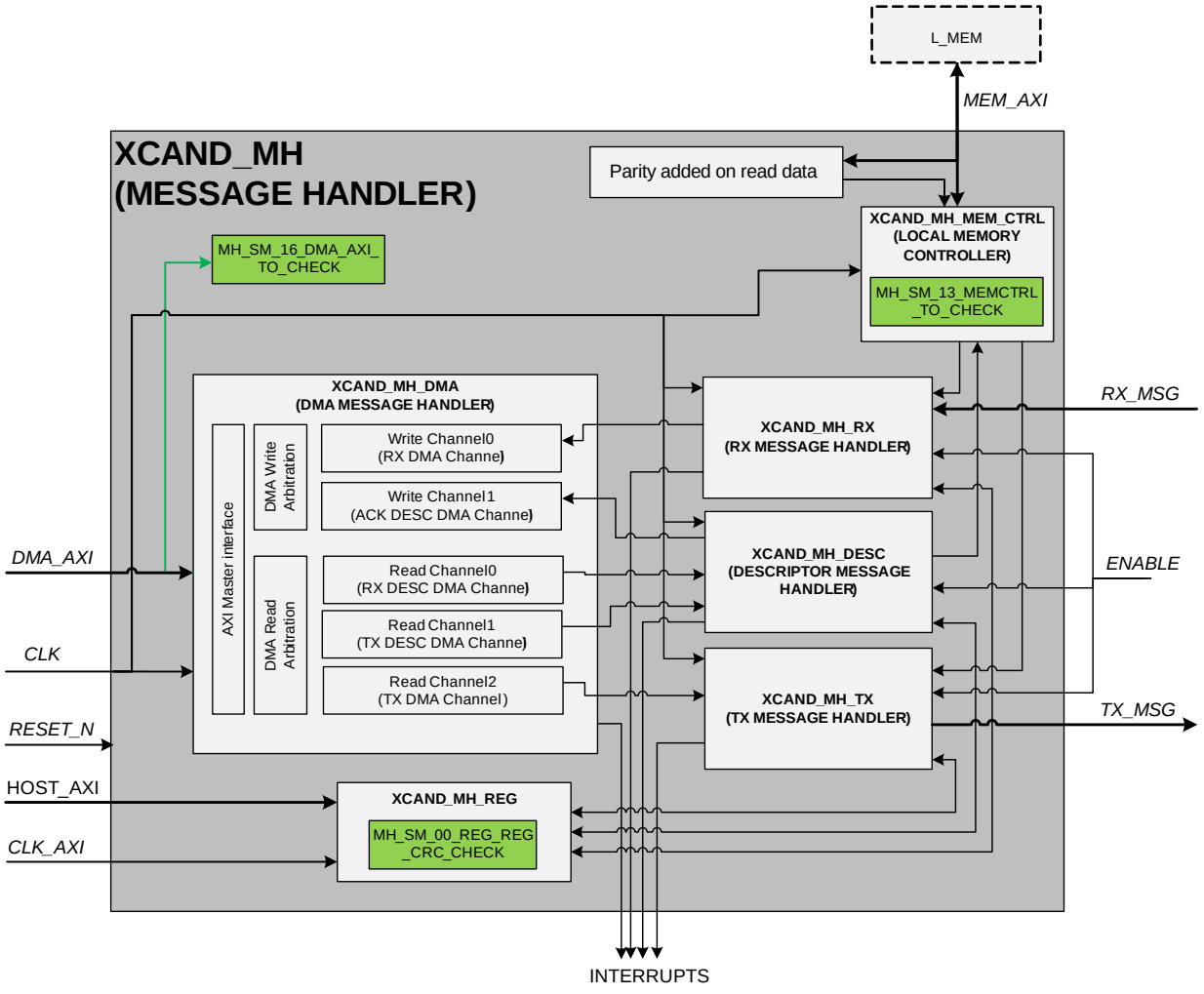

Figure: Message Handler block diagram

1.4.4 Software Interface

1.4.4.1 Register Map

Address offset	Register name	Description	Access	Initial value
0x000	VERSION	Release Identification Register	read-only	0x05600000
MH global control and status registers				
0x004	MH_CTRL	Message Handler Control register	read-write	0x00
0x008	MH_CFG	Message Handler Configuration register	read-write	0x0700
0x00C	MH_STS	Message Handler Status register	read-only	0x00
0x010	MH_SFTY_CFG	Message Handler Safety Configuration register	read-write	0x00

0x14	MH_SFTY_CTRL	Message Handler Safety Control register	read-write	0x00
0x18	RX_FILTER_MEM_ADD	RX Filter Base Address register	read-write	0x0
0x1C	TX_DESC_MEM_ADD	TX Descriptor Base Address register	read-write	0x0
0x20	AXI_ADD_EXT	AXI address extension register	read-write	0x0
0x24	AXI_PARAMS	AXI parameter register	read-write	0x0
0x028	MH_LOCK	Message Handler Lock register	read-write	0x00
TX FIFO Queues control and status registers				
0x100	TX_DESC_ADD_PT	TX descriptor current address pointer register	read-only	0x0
0x104	TX_STATISTICS	Unsuccessful and Successful message counter registers	read-write	0x0
0x108	TX_FQ_STS0	TX FIFO Queue Status register	read-only	0x0
0x10C	TX_FQ_STS1	TX FIFO Queue Status register	read-only	0x0
0x110	TX_FQ_CTRL0	TX FIFO Queue Control register 0	read-write	0x0
0x114	TX_FQ_CTRL1	TX FIFO Queue Control register 1	read-write	0x0
0x118	TX_FQ_CTRL2	TX FIFO Queue Control register 2	read-write	0x0
0x120	TX_FQ_ADD_PT0	TX FIFO Queue 0 Current Address Pointer register	read-only	0x0
0x124	TX_FQ_START_ADD0	TX FIFO Queue 0 Start Address register	read-write	0x0
0x128	TX_FQ_SIZE0	TX FIFO Queue 0 Size register	read-write	0x0
0x130	TX_FQ_ADD_PT1	TX FIFO Queue 1 Current Address Pointer register	read-only	0x0
0x134	TX_FQ_START_ADD1	TX FIFO Queue 1 Start Address register	read-write	0x0
0x138	TX_FQ_SIZE1	TX FIFO Queue 1 Size register	read-write	0x0
0x140	TX_FQ_ADD_PT2	TX FIFO Queue 2 Current Address Pointer register	read-only	0x0
0x144	TX_FQ_START_ADD2	TX FIFO Queue 2 Start Address register	read-write	0x0
0x148	TX_FQ_SIZE2	TX FIFO Queue 2 Size register	read-write	0x0
0x150	TX_FQ_ADD_PT3	TX FIFO Queue 3 Current Address Pointer register	read-only	0x0
0x154	TX_FQ_START_ADD3	TX FIFO Queue 3 Start Address register	read-write	0x0
0x158	TX_FQ_SIZE3	TX FIFO Queue 3 Size register	read-write	0x0
0x160	TX_FQ_ADD_PT4	TX FIFO Queue 4 Current Address Pointer register	read-only	0x0
0x164	TX_FQ_START_ADD4	TX FIFO Queue 4 Start Address register	read-write	0x0
0x168	TX_FQ_SIZE4	TX FIFO Queue 4 Size register	read-write	0x0
0x170	TX_FQ_ADD_PT5	TX FIFO Queue 5 Current Address	read-only	0x0

		Pointer register		
0x174	TX_FQ_START_ADD5	TX FIFO Queue 5 Start Address register	read-write	0x0
0x178	TX_FQ_SIZE5	TX FIFO Queue 5 Size register	read-write	0x0
0x180	TX_FQ_ADD_PT6	TX FIFO Queue 6 Current Address Pointer register	read-only	0x0
0x184	TX_FQ_START_ADD6	TX FIFO Queue 6 Start Address register	read-write	0x0
0x188	TX_FQ_SIZE6	TX FIFO Queue 6 Size register	read-write	0x0
0x190	TX_FQ_ADD_PT7	TX FIFO Queue 7 Current Address Pointer register	read-only	0x0
0x194	TX_FQ_START_ADD7	TX FIFO Queue 7 Start Address register	read-write	0x0
0x198	TX_FQ_SIZE7	TX FIFO Queue 7 Size register	read-write	0x0
TX Priority Queue control and status registers				
0x300	TX_PQ_STS0	TX Priority Queue Status register	read-only	0x0
0x304	TX_PQ_STS1	TX Priority Queue Status register	read-only	0x0
0x30C	TX_PQ_CTRL0	TX Priority Queue Control register 0	read-write	0x0
0x310	TX_PQ_CTRL1	TX Priority Queue Control register 1	read-write	0x0
0x314	TX_PQ_CTRL2	TX Priority Queue Control register 2	read-write	0x0
0x318	TX_PQ_START_ADD	TX Priority Queue Start Address	read-write	0x0
RX FIFO Queues control and status registers				
0x400	RX_DESC_ADD_PT	RX descriptor Current Address Pointer	read-only	0x0
0x404	RX_STATISTICS	Unsuccessful and Successful Message Received Counter	read-write	0x0
0x408	RX_FQ_STS0	RX FIFO Queue Status register 0	read-only	0x0
0x40C	RX_FQ_STS1	RX FIFO Queue Status register 1	read-only	0x0
0x410	RX_FQ_STS2	RX FIFO Queue Status register 2	read-only	0x0
0x414	RX_FQ_CTRL0	RX FIFO Queue Control register 0	read-write	0x0
0x418	RX_FQ_CTRL1	RX FIFO Queue Control register 1	read-write	0x0
0x41C	RX_FQ_CTRL2	RX FIFO Queue Control register 2	read-write	0x0
0x420	RX_FQ_ADD_PT0	RX FIFO Queue 0 Current Address Pointer	read-only	0x0
0x424	RX_FQ_START_ADD0	RX FIFO Queue 0 link list Start Address	read-write	0x0
0x428	RX_FQ_SIZE0	RX FIFO Queue 0 link list and data container Size	read-write	0x0
0x42C	RX_FQ_DC_START_ADD0	RX FIFO Queue 0 Data Container Start Address	read-write	0x0
0x430	RX_FQ_RD_ADD_PT0	RX FIFO Queue 0 Read Address Pointer	read-write	0x0
0x438	RX_FQ_ADD_PT1	RX FIFO Queue 1 Current Address Pointer	read-only	0x0

0x43C	RX_FQ_START_ADD1	RX FIFO Queue 1 link list Start Address	read-write	0x0
0x440	RX_FQ_SIZE1	RX FIFO Queue 1 link list and data container Size	read-write	0x0
0x444	RX_FQ_DC_START_ADD1	RX FIFO Queue 1 Data Container Start Address	read-write	0x0
0x448	RX_FQ_RD_ADD_PT1	RX FIFO Queue 1 Read Address Pointer	read-write	0x0
0x450	RX_FQ_ADD_PT2	RX FIFO Queue 2 Current Address Pointer	read-only	0x0
0x454	RX_FQ_START_ADD2	RX FIFO Queue 2 link list Start Address	read-write	0x0
0x458	RX_FQ_SIZE2	RX FIFO Queue 2 link list and data container Size	read-write	0x0
0x45C	RX_FQ_DC_START_ADD2	RX FIFO Queue 2 Data Container Start Address	read-write	0x0
0x460	RX_FQ_RD_ADD_PT2	RX FIFO Queue 2 Read Address Pointer	read-write	0x0
0x468	RX_FQ_ADD_PT3	RX FIFO Queue 3 Current Address Pointer	read-only	0x0
0x46C	RX_FQ_START_ADD3	RX FIFO Queue 3 link list Start Address	read-write	0x0
0x470	RX_FQ_SIZE3	RX FIFO Queue 3 link list and data container Size	read-write	0x0
0x474	RX_FQ_DC_START_ADD3	RX FIFO Queue 3 Data Container Start Address	read-write	0x0
0x478	RX_FQ_RD_ADD_PT3	RX FIFO Queue 3 Read Address Pointer	read-write	0x0
0x480	RX_FQ_ADD_PT4	RX FIFO Queue 4 Current Address Pointer	read-only	0x0
0x484	RX_FQ_START_ADD4	RX FIFO Queue 4 link list Start Address	read-write	0x0
0x488	RX_FQ_SIZE4	RX FIFO Queue 4 link list and data container Size	read-write	0x0
0x48C	RX_FQ_DC_START_ADD4	RX FIFO Queue 4 Data Container Start Address	read-write	0x0
0x490	RX_FQ_RD_ADD_PT4	RX FIFO Queue 4 Read Address Pointer	read-write	0x0
0x498	RX_FQ_ADD_PT5	RX FIFO Queue 5 Current Address Pointer	read-only	0x0
0x49C	RX_FQ_START_ADD5	RX FIFO Queue 5 link list Start Address	read-write	0x0
0x4A0	RX_FQ_SIZE5	RX FIFO Queue 5 link list and data	read-write	0x0

		container Size		
0x4A4	RX_FQ_DC_START_ADD5	RX FIFO Queue 5 Data Container Start Address	read-write	0x0
0x4A8	RX_FQ_RD_ADD_PT5	RX FIFO Queue 5 Read Address Pointer	read-write	0x0
0x4B0	RX_FQ_ADD_PT6	RX FIFO Queue 6 Current Address Pointer	read-only	0x0
0x4B4	RX_FQ_START_ADD6	RX FIFO Queue 6 link list Start Address	read-write	0x0
0x4B8	RX_FQ_SIZE6	RX FIFO Queue 6 link list and data container Size	read-write	0x0
0x4BC	RX_FQ_DC_START_ADD6	RX FIFO Queue 6 Data Container Start Address	read-write	0x0
0x4C0	RX_FQ_RD_ADD_PT6	RX FIFO Queue 6 Read Address Pointer	read-write	0x0
0x4C8	RX_FQ_ADD_PT7	RX FIFO Queue 7 Current Address Pointer	read-only	0x0
0x4CC	RX_FQ_START_ADD7	RX FIFO Queue 7 link list Start Address	read-write	0x0
0x4D0	RX_FQ_SIZE7	RX FIFO Queue 7 link list and data container Size	read-write	0x0
0x4D4	RX_FQ_DC_START_ADD7	RX FIFO Queue 7 Data Container Start Address	read-write	0x0
0x4D8	RX_FQ_RD_ADD_PT7	RX FIFO Queue 7 Read Address Pointer	read-write	0x0
TX filter control registers				
0x600	TX_FILTER_CTRL0	TX Filter Control register 0	read-write	0x0
0x604	TX_FILTER_CTRL1	TX Filter Control register 1	read-write	0x0
0x608	TX_FILTER_REFVAL0	TX Filter Reference Value register 0	read-write	0x0
0x60C	TX_FILTER_REFVAL1	TX Filter Reference Value register 1	read-write	0x0
0x610	TX_FILTER_REFVAL2	TX Filter Reference Value register 2	read-write	0x0
0x614	TX_FILTER_REFVAL3	TX Filter Reference Value register 3	read-write	0x0
RX filter control registers				
0x680	RX_FILTER_CTRL	RX Filter Control register	read-write	0x0
Interrupts control and status registers				
0x700	TX_FQ_INT_STS	TX FIFO Queue Interrupt Status register	read-write	0x0
0x704	RX_FQ_INT_STS	RX FIFO Queue Interrupt Status register	read-write	0x0
0x708	TX_PQ_INT_STS0	TX Priority Queue Interrupt Status register 0	read-write	0x0
0x70C	TX_PQ_INT_STS1	TX Priority Queue Interrupt Status register 1	read-write	0x0

0x710	STATS_INT_STS	Statistics Interrupt Status register	read-write	0x0
0x714	ERR_INT_STS	Error Interrupt Status register	read-write	0x0
0x718	SFTY_INT_STS	Safety Interrupt Status register	read-write	0x0
0x71C	AXI_ERR_INFO	AXI Error Information	read-only	0x0
0x720	DESC_ERR_INFO0	Descriptor Error Information 0	read-only	0x0
0x724	DESC_ERR_INFO1	Descriptor Error Information 1	read-only	0x0
0x728	TX_FILTER_ERR_INFO	TX Filter Error Information	read-only	0x0
Integration/Debug control and status registers				
0x800	DEBUG_TEST_CTRL	Debug Control register	read-write	0x0
0x804	INT_TEST0	Interrupt Test register 0	read-write	0x0
0x808	INT_TEST1	Interrupt Test register 1	read-write	0x0
0x810	TX_SCAN_FC	TX-SCAN first candidates register	read-only	0x0
0x814	TX_SCAN_BC	TX-SCAN best candidates register	read-only	0x0
0x818	TX_FQ_DESC_VALID	Valid TX FIFO Queue descriptors in local memory	read-only	0x0
0x81C	TX_PQ_DESC_VALID	Valid TX Priority Queue descriptors in local memory	read-only	0x0
CRC control registers				
0x880	CRC_CTRL	CRC Control register	write-only	0x0
0x884	CRC_REG	CRC register	read-write	0x0

1.4.4.2 Register Description

1.4.4.2.1 xcand_mh_creg

REGISTER DESCRIPTION: Global MH control and status registers
SIZE:
Register Base Address: 0x000
Register Address Range: 0x900

1.4.4.2.1.1 VERSION

Release Identification Register
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000000																Initial Value:								0x05600000							
	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
Bit	REL				STEP				SUBSTEP				YEAR				MON								DAY							
Mode	R				R				R				R				R								R							
Initial Value	0x0				0x5				0x6				0x0				0x0								0x0							

Bit 7:0 Define the day of the release using a binary coded decimal representation (1 being the first day of the month and so forth). This reset value is defined by the generic parameter DESIGN_TIME_STAMP_G[7:0]. If the generic parameter DESIGN_TIME_STAMP_G is not set, the default value is the one defined here

Bit 15:8 Define the month of the release using a binary coded decimal representation (1 being January and so forth). This reset value is defined by the generic parameter DESIGN_TIME_STAMP_G[15:8]. If the generic parameter DESIGN_TIME_STAMP_G is not set, the default value is the one defined here

Bit 19:16 Define the year of the release using a binary coded decimal representation (0 being 2020 and so forth...). This reset value is defined by the generic parameter DESIGN_TIME_STAMP_G[19:16]. If the generic parameter DESIGN_TIME_STAMP_G is not set, the default value is the one defined here

Bit 23:20 Sub-Step value according to Step

Bit 27:24 Step value according to Release

Bit 31:28 Release value, used to identify the main release of the X_CAN

1.4.4.2.1.2 MH_CTRL

Message Handler Control register

Address Offset:	0x00000004																Initial Value:															
	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
Bit																																START
Mode																																RW
Initial Value																																0x0

Bit 0 Before starting any RX/TX FIFO Queues or TX FIFO Queue slots the MH must write 1 to this bit prior launching the PRT. At initial start, as long as the PRT is not started, this bit can be set back to 0. When set to 1, the global configuration registers are write-protected. As soon as the PRT is started, this bit cannot be set to 0.

This bit can only be set to back to 0 if MH_STS.ENABLE = 0 and MH_STS.BUSY =0. For more details on starting/stopping or restarting the MH, refer to the Programming Guidelines chapter.

1.4.4.2.1.3 MH_CFG

Message Handler Configuration register

This register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000008																Initial Value:															
	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
Bit														INST_NUM								MAX_RETRANS										RX_CONT_DC
Mode														RW								RW										RW
Initial Value														0x0								0x7										0x0

Bit 0 When set to 1, the Continuous mode is active. This mode provides the option to have a linear and continuous memory organization of the RX message data. Only one RX descriptor is used by RX message data and

one single data container is required. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 10:8 Maximum number of TX message re-transmissions. Different configurations are possible: 0 -> no re-transmission; 1 to 6 -> 1 to 6 re-transmissions; 7-> unlimited re-transmissions; This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 18:16 In case that a cluster of X_CAN is defined, this bit field is used as a unique identifier per instance. This identifier is used by the MH to determine if the TX/RX descriptors are fetched by the right instance, see RX/TX description. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

1.4.4.2.1.4 MH_STS

Message Handler Status register

Address Offset:	0x0000000c																Initial Value:															
	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
Bit																								CLOCK ACTIVE				ENABLE				BUSY
Mode																								R				R				R
Initial Value																								0x0				0x0				0x0

Bit 0 This bit is the general busy flag, it is an ORED(RX/TX FIFO Queues and TX Priority Queue slots busy flags)

Bit 4 Value of the ENABLE signal driven by the PRT. The PRT signalizes via ENABLE whether it is active (ENABLE = 1) and requires message handling or not (ENABLE = 0).

Bit 8 Status of MH core clock: 0 = clock off, 1 = clock on.

1.4.4.2.1.5 MH_SFTY_CFG

Message Handler Safety Configuration register

This register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000010																Initial Value:								0x00000000							
	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
Bit	PRESCALER		PRT_TO_VAL														MEM_TO_VAL								DMA_TO_VAL							
Mode	RW		RW														RW								RW							
Initial Value	0x0		0x0														0x0								0x0							

Bit 7:0 This value is used by the watchdog timer for the DMA_AXI interface and defines the maximum number of timer ticks until a read or write access has to be completed. This value must be configured according to the maximum system latency, expected on the DMA_AXI interface. If this value is set to 0 and MH_SFTY_CTRL.DMA_TO_EN = 1 then the DMA_TO_ERR interrupt is triggered right away when accessing the S_MEM. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 15:8 This value is used by the watchdog timer for the MEM_AXI interface and defines the maximum number of timer ticks until a read or write access has to be completed. This value must be configured to the expected maximum latency on the MEM_AXI interface. If this value is set to 0 and MH_SFTY_CTRL.MEM_TO_EN = 1 then the MEM_TO_ERR interrupt is triggered right away when accessing the L_MEM. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 29:16 This value is used by the watchdog timers for the internal RX_MSG and TX_MSG interfaces. It defines the maximum number of timer ticks until a message has to be transferred from PRT to MH respective MH to PRT. The value must be configured according to the CAN frame which requires the longest time to be transported on the CAN bus. If this value is set to 0 and MH_SFTY_CTRL.PRT_TO_EN = 1 then the DP_TO_ERR interrupt is triggered right away at the beginning of a RX message or when starting a TX message. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 31:30 Prescaler used to generate the timer ticks for the watchdogs. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0. According to the value a different clock ratio can be selected:

0: clk divided by 32
1: clk divided by 64
2: clk divided by 128
3: clk divided by 512

1.4.4.2.1.6 MH_SFTY_CTRL

Message Handler Safety Control register
This register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000014																Initial Value:											0x00000000	
	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
Bit																						PRT_TO_EN	MEM_TO_EN	DMA_TO_EN	DMA_CH_CHK_EN	RX_AP_PARITY_EN	TX_AP_PARITY_EN	TX_DP_PARITY_EN	RX_DP_PARITY_EN	MEM_PROT_EN	RX_DESC_CRC_EN	TX_DESC_CRC_EN
Mode																						RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Initial Value																						0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

Bit 0 When set to 1, the CRC check for the TX descriptors is enabled. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 1 When set to 1, the CRC check for the RX descriptors is enabled. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 2 When set to 1, the sfty_err signal from the local memory interface is checked. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 3 When set to 1, the data path parity check performed on the RX path is enabled. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 4 When set to 1, the data path parity check performed on the TX path is enabled. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 5 When set to 1, the address pointer parity check on the TX path is enabled. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 6 When set to 1, the address pointer parity check on the RX path is enabled. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 7 When set to 1, the read/write DMA channels routing is checked. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 8 When set to 1, the watchdog for the DMA_AXI interface is enabled, otherwise disabled. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 9 When set to 1, the watchdog for the MEM_AXI interface is enabled, otherwise disabled. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 10 When set to 1, the watchdogs for the internal RX_MSG and TX_MSG interfaces are enabled, otherwise disabled. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

1.4.4.2.1.7 RX_FILTER_MEM_ADD

RX Filter Base Address register

This register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000018																Initial Value: 0x00000000															
	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
Bit																	BASE_ADDR															
Mode																	RW															
Initial Value																	0x0															

Bit 15:0 Define the base address where the RX filter elements are defined in L_MEM (up to 64Kbytes can be addressed). The BASE_ADDR[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

1.4.4.2.1.8 TX_DESC_MEM_ADD

TX Descriptor Base Address register

This register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x0000001c																Initial Value:	0x00000000													
	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
Bit	PQ_BASE_ADDR																FQ_BASE_ADDR															
Mode	RW																RW															
Initial Value	0x0																0x0															

Bit 15:0 Define the base address where the TX FIFO Queue descriptors are stored in L_MEM (up to 64Kbytes can be addressed). The FQ_BASE_ADDR[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 31:16 Define the base address where the TX Priority Queue descriptors are stored in L_MEM (up to 64Kbytes can be addressed). The PQ_BASE_ADDR[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

1.4.4.2.1.9 AXI_ADD_EXT

AXI address extension register

This register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000020																Initial Value:															
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the MSB of the read/write AXI address bus used on the DMA_AXI interface. If not required, leave the default value and do not connect the upper part of the DMA_AXI read/write address bus. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

1.4.4.2.1.10AXI_PARAMS

AXI parameter register
This register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000024																Initial Value:															
	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
Bit																											AW_MAX_PEND				AR_MAX_PEND	
Mode																											RW				RW	
Initial Value																											0x0				0x0	

Bit 1:0 AR_MAX_PEND[1:0] defines the maximum read pending transactions on DMA_AXI interface: 0 -> no read transfer; 1 -> 1 outstanding read transaction; 2 -> 2 outstanding read transactions, 3 -> 3 outstanding read transactions. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

Bit 5:4 AW_MAX_PEND[1:0] defines the maximum write pending transactions on DMA_AXI interface: 0 -> no write transfer; 1 -> 1 outstanding write transaction allowed; 2 -> 2 outstanding write transactions, 3 -> 3 outstanding write transactions. This bit field register is only accessible in write mode if the MH is not started, see MH_CTRL.START = 0.

1.4.4.2.1.11MH_LOCK

Message Handler Lock register

Address Offset:	0x00000028																Initial Value:															
	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
Bit	TMK																ULK															
Mode	RW																RW															
Initial Value	0x0																0x0															

Bit 15:0 Unlock key register. Two consecutive writes to this bit field, starting with 0x1234 and 0x04321, must be done before writing to a register being locked.

Bit 31:16 Test mode key register. Two consecutive writes to this bit field, starting with 0x6789 and 0x9876, must be done before writing to the DEBUG_TEST_CTRL register.

1.4.4.2.1.12TX_DESC_ADD_PT

TX descriptor current address pointer register

Address Offset:	0x00000100																Initial Value:															
	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
Bit	VAL																															
Mode	R																															
Initial Value	0x0																															

Bit 31:0 Address used to fetch a TX descriptor for the TX FIFO Queues or TX Priority Queue slots. It could be for several reasons: a new message needs to be fetched from a TX FIFO Queue or a new message is defined in a TX Priority Queue slot. This address value is always word aligned (32bit).

1.4.4.2.1.13 TX_STATISTICS

TX Message Counter register

Address Offset:	0x00000104																Initial Value:															
	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
Bit					UNSUCC																SUCC											
Mode					RW																RW											
Initial Value					0x0																0x0											

Bit 11:0 Counter incremented with every successful transmission of a CAN message to the CAN bus. The counter wraps automatically to 0 and can be cleared when writing 0 to the bit field. A STATS_IRQ interrupt is generated when the counter wraps.

Bit 27:16 Counter incremented with every unsuccessful transmission of a CAN message to the CAN bus. The counter wraps automatically to 0 and can be cleared when writing 0 to the bit field. A STATS_IRQ interrupt is generated when the counter wraps.

1.4.4.2.1.14TX_FQ_STS0

TX FIFO Queue Status register

Address Offset:	0x00000108																Initial Value:								0x00000000							
	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
Bit									STOP																BUSY							
Mode									R																R							
Initial Value									0x0																0x0							

Bit 7:0 When BUSY[n] = 1 the TX FIFO Queue n is active, this means the FIFO Queue is started and running (TX message defined in the TX FIFO Queue n can be processed). When the BUSY[n] = 0, the TX FIFO Queue n is stopped and would require a write to the TX_FQ_CTRL0.START[n] to make it active again. A TX FIFO Queue can go inactive if the END bit in the last TX descriptor of a TX message is set. In this case the, the BUSY[n] = 0 can occur only if the TX header descriptor of this last message has been acknowledged for the TX FIFO Queue n. When the TX FIFO Queue n is aborted, the BUSY[n] flag is set to 0 only when no acknowledge is pending.

Bit 23:16 When STOP[n] = 1 the TX FIFO Queue n is on hold, this means the FIFO Queue is started and running but waits for the SW to keep going. The STOP[n] can be set only if the BUSY[n] = 1. Several root causes may lead to this state: an error is detected, or a TX descriptor is not valid. To identify the potential issues, refer to the TX_FQ_STS1 register. In order to keep going with the TX FIFO Queue n, a write to the TX_FQ_CTRL0.START[n] is required. When BUSY[n] = 0, this bit is automatically set to 0

1.4.4.2.1.15TX_FQ_STS1

TX FIFO Queue Status register

Address Offset:	0x0000010c																Initial Value:								0x00000000							
	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
Bit									ERROR																UNVALID							
Mode									R																R							
Initial Value									0x0																0x0							

Bit 7:0 When UNVALID[n] = 1 the TX FIFO Queue n is on hold due to an TX descriptor with VALID=0 was loaded.

Bit 23:16 When ERROR[n] = 1 the TX FIFO Queue n is on hold due to an inconsistent TX descriptor was loaded, see chapter Descriptor Protection.

1.4.4.2.1.16 TX_FQ_CTRL0

TX FIFO Queue Control register 0

Address Offset:	0x00000110																Initial Value:								0x00000000							
	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
Bit																									START							
Mode																									RW							
Initial Value																									0x0							

Bit 7:0 When writing a 1 to the START[n], the TX FIFO Queue n is started. This bit is autocleared. Once started, the TX_FQ_STS0.BUSY[n] is set to 1. The MH must be started prior to any TX FIFO Queue start (MH_STS.BUSY set to 1). A TX FIFO Queue n can only be started if TX_FQ_CTRL2.ENABLE[n] is set to 1 and in order to avoid a dead lock situation with the PRT, the ENABLE signal from the PRT is high.

1.4.4.2.1.17TX_FQ_CTRL1

TX FIFO Queue Control register 1

This register is only accessible in write mode if the unlock key sequence has been performed prior to write

Address Offset:	0x00000114																Initial Value:								0x00000000							
	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
Bit																									ABORT							
Mode																									RW							
Initial Value																									0x0							

Bit 7:0 When ABORT[n] is set to 1, the TX FIFO Queue n is aborted. Once set to 1, the MH will abort all pending transaction related to the TX FIFO Queue n whenever required. This bit must be set back to 0 only when the TX FIFO Queue n is inactive, TX_FQ_STS0.BUSY[n] = 0. This bit field register is only accessible in write mode if the unlock key sequence has been performed prior to write.

1.4.4.2.1.18TX_FQ_CTRL2

TX FIFO Queue Control register 2

Address Offset:	0x00000118																Initial Value:								0x00000000							
	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
Bit																									ENABLE							
Mode																									RW							
Initial Value																									0x0							

Bit 7:0 When ENABLE[n] is set to 1, the TX FIFO Queue n is enabled. A TX FIFO Queue cannot be started if it is not enabled. Aborting a not started TX FIFO Queue has no effect.

1.4.4.2.1.19 TX_FQ_ADD_PT0

TX FIFO Queue 0 Current Address Pointer register

Address Offset:	0x00000120																Initial Value:																0x00000000
	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
Bit	VAL																															
Mode	R																															
Initial Value	0x0																															

Bit 31:0 Provide the header descriptor address of the TX message being in used by the arbiter for the TX FIFO Queue. To follow TX descriptors over time while running TX FIFO Queues, refer to the TX_DESC_ADD_PT register. This address value is always word aligned (32bit).

1.4.4.2.1.20 TX_FQ_START_ADD0

TX FIFO Queue 0 Start Address register

This register is only accessible in write mode if the TX FIFO Queue 0 is not busy, see BUSY flag in TX_FQ_STS0 register.

This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000124																Initial Value:																0x00000000
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the start address of the TX FIFO Queue link list descriptor in the system memory. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word

aligned (32bit). This bit field register is only accessible in write mode if the TX FIFO Queue 0 is not busy, see BUSY flag in TX_FQ_STS0 register.

1.4.4.2.1.21 TX_FQ_SIZE0

TX FIFO Queue 0 Size register

This register is only accessible in write mode if the TX FIFO Queue 0 is not busy, see BUSY flag in TX_FQ_STS0 register.

This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000128																Initial Value:										0x00000000										
	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				
Bit																							MAX_DESC														
Mode																							RW														
Initial Value																							0x0														

Bit 9:0 Define the maximum number of TX descriptors in the TX FIFO Queue link list descriptors. It is important to note that MAX_DESC = 0 does not prevent the TX FIFO Queue to be enabled and started. An active and running TX FIFO Queue with MAX_DESC = 0 is not allowed and will result in a DESC_ERR interrupt if no TX descriptor is defined. The memory size to allocate is MAX_DESC * 32bytes for MAX_DESC > = 1. This bit field register is only accessible in write mode if the TX FIFO Queue 0 is not busy, see BUSY flag in TX_FQ_STS0 register.

1.4.4.2.1.22 TX_FQ_ADD_PT1

TX FIFO Queue 1 Current Address Pointer register

Address Offset:	0x00000130	Initial Value: 0x00000000
	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
Bit		VAL
Mode		R
Initial Value		0x0

Bit 31:0 Provide the header descriptor address of the TX message being in used by the arbiter for the TX FIFO Queue. To follow TX descriptors over time while running TX FIFO Queues, refer to the TX_DESC_ADD_PT register. This address value is always word aligned (32bit).

1.4.4.2.1.23TX_FQ_START_ADD1

TX FIFO Queue 1 Start Address register

This register is only accessible in write mode if the TX FIFO Queue 1 is not busy, see BUSY flag in TX_FQ_STS0 register.

This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000134	Initial Value: 0x00000000
	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
Bit		VAL
Mode		RW
Initial Value		0x0

Bit 31:0 Define the start address of the TX FIFO Queue link list descriptor in the system memory. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This bit field register is only accessible in write mode if the TX FIFO Queue 1 is not busy, see BUSY flag in TX_FQ_STS0 register.

1.4.4.2.1.24TX_FQ_SIZE1

TX FIFO Queue 1 Size register
This register is only accessible in write mode if the TX FIFO Queue 1 is not busy, see BUSY flag in TX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000138																Initial Value:										0x00000000									
	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
Bit																							MAX_DESC										
Mode																							RW										
Initial Value																							0x0										

Bit 9:0 Define the maximum number of TX descriptors in the TX FIFO Queue link list descriptors. It is important to note that MAX_DESC = 0 does not prevent the TX FIFO Queue to be enabled and started. An active and running TX FIFO Queue with MAX_DESC = 0 is not allowed and will result in a DESC_ERR interrupt if no TX descriptor is defined. The memory size to allocate is MAX_DESC * 32bytes for MAX_DESC > = 1. This bit field register is only accessible in write mode if the TX FIFO Queue 1 is not busy, see BUSY flag in TX_FQ_STS0 register.

1.4.4.2.1.25TX_FQ_ADD_PT2

TX FIFO Queue 2 Current Address Pointer register

Address Offset:	0x00000140	Initial Value: 0x00000000
	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
Bit		VAL
Mode		R
Initial Value		0x0

Bit 31:0 Provide the header descriptor address of the TX message being in used by the arbiter for the TX FIFO Queue. To follow TX descriptors over time while running TX FIFO Queues, refer to the TX_DESC_ADD_PT register. This address value is always word aligned (32bit).

1.4.4.2.1.26 TX_FQ_START_ADD2

TX FIFO Queue 2 Start Address register
This register is only accessible in write mode if the TX FIFO Queue 2 is not busy, see BUSY flag in TX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000144	Initial Value: 0x00000000
	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
Bit		VAL
Mode		RW
Initial Value		0x0

Bit 31:0 Define the start address of the TX FIFO Queue link list descriptor in the system memory. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This bit field register is only accessible in write mode if the TX FIFO Queue 2 is not busy, see BUSY flag in TX_FQ_STS0 register.

1.4.4.2.1.27TX_FQ_SIZE2

TX FIFO Queue 2 Size register
This register is only accessible in write mode if the TX FIFO Queue 2 is not busy, see BUSY flag in TX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000148																Initial Value:										0x00000000									
	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
Bit																							MAX_DESC										
Mode																							RW										
Initial Value																							0x0										

Bit 9:0 Define the maximum number of TX descriptors in the TX FIFO Queue link list descriptors. It is important to note that MAX_DESC = 0 does not prevent the TX FIFO Queue to be enabled and started. An active and running TX FIFO Queue with MAX_DESC = 0 is not allowed and will result in a DESC_ERR interrupt if no TX descriptor is defined. The memory size to allocate is MAX_DESC * 32bytes for MAX_DESC > = 1. This bit field register is only accessible in write mode if the TX FIFO Queue 2 is not busy, see BUSY flag in TX_FQ_STS0 register.

1.4.4.2.1.28TX_FQ_ADD_PT3

TX FIFO Queue 3 Current Address Pointer register

Address Offset:	0x00000150	Initial Value: 0x00000000
	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
Bit		VAL
Mode		R
Initial Value		0x0

Bit 31:0 Provide the header descriptor address of the TX message being in used by the arbiter for the TX FIFO Queue. To follow TX descriptors over time while running TX FIFO Queues, refer to the TX_DESC_ADD_PT register. This address value is always word aligned (32bit).

1.4.4.2.1.29 TX_FQ_START_ADD3

TX FIFO Queue 3 Start Address register
This register is only accessible in write mode if the TX FIFO Queue 3 is not busy, see BUSY flag in TX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000154	Initial Value: 0x00000000
	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
Bit		VAL
Mode		RW
Initial Value		0x0

Bit 31:0 Define the start address of the TX FIFO Queue link list descriptor in the system memory. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This bit field register is only accessible in write mode if the TX FIFO Queue 3 is not busy, see BUSY flag in TX_FQ_STS0 register.

1.4.4.2.1.30 TX_FQ_SIZE3

TX FIFO Queue 3 Size register
This register is only accessible in write mode if the TX FIFO Queue 3 is not busy, see BUSY flag in TX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000158																Initial Value:										0x00000000									
	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			
Bit																							MAX_DESC												
Mode																							RW												
Initial Value																							0x0												

Bit 9:0 Define the maximum number of TX descriptors in the TX FIFO Queue link list descriptors. It is important to note that MAX_DESC = 0 does not prevent the TX FIFO Queue to be enabled and started. An active and running TX FIFO Queue with MAX_DESC = 0 is not allowed and will result in a DESC_ERR interrupt if no TX descriptor is defined. The memory size to allocate is MAX_DESC * 32bytes for MAX_DESC > = 1. This bit field register is only accessible in write mode if the TX FIFO Queue 3 is not busy, see BUSY flag in TX_FQ_STS0 register.

1.4.4.2.1.31 TX_FQ_ADD_PT4

TX FIFO Queue 4 Current Address Pointer register

Address Offset:	0x00000160	Initial Value: 0x00000000
	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
Bit		VAL
Mode		R
Initial Value		0x0

Bit 31:0 Provide the header descriptor address of the TX message being in used by the arbiter for the TX FIFO Queue. To follow TX descriptors over time while running TX FIFO Queues, refer to the TX_DESC_ADD_PT register. This address value is always word aligned (32bit).

1.4.4.2.1.32 TX_FQ_START_ADD4

TX FIFO Queue 4 Start Address register
This register is only accessible in write mode if the TX FIFO Queue 4 is not busy, see BUSY flag in TX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000164	Initial Value: 0x00000000
	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
Bit		VAL
Mode		RW
Initial Value		0x0

Bit 31:0 Define the start address of the TX FIFO Queue link list descriptor in the system memory. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This bit field register is only accessible in write mode if the TX FIFO Queue 4 is not busy, see BUSY flag in TX_FQ_STS0 register.

1.4.4.2.1.33TX_FQ_SIZE4

TX FIFO Queue 4 Size register
This register is only accessible in write mode if the TX FIFO Queue 4 is not busy, see BUSY flag in TX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000168																Initial Value:										0x00000000									
	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			
Bit																							MAX_DESC												
Mode																							RW												
Initial Value																							0x0												

Bit 9:0 Define the maximum number of TX descriptors in the TX FIFO Queue link list descriptors. It is important to note that MAX_DESC = 0 does not prevent the TX FIFO Queue to be enabled and started. An active and running TX FIFO Queue with MAX_DESC = 0 is not allowed and will result in a DESC_ERR interrupt if no TX descriptor is defined. The memory size to allocate is MAX_DESC * 32bytes for MAX_DESC > = 1. This bit field register is only accessible in write mode if the TX FIFO Queue 4 is not busy, see BUSY flag in TX_FQ_STS0 register.

1.4.4.2.1.34TX_FQ_ADD_PT5

TX FIFO Queue 5 Current Address Pointer register

Address Offset:	0x00000170	Initial Value: 0x00000000
	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
Bit		VAL
Mode		R
Initial Value		0x0

Bit 31:0 Provide the header descriptor address of the TX message being in used by the arbiter for the TX FIFO Queue. To follow TX descriptors over time while running TX FIFO Queues, refer to the TX_DESC_ADD_PT register. This address value is always word aligned (32bit).

1.4.4.2.1.35 TX_FQ_START_ADD5

TX FIFO Queue 5 Start Address register
This register is only accessible in write mode if the TX FIFO Queue 5 is not busy, see BUSY flag in TX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000174	Initial Value: 0x00000000
	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
Bit		VAL
Mode		RW
Initial Value		0x0

Bit 31:0 Define the start address of the TX FIFO Queue link list descriptor in the system memory. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This bit field register is only accessible in write mode if the TX FIFO Queue 5 is not busy, see BUSY flag in TX_FQ_STS0 register.

1.4.4.2.1.36TX_FQ_SIZE5

TX FIFO Queue 5 Size register
This register is only accessible in write mode if the TX FIFO Queue 5 is not busy, see BUSY flag in TX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000178																Initial Value:										0x00000000									
	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
Bit																							MAX_DESC										
Mode																							RW										
Initial Value																							0x0										

Bit 9:0 Define the maximum number of TX descriptors in the TX FIFO Queue link list descriptors. It is important to note that MAX_DESC = 0 does not prevent the TX FIFO Queue to be enabled and started. An active and running TX FIFO Queue with MAX_DESC = 0 is not allowed and will result in a DESC_ERR interrupt if no TX descriptor is defined. The memory size to allocate is MAX_DESC * 32bytes for MAX_DESC > = 1. This bit field register is only accessible in write mode if the TX FIFO Queue 5 is not busy, see BUSY flag in TX_FQ_STS0 register.

1.4.4.2.1.37TX_FQ_ADD_PT6

TX FIFO Queue 6 Current Address Pointer register

Address Offset:	0x00000180	Initial Value: 0x00000000
	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
Bit		VAL
Mode		R
Initial Value		0x0

Bit 31:0 Provide the header descriptor address of the TX message being in used by the arbiter for the TX FIFO Queue. To follow TX descriptors over time while running TX FIFO Queues, refer to the TX_DESC_ADD_PT register. This address value is always word aligned (32bit).

1.4.4.2.1.38 TX_FQ_START_ADD6

TX FIFO Queue 6 Start Address register
This register is only accessible in write mode if the TX FIFO Queue 6 is not busy, see BUSY flag in TX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000184	Initial Value: 0x00000000
	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
Bit		VAL
Mode		RW
Initial Value		0x0

Bit 31:0 Define the start address of the TX FIFO Queue link list descriptor in the system memory. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This bit field register is only accessible in write mode if the TX FIFO Queue 6 is not busy, see BUSY flag in TX_FQ_STS0 register.

1.4.4.2.1.39TX_FQ_SIZE6

TX FIFO Queue 6 Size register
This register is only accessible in write mode if the TX FIFO Queue 6 is not busy, see BUSY flag in TX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000188																Initial Value:										0x00000000									
	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
Bit																							MAX_DESC										
Mode																							RW										
Initial Value																							0x0										

Bit 9:0 Define the maximum number of TX descriptors in the TX FIFO Queue link list descriptors. It is important to note that MAX_DESC = 0 does not prevent the TX FIFO Queue to be enabled and started. An active and running TX FIFO Queue with MAX_DESC = 0 is not allowed and will result in a DESC_ERR interrupt if no TX descriptor is defined. The memory size to allocate is MAX_DESC * 32bytes for MAX_DESC > = 1. This bit field register is only accessible in write mode if the TX FIFO Queue 6 is not busy, see BUSY flag in TX_FQ_STS0 register.

1.4.4.2.1.40TX_FQ_ADD_PT7

TX FIFO Queue 7 Current Address Pointer register

Address Offset:	0x00000190	Initial Value: 0x00000000
	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
Bit		VAL
Mode		R
Initial Value		0x0

Bit 31:0 Provide the header descriptor address of the TX message being in used by the arbiter for the TX FIFO Queue. To follow TX descriptors over time while running TX FIFO Queues, refer to the TX_DESC_ADD_PT register. This address value is always word aligned (32bit).

1.4.4.2.1.41 TX_FQ_START_ADD7

TX FIFO Queue 7 Start Address register
This register is only accessible in write mode if the TX FIFO Queue 7 is not busy, see BUSY flag in TX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000194	Initial Value: 0x00000000
	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
Bit		VAL
Mode		RW
Initial Value		0x0

Bit 31:0 Define the start address of the TX FIFO Queue link list descriptor in the system memory. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This bit field register is only accessible in write mode if the TX FIFO Queue 7 is not busy, see BUSY flag in TX_FQ_STS0 register.

1.4.4.2.1.42TX_FQ_SIZE7

TX FIFO Queue 7 Size register
This register is only accessible in write mode if the TX FIFO Queue 7 is not busy, see BUSY flag in TX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000198																Initial Value:																0x00000000										
	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										
Bit																							MAX_DESC																			
Mode																							RW																			
Initial Value																							0x0																			

Bit 9:0 Define the maximum number of TX descriptors in the TX FIFO Queue link list descriptors. It is important to note that MAX_DESC = 0 does not prevent the TX FIFO Queue to be enabled and started. An active and running TX FIFO Queue with MAX_DESC = 0 is not allowed and will result in a DESC_ERR interrupt if no TX descriptor is defined. The memory size to allocate is MAX_DESC * 32bytes for MAX_DESC > = 1. This bit field register is only accessible in write mode if the TX FIFO Queue 7 is not busy, see BUSY flag in TX_FQ_STS0 register.

1.4.4.2.1.43TX_PQ_STS0

TX Priority Queue Status register

Address Offset:	0x00000300																Initial Value:															0x00000000
	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
Bit	BUSY																															
Mode	R																															
Initial Value	0x0																															

Bit 31:0 When BUSY[n] = 1, the TX Priority Queue slot n is busy, which means that the TX descriptor in the slot n is being loaded in L_MEM and considered by the TX-Scan. As long as this bit remains high, the message attached to the slot n has not been sent yet. The BUSY[n] = 0 can occur only if the TX header descriptor of the slot n has been acknowledged.

1.4.4.2.1.44TX_PQ_STS1

TX Priority Queue Status register

Address Offset:	0x00000304																Initial Value:															0x00000000
	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
Bit	SENT																															
Mode	R																															
Initial Value	0x0																															

Bit 31:0 When SENT[n] = 1 the TX message assigned to the TX Priority Queue slot n has been transmitted and the TX descriptor attached to the slot n is acknowledged. This bit will be cleared once a new start on this slot will occur.

1.4.4.2.1.45TX_PQ_CTRL0

TX Priority Queue Control register 0

Address Offset:	0x0000030c																Initial Value:															
	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
Bit	START																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 When writing a 1 to the START[n], the TX Priority Queue slot n is started and running. This bit is autocleared and once started, the TX_PQ_STS0.BUSY[n] is set to 1. The MH must be started prior to any TX Priority Queue slot start (MH_STS.BUSY set to 1). A TX Priority Queue slot n can only be started if TX_PQ_CTRL2.ENABLE[n] is set to 1 and in order to avoid a dead lock situation with the PRT, the ENABLE signal from the PRT is high.

1.4.4.2.1.46 TX_PQ_CTRL1

TX Priority Queue Control register 1
This register is only accessible in write mode if the unlock key sequence has been performed prior to write

Address Offset:	0x00000310																Initial Value:															
	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
Bit	ABORT																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 When ABORT[n] is set to 1, the TX Priority Queue slot n is aborted. This bit must be set back to 0 only when the TX Priority Queue slot n is inactive, TX_FQ_STS0.BUSY[n] = 0. A TX message attached to a slot can only be aborted if it is not stored in the two internal buffers holding the

two best candidates for the next TX message. Despite a TX message is aborted, it may have been sent, check the TX_PQ_STS1.SENT[n] bit register for the slot n. This bit field register is only accessible in write mode if the unlock key sequence has been performed prior to write.

1.4.4.2.1.47 TX_PQ_CTRL2

TX Priority Queue Control register 2

Address Offset:	0x00000314																Initial Value:															0x00000000
	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
Bit	ENABLE																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 When ENABLE[n] is set to 1, the slot n in the TX Priority Queue is enabled. A TX Priority Queue slot cannot be started if not enabled. Aborting a not started slot n has no effect

1.4.4.2.1.48 TX_PQ_START_ADD

TX Priority Queue Start Address

This register is only accessible in write mode if the TX Priority Queue is not busy, see BUSY flag in TX_PQ_STS register. It means TX_PQ_STS register is equal to 0x0.

This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000318	Initial Value: 0x00000000
	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
Bit	VAL	
Mode	RW	
Initial Value	0x0	

Bit 31:0 Define the start address of the TX Priority Queue in the system memory. All TX header descriptors in the TX Priority Queue are continuously defined from this start address. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This bit field register is only accessible in write mode if the TX Priority Queue is not busy, see BUSY flag in TX_PQ_STS register

1.4.4.2.1.49RX_DESC_ADD_PT

RX descriptor Current Address Pointer

Address Offset:	0x00000400	Initial Value: 0x00000000
	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
Bit	VAL	
Mode	R	
Initial Value	0x0	

Bit 31:0 Provide the address used to fetch the current RX descriptor. This address value is always word aligned (32bit).

1.4.4.2.1.50RX_STATISTICS

RX Message Counter register

Address Offset:	0x00000404																Initial Value:															
	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
Bit					UNSUCC																SUCC											
Mode					RW																RW											
Initial Value					0x0																0x0											

Bit 11:0 Counter incremented with every successful reception of a CAN message from the CAN bus. The counter wraps automatically to 0 and can be cleared when writing 0x00 to the bit field. An interrupt is generated when the counter wraps.

Bit 27:16 Counter incremented with every unsuccessful reception of a CAN message from the CAN bus. The counter wraps automatically to 0 and can be cleared when writing 0x00 to the bit field. . An interrupt is generated when the counter wraps.

1.4.4.2.1.51RX_FQ_STS0

RX FIFO Queue Status register 0

Address Offset:	0x00000408																Initial Value:															
	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
Bit									STOP																BUSY							
Mode									R																R							
Initial Value									0x0																0x0							

Bit 7:0 When BUSY[n] = 1 the RX FIFO Queue n is busy, this means the FIFO Queue is started and running (RX message to be written to the RX FIFO Queue can be processed). When the BUSY[n] = 0, the RX FIFO Queue n is stopped and would require a write to the RX_FQ_CTRL0.START[n] to

make it active again. When the RX FIFO Queue n is aborted, the BUSY[n] flag is set to 0 only when no acknowledge is pending.

Bit 23:16 When STOP[n] = 1 the RX FIFO Queue n is on hold, it means started but waiting for the SW to react. The STOP[n] can be set only if the BUSY[n] = 1. Several root causes may lead to the RX FIFO Queue n to stop: an error is detected, or an RX descriptor is not valid, or the FIFO is full. To identify the potential issues, refer to the RX_FQ_STS1 and RX_FQ_STS2 registers. In order to keep going with the RX FIFO Queue n, a write to the RX_FQ_CTRL0.START[n] is required. When BUSY[n] = 0, this bit is automatically set to 0

1.4.4.2.1.52RX_FQ_STS1

RX FIFO Queue Status register 1

Address Offset:	0x0000040c																Initial Value:								0x00000000							
	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
Bit									ERROR																UNVALID							
Mode									R																R							
Initial Value									0x0																0x0							

Bit 7:0 When UNVALID[n] = 1 the RX FIFO Queue n is on hold due to an RX descriptor detected with VALID=0

Bit 23:16 When ERROR[n] = 1 the RX FIFO Queue n is on hold due to an inconsistent RX descriptor being loaded, see chapter Descriptor Protection.

1.4.4.2.1.53RX_FQ_STS2

RX FIFO Queue Status register 2

Address Offset:	0x00000410																Initial Value:								0x00000000							
	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
Bit																									DC_FULL							
Mode																									R							
Initial Value																									0x0							

Bit 7:0 When DC_FULL[n] = 1 the RX FIFO Queue n is stopped due to the RX FIFO Queue n being full. This register is relevant only for the Continuous Mode as in Normal mode, there is no need to provide such information to the MH

1.4.4.2.1.54RX_FQ_CTRL0

RX FIFO Queue Control register 0

Address Offset:	0x00000414																Initial Value:								0x00000000							
	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
Bit																									START							
Mode																									RW							
Initial Value																									0x0							

Bit 7:0 When writing a 1 to the START[n], the RX FIFO Queue n is started. This bit is autocleared and once started, the RX_FQ_STS0.BUSY[n] is set to 1. The MH must be started prior to any RX FIFO Queue start (MH_STS.BUSY set to 1). An RX FIFO Queue n can only be started if RX_FQ_CTRL2.ENABLE[n] is set to 1.

1.4.4.2.1.55RX_FQ_CTRL1

RX FIFO Queue Control register 1

This register is only accessible in write mode if the unlock key sequence has been performed prior to write

Address Offset:	0x00000418																Initial Value:								0x00000000							
	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
Bit																									ABORT							
Mode																									RW							
Initial Value																									0x0							

Bit 7:0 When ABORT[n] is set to 1, the RX FIFO Queue n is aborted. Once set to 1, the MH will abort all pending transactions related to the RX FIFO Queue n whenever required. The abort can be effective only if the RX FIFO Queue n is enabled. This bit must be set back to 0 only when the RX FIFO Queue n is inactive, RX_FQ_STS0.BUSY[n] = 0. This bit field register is only accessible in write mode if the unlock key sequence has been performed prior to write.

1.4.4.2.1.56RX_FQ_CTRL2

RX FIFO Queue Control register 2

Address Offset:	0x0000041c																Initial Value:								0x00000000							
	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
Bit																									ENABLE							
Mode																									RW							
Initial Value																									0x0							

Bit 7:0 When ENABLE[n] is set to 1, the RX FIFO Queue n is enabled. The RX FIFO Queue n cannot be started if not enabled. The abort of an RX FIFO Queue n not started would have no effect.

1.4.4.2.1.57RX_FQ_ADD_PT0

RX FIFO Queue 0 Current Address Pointer

Address Offset:	0x00000420	Initial Value:	0x00000000

	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
Bit	VAL																															
Mode	R																															
Initial Value	0x0																															

Bit 31:0 Provide the current RX Header Descriptor address pointer for the RX FIFO Queue 0 in the system memory. To follow RX descriptor over time, refer to the RX_DESC_ADD_PT register. This address value is always word aligned (32bit).

1.4.4.2.1.58RX_FQ_START_ADD0

RX FIFO Queue 0 link list Start Address

This register is only accessible in write mode if the RX FIFO Queue 0 is not busy, see BUSY flag in RX_FQ_STS0 register.

This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000424	Initial Value:	0x00000000

	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the start address of the RX FIFO Queue link list descriptor in system memory. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word

aligned (32bit). This register is only accessible in write mode if the RX FIFO Queue 0 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.59RX_FQ_SIZE0

RX FIFO Queue 0 link list and data container Size

This register is only accessible in write mode if the RX FIFO Queue 0 is not busy, see BUSY flag in RX_FQ_STS0 register.

This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000428																Initial Value:																0x00000000															
	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
Bit					DC_SIZE																		MAX_DESC									
Mode					RW																		RW									
Initial Value					0x0																		0x0									

Bit 9:0 Define the maximum number of descriptors in the RX FIFO Queue link list. It is important to note that MAX_DESC = 0 does not prevent the RX FIFO Queue to be enabled and started. An active and running RX FIFO Queue with MAX_DESC = 0 is not allowed and will result in a DESC_ERR interrupt if no RX descriptor is defined. The size to be allocated to the link list must be equal to MAX_DESC * 16bytes for MAX_DESC >= 1. This register is only accessible in write mode if the RX FIFO Queue 0 is not busy, see BUSY flag in RX_FQ_STS0 register

Bit 27:16 In Normal mode only the DC_SIZE[6:0] is used to define the maximum size of an RX data container for the RX FIFO Queue. The data container size is DC_SIZE[6:0] * 32bytes and one is attached to every RX descriptor. In continuous mode, it defines the size of the single data container used to write all RX messages. The overall data container size is DC_SIZE[11:0] * 32bytes for MAX_DESC > = 1. When set to 0, the RX FIFO Queue can be enabled but not started. This register is only accessible in write mode if the RX FIFO Queue 0 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.60RX_FQ_DC_START_ADD0

RX FIFO Queue 0 Data Container Start Address

This register is accessible in write mode if the RX FIFO Queue 0 is not busy, see BUSY flag in RX_FQ_STS0 register.

This register is protected by a register bank CRC defined in CRC_REG register. This register is used only in Continuous Mode

Address Offset:	0x0000042c																Initial Value:															
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the Data Container Start Address in system memory. This bit field is relevant only when the MH is configured in Continuous Mode. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This register is only accessible in write mode if the RX FIFO Queue 0 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.61RX_FQ_RD_ADD_PT0

RX FIFO Queue 0 Read Address Pointer

This register is used only in Continuous Mode.

Address Offset:	0x00000430																Initial Value:															
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 The SW uses this register to indicate the Data Read Address of the RX message being read to the MH. This address must point to the last word of the RX message considered in the data container. This bit field is relevant only when the MH is configured in Continuous mode. The MH

uses this information to ensure that enough memory space is available to write the next message. For an initial start, it is mandatory to set VAL[1:0] to 0b11, to avoid RX_FQ_RD_ADD_PT0 register to be equal to the RX_FQ_START_ADDR0 registers. Excepted for the initial value, the address value must always be word aligned (32bit), VAL[1:0] must be set to 0b00.

1.4.4.2.1.62RX_FQ_ADD_PT1

RX FIFO Queue 1 Current Address Pointer

Address Offset:	0x00000438	Initial Value:	0x00000000

	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
Bit	VAL																															
Mode	R																															
Initial Value	0x0																															

Bit 31:0 Provide the current RX Header Descriptor address pointer for the RX FIFO Queue 1 in the system memory. To follow RX descriptor over time, refer to the RX_DESC_ADD_PT register. This address value is always word aligned (32bit).

1.4.4.2.1.63RX_FQ_START_ADD1

RX FIFO Queue 1 link list Start Address
This register is only accessible in write mode if the RX FIFO Queue 1 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x0000043c																Initial Value:															
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the start address of the RX FIFO Queue link list descriptor in system memory. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This register is only accessible in write mode if the RX FIFO Queue 1 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.64RX_FQ_SIZE1

RX FIFO Queue 1 link list and data container Size
This register is only accessible in write mode if the RX FIFO Queue 1 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000440																Initial Value:															
	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
Bit					DC_SIZE																		MAX_DESC									
Mode					RW																		RW									
Initial Value					0x0																		0x0									

Bit 9:0 Define the maximum number of descriptors in the RX FIFO Queue link list. It is important to note that MAX_DESC = 0 does not prevent the RX FIFO Queue to be enabled and started. An active and running RX FIFO Queue with MAX_DESC = 0 is not allowed and will result in a DESC_ERR interrupt if no RX descriptor is defined. The size to be allocated to the

link list must be equal to MAX_DESC * 16bytes for MAX_DESC >= 1. This register is only accessible in write mode if the RX FIFO Queue 1 is not busy, see BUSY flag in RX_FQ_STS0 register

Bit 27:16 In Normal mode only the DC_SIZE[6:0] is used to define the maximum size of an RX data container for the RX FIFO Queue. The data container size is DC_SIZE[6:0] * 32bytes and one is attached to every RX descriptor. In continuous mode, it defines the size of the single data container used to write all RX messages. The overall data container size is DC_SIZE[11:0] * 32bytes for MAX_DESC > = 1. When set to 0, the RX FIFO Queue can be enabled but not started. This register is only accessible in write mode if the RX FIFO Queue 1 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.65RX_FQ_DC_START_ADD1

RX FIFO Queue 1 Data Container Start Address
This register is accessible in write mode if the RX FIFO Queue 1 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register. This register is used only in Continuous Mode

Address Offset:	0x00000444																Initial Value:																0x00000000
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the Data Container Start Address in system memory. This bit field is relevant only when the MH is configured in Continuous Mode. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This register is only accessible in write mode if the RX FIFO Queue 1 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.66RX_FQ_RD_ADD_PT1

RX FIFO Queue 1 Read Address Pointer
This register is used only in Continuous Mode.

Address Offset:	0x00000448																Initial Value:															
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 The SW uses this register to indicate the Data Read Address of the RX message being read to the MH. This address must point to the last word of the RX message considered in the data container. This bit field is relevant only when the MH is configured in Continuous mode. The MH uses this information to ensure enough memory space is available to write the next message. For an initial start, it is mandatory to set VAL[1:0] to 0b11, to avoid RX_FQ_RD_ADD_PT1 register to be equal to the RX_FQ_START_ADDR1 registers. Excepted for the initial value, the address value must always be word aligned (32bit), VAL[1:0] must be set to 0b00.

1.4.4.2.1.67RX_FQ_ADD_PT2

RX FIFO Queue 2 Current Address Pointer

Address Offset:	0x00000450																Initial Value:															
	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
Bit	VAL																															
Mode	R																															
Initial Value	0x0																															

Bit 31:0 Provide the current RX Header Descriptor address pointer for the RX FIFO Queue 2 in the system memory. To follow RX descriptor over time,

refer to the RX_DESC_ADD_PT register. This address value is always word aligned (32bit).

1.4.4.2.1.68RX_FQ_START_ADD2

RX FIFO Queue 2 link list Start Address
This register is only accessible in write mode if the RX FIFO Queue 2 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000454																Initial Value:																0x00000000
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the start address of the RX FIFO Queue link list descriptor in system memory. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This register is only accessible in write mode if the RX FIFO Queue 2 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.69RX_FQ_SIZE2

RX FIFO Queue 2 link list and data container Size
This register is only accessible in write mode if the RX FIFO Queue 2 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000458																Initial Value:										0x00000000										
	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
Bit					DC_SIZE																		MAX_DESC													
Mode					RW																		RW													
Initial Value					0x0																		0x0													

Bit 9:0 Define the maximum number of descriptors in the RX FIFO Queue link list. It is important to note that MAX_DESC = 0 does not prevent the RX FIFO Queue to be enabled and started. An active and running RX FIFO Queue with MAX_DESC = 0 is not allowed and will result in a DESC_ERR interrupt if no RX descriptor is defined. The size to be allocated to the link list must be equal to MAX_DESC * 16bytes for MAX_DESC >= 1. This register is only accessible in write mode if the RX FIFO Queue 2 is not busy, see BUSY flag in RX_FQ_STS0 register

Bit 27:16 In Normal mode only the DC_SIZE[6:0] is used to define the maximum size of an RX data container for the RX FIFO Queue. The data container size is DC_SIZE[6:0] * 32bytes and one is attached to every RX descriptor. In continuous mode, it defines the size of the single data container used to write all RX messages. The overall data container size is DC_SIZE[11:0] * 32bytes for MAX_DESC > = 1. When set to 0, the RX FIFO Queue can be enabled but not started. This register is only accessible in write mode if the RX FIFO Queue 2 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.70RX_FQ_DC_START_ADD2

RX FIFO Queue 2 Data Container Start Address
This register is accessible in write mode if the RX FIFO Queue 2 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register. This register is used only in Continuous Mode

Address Offset:	0x0000045c	Initial Value:	0x00000000

	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the Data Container Start Address in system memory. This bit field is relevant only when the MH is configured in Continuous Mode. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This register is only accessible in write mode if the RX FIFO Queue 2 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.71RX_FQ_RD_ADD_PT2

RX FIFO Queue 2 Read Address Pointer

This register is used only in Continuous Mode.

Address Offset:	0x00000460	Initial Value:	0x00000000

	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 The SW uses this register to indicate the Data Read Address of the RX message being read to the MH. This address must point to the last word of the RX message considered in the data container. This bit field is relevant only when the MH is configured in Continuous mode. The MH uses this information to ensure enough memory space is available to write the next message. For an initial start, it is mandatory to set

VAL[1:0] to 0b11, to avoid RX_FQ_RD_ADD_PT2 register to be equal to the RX_FQ_START_ADDR2 registers. Excepted for the initial value, the address value must always be word aligned (32bit), VAL[1:0] must be set to 0b00.

1.4.4.2.1.72RX_FQ_ADD_PT3

RX FIFO Queue 3 Current Address Pointer

Address Offset:	0x00000468																Initial Value:															0x00000000
	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
Bit	VAL																															
Mode	R																															
Initial Value	0x0																															

Bit 31:0 Provide the current RX Header Descriptor address pointer for the RX FIFO Queue 3 in the system memory. To follow RX descriptor over time, refer to the RX_DESC_ADD_PT register. This address value is always word aligned (32bit).

1.4.4.2.1.73RX_FQ_START_ADD3

RX FIFO Queue 3 link list Start Address
This register is only accessible in write mode if the RX FIFO Queue 3 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x0000046c																Initial Value:															
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the start address of the RX FIFO Queue link list descriptor in system memory. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This register is only accessible in write mode if the RX FIFO Queue 3 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.74RX_FQ_SIZE3

RX FIFO Queue 3 link list and data container Size
This register is only accessible in write mode if the RX FIFO Queue 3 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000470																Initial Value:															
	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
Bit					DC_SIZE																		MAX_DESC									
Mode					RW																		RW									
Initial Value					0x0																		0x0									

Bit 9:0 Define the maximum number of descriptors in the RX FIFO Queue link list. It is important to note that MAX_DESC = 0 does not prevent the RX FIFO Queue to be enabled and started. An active and running RX FIFO Queue with MAX_DESC = 0 is not allowed and will result in a DESC_ERR interrupt if no RX descriptor is defined. The size to be allocated to the

link list must be equal to MAX_DESC * 16bytes for MAX_DESC >= 1. This register is only accessible in write mode if the RX FIFO Queue 3 is not busy, see BUSY flag in RX_FQ_STS0 register

Bit 27:16 In Normal mode only the DC_SIZE[6:0] is used to define the maximum size of an RX data container for the RX FIFO Queue. The data container size is DC_SIZE[6:0] * 32bytes and one is attached to every RX descriptor. In continuous mode, it defines the size of the single data container used to write all RX messages. The overall data container size is DC_SIZE[11:0] * 32bytes for MAX_DESC > = 1. When set to 0, the RX FIFO Queue can be enabled but not started. This register is only accessible in write mode if the RX FIFO Queue 3 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.75RX_FQ_DC_START_ADD3

RX FIFO Queue 3 Data Container Start Address
This register is accessible in write mode if the RX FIFO Queue 3 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register. This register is used only in Continuous Mode

Address Offset:	0x00000474																Initial Value:																0x00000000
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the Data Container Start Address in system memory. This bit field is relevant only when the MH is configured in Continuous Mode. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This register is only accessible in write mode if the RX FIFO Queue 3 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.76RX_FQ_RD_ADD_PT3

RX FIFO Queue 3 Read Address Pointer
This register is used only in Continuous Mode.

Address Offset:	0x00000478																Initial Value:															
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 The SW uses this register to indicate the Data Read Address of the RX message being read to the MH. This address must point to the last word of the RX message considered in the data container. This bit field is relevant only when the MH is configured in Continuous mode. The MH uses this information to ensure enough memory space is available to write the next message. For an initial start, it is mandatory to set VAL[1:0] to 0b11, to avoid RX_FQ_RD_ADD_PT3 register to be equal to the RX_FQ_START_ADDR3 registers. Excepted for the initial value, the address value must always be word aligned (32bit), VAL[1:0] must be set to 0b00.

1.4.4.2.1.77RX_FQ_ADD_PT4

RX FIFO Queue 4 Current Address Pointer

Address Offset:	0x00000480																Initial Value:															
	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
Bit	VAL																															
Mode	R																															
Initial Value	0x0																															

Bit 31:0 Provide the current RX Header Descriptor address pointer for the RX FIFO Queue 4 in the system memory. To follow RX descriptor over time,

refer to the RX_DESC_ADD_PT register. This address value is always word aligned (32bit).

1.4.4.2.1.78RX_FQ_START_ADD4

RX FIFO Queue 4 link list Start Address
This register is only accessible in write mode if the RX FIFO Queue 4 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000484	Initial Value:	0x00000000

	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the start address of the RX FIFO Queue link list descriptor in system memory. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This register is only accessible in write mode if the RX FIFO Queue 4 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.79RX_FQ_SIZE4

RX FIFO Queue 4 link list and data container Size
This register is only accessible in write mode if the RX FIFO Queue 4 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000488																Initial Value:										0x00000000										
	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
Bit					DC_SIZE																		MAX_DESC													
Mode					RW																		RW													
Initial Value					0x0																		0x0													

Bit 9:0 Define the maximum number of descriptors in the RX FIFO Queue link list. It is important to note that MAX_DESC = 0 does not prevent the RX FIFO Queue to be enabled and started. An active and running RX FIFO Queue with MAX_DESC = 0 is not allowed and will result in a DESC_ERR interrupt if no RX descriptor is defined. The size to be allocated to the link list must be equal to MAX_DESC * 16bytes for MAX_DESC >= 1. This register is only accessible in write mode if the RX FIFO Queue 4 is not busy, see BUSY flag in RX_FQ_STS0 register

Bit 27:16 In Normal mode only the DC_SIZE[6:0] is used to define the maximum size of an RX data container for the RX FIFO Queue. The data container size is DC_SIZE[6:0] * 32bytes and one is attached to every RX descriptor. In continuous mode, it defines the size of the single data container used to write all RX messages. The overall data container size is DC_SIZE[11:0] * 32bytes for MAX_DESC > = 1. When set to 0, the RX FIFO Queue can be enabled but not started. This register is only accessible in write mode if the RX FIFO Queue 4 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.80RX_FQ_DC_START_ADD4

RX FIFO Queue 4 Data Container Start Address
This register is accessible in write mode if the RX FIFO Queue 4 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register. This register is used only in Continuous Mode

Address Offset:	0x0000048c	Initial Value:	0x00000000

	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the Data Container Start Address in system memory. This bit field is relevant only when the MH is configured in Continuous Mode. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This register is only accessible in write mode if the RX FIFO Queue 4 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.81RX_FQ_RD_ADD_PT4

RX FIFO Queue 4 Read Address Pointer

This register is used only in Continuous Mode.

Address Offset:	0x00000490	Initial Value:	0x00000000

	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 The SW uses this register to indicate the Data Read Address of the RX message being read to the MH. This address must point to the last word of the RX message considered in the data container. This bit field is relevant only when the MH is configured in Continuous mode. The MH uses this information to ensure enough memory space is available to write the next message. For an initial start, it is mandatory to set

VAL[1:0] to 0b11, to avoid RX_FQ_RD_ADD_PT4 register to be equal to the RX_FQ_START_ADDR4 registers. Excepted for the initial value, the address value must always be word aligned (32bit), VAL[1:0] must be set to 0b00.

1.4.4.2.1.82RX_FQ_ADD_PT5

RX FIFO Queue 5 Current Address Pointer

Address Offset:	0x00000498	Initial Value:	0x00000000

	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
Bit	VAL																															
Mode	R																															
Initial Value	0x0																															

Bit 31:0 Provide the current RX Header Descriptor address pointer for the RX FIFO Queue 5 in the system memory. To follow RX descriptor over time, refer to the RX_DESC_ADD_PT register. This address value is always word aligned (32bit).

1.4.4.2.1.83RX_FQ_START_ADD5

RX FIFO Queue 5 link list Start Address

This register is only accessible in write mode if the RX FIFO Queue 5 is not busy, see BUSY flag in RX_FQ_STS0 register.

This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x0000049c																Initial Value:															
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the start address of the RX FIFO Queue link list descriptor in system memory. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This register is only accessible in write mode if the RX FIFO Queue 5 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.84RX_FQ_SIZE5

RX FIFO Queue 5 link list and data container Size
This register is only accessible in write mode if the RX FIFO Queue 5 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x000004a0																Initial Value:															
	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
Bit					DC_SIZE																		MAX_DESC									
Mode					RW																		RW									
Initial Value					0x0																		0x0									

Bit 9:0 Define the maximum number of descriptors in the RX FIFO Queue link list. It is important to note that MAX_DESC = 0 does not prevent the RX FIFO Queue to be enabled and started. An active and running RX FIFO Queue with MAX_DESC = 0 is not allowed and will result in a DESC_ERR interrupt if no RX descriptor is defined. The size to be allocated to the

link list must be equal to MAX_DESC * 16bytes for MAX_DESC >= 1. This register is only accessible in write mode if the RX FIFO Queue 5 is not busy, see BUSY flag in RX_FQ_STS0 register

Bit 27:16 In Normal mode only the DC_SIZE[6:0] is used to define the maximum size of an RX data container for the RX FIFO Queue. The data container size is DC_SIZE[6:0] * 32bytes and one is attached to every RX descriptor. In continuous mode, it defines the size of the single data container used to write all RX messages. The overall data container size is DC_SIZE[11:0] * 32bytes for MAX_DESC > = 1. When set to 0, the RX FIFO Queue can be enabled but not started. This register is only accessible in write mode if the RX FIFO Queue 5 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.85RX_FQ_DC_START_ADD5

RX FIFO Queue 5 Data Container Start Address
This register is accessible in write mode if the RX FIFO Queue 5 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register. This register is used only in Continuous Mode

Address Offset:	0x000004a4																Initial Value:															
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the Data Container Start Address in system memory. This bit field is relevant only when the MH is configured in Continuous Mode. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This register is only accessible in write mode if the RX FIFO Queue 5 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.86RX_FQ_RD_ADD_PT5

RX FIFO Queue 5 Read Address Pointer
This register is used only in Continuous Mode.

Address Offset:	0x000004a8	Initial Value: 0x00000000
	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
Bit		VAL
Mode		RW
Initial Value		0x0

Bit 31:0 The SW uses this register to indicate the Data Read Address of the RX message being read to the MH. This address must point to the last word of the RX message considered in the data container. This bit field is relevant only when the MH is configured in Continuous mode. The MH uses this information to ensure enough memory space is available to write the next message. For an initial start, it is mandatory to set VAL[1:0] to 0b11, to avoid RX_FQ_RD_ADD_PT5 register to be equal to the RX_FQ_START_ADDR5 registers. Excepted for the initial value, the address value must always be word aligned (32bit), VAL[1:0] must be set to 0b00.

1.4.4.2.1.87RX_FQ_ADD_PT6

RX FIFO Queue 6 Current Address Pointer

Address Offset:	0x000004b0	Initial Value: 0x00000000
	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
Bit		VAL
Mode		R
Initial Value		0x0

Bit 31:0 Provide the current RX Header Descriptor address pointer for the RX FIFO Queue 6 in the system memory. To follow RX descriptor over time,

refer to the RX_DESC_ADD_PT register. This address value is always word aligned (32bit).

1.4.4.2.1.88RX_FQ_START_ADD6

RX FIFO Queue 6 link list Start Address
This register is only accessible in write mode if the RX FIFO Queue 6 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x000004b4	Initial Value:	0x00000000

	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the start address of the RX FIFO Queue link list descriptor in system memory. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This register is only accessible in write mode if the RX FIFO Queue 6 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.89RX_FQ_SIZE6

RX FIFO Queue 6 link list and data container Size
This register is only accessible in write mode if the RX FIFO Queue 6 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x000004b8																Initial Value:										0x00000000										
	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
Bit					DC_SIZE																		MAX_DESC													
Mode					RW																		RW													
Initial Value					0x0																		0x0													

Bit 9:0 Define the maximum number of descriptors in the RX FIFO Queue link list. It is important to note that MAX_DESC = 0 does not prevent the RX FIFO Queue to be enabled and started. An active and running RX FIFO Queue with MAX_DESC = 0 is not allowed and will result in a DESC_ERR interrupt if no RX descriptor is defined. The size to be allocated to the link list must be equal to MAX_DESC * 16bytes for MAX_DESC >= 1. This register is only accessible in write mode if the RX FIFO Queue 6 is not busy, see BUSY flag in RX_FQ_STS0 register

Bit 27:16 In Normal mode only the DC_SIZE[6:0] is used to define the maximum size of an RX data container for the RX FIFO Queue. The data container size is DC_SIZE[6:0] * 32bytes and one is attached to every RX descriptor. In continuous mode, it defines the size of the single data container used to write all RX messages. The overall data container size is DC_SIZE[11:0] * 32bytes for MAX_DESC > = 1. When set to 0, the RX FIFO Queue can be enabled but not started. This register is only accessible in write mode if the RX FIFO Queue 6 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.90RX_FQ_DC_START_ADD6

RX FIFO Queue 6 Data Container Start Address
This register is accessible in write mode if the RX FIFO Queue 6 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register. This register is used only in Continuous Mode

Address Offset:	0x000004bc																Initial Value:															
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the Data Container Start Address in system memory. This bit field is relevant only when the MH is configured in Continuous Mode. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This register is only accessible in write mode if the RX FIFO Queue 6 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.91RX_FQ_RD_ADD_PT6

RX FIFO Queue 6 Read Address Pointer

This register is used only in Continuous Mode.

Address Offset:	0x000004c0																Initial Value:															
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 The SW uses this register to indicate the Data Read Address of the RX message being read to the MH. This address must point to the last word of the RX message considered in the data container. This bit field is relevant only when the MH is configured in Continuous mode. The MH uses this information to ensure enough memory space is available to write the next message. For an initial start, it is mandatory to set

VAL[1:0] to 0b11, to avoid RX_FQ_RD_ADD_PT6 register to be equal to the RX_FQ_START_ADDR6 registers. Excepted for the initial value, the address value must always be word aligned (32bit), VAL[1:0] must be set to 0b00.

1.4.4.2.1.92RX_FQ_ADD_PT7

RX FIFO Queue 7 Current Address Pointer

Address Offset:	0x000004c8	Initial Value:	0x00000000

	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
Bit	VAL																															
Mode	R																															
Initial Value	0x0																															

Bit 31:0 Provide the current RX Header Descriptor address pointer for the RX FIFO Queue 7 in the system memory. To follow RX descriptor over time, refer to the RX_DESC_ADD_PT register. This address value is always word aligned (32bit).

1.4.4.2.1.93RX_FQ_START_ADD7

RX FIFO Queue 7 link list Start Address
This register is only accessible in write mode if the RX FIFO Queue 7 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x000004cc																Initial Value:															
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the start address of the RX FIFO Queue link list descriptor in system memory. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This register is only accessible in write mode if the RX FIFO Queue 7 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.94RX_FQ_SIZE7

RX FIFO Queue 7 link list and data container Size
This register is only accessible in write mode if the RX FIFO Queue 7 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x000004d0																Initial Value:															
	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
Bit					DC_SIZE																		MAX_DESC									
Mode					RW																		RW									
Initial Value					0x0																		0x0									

Bit 9:0 Define the maximum number of descriptors in the RX FIFO Queue link list. It is important to note that MAX_DESC = 0 does not prevent the RX FIFO Queue to be enabled and started. An active and running RX FIFO Queue with MAX_DESC = 0 is not allowed and will result in a DESC_ERR interrupt if no RX descriptor is defined. The size to be allocated to the

link list must be equal to MAX_DESC * 16bytes for MAX_DESC >= 1. This register is only accessible in write mode if the RX FIFO Queue 7 is not busy, see BUSY flag in RX_FQ_STS0 register

Bit 27:16 In Normal mode only the DC_SIZE[6:0] is used to define the maximum size of an RX data container for the RX FIFO Queue. The data container size is DC_SIZE[6:0] * 32bytes and one is attached to every RX descriptor. In continuous mode, it defines the size of the single data container used to write all RX messages. The overall data container size is DC_SIZE[11:0] * 32bytes for MAX_DESC > = 1. When set to 0, the RX FIFO Queue can be enabled but not started. This register is only accessible in write mode if the RX FIFO Queue 7 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.95RX_FQ_DC_START_ADD7

RX FIFO Queue 7 Data Container Start Address
This register is accessible in write mode if the RX FIFO Queue 7 is not busy, see BUSY flag in RX_FQ_STS0 register.
This register is protected by a register bank CRC defined in CRC_REG register. This register is used only in Continuous Mode

Address Offset:	0x000004d4																Initial Value:																0x00000000
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 Define the Data Container Start Address in system memory. This bit field is relevant only when the MH is configured in Continuous Mode. The VAL[1:0] bits are always assumed to be 0b00 whatever the value written. This address value must always be word aligned (32bit). This register is only accessible in write mode if the RX FIFO Queue 7 is not busy, see BUSY flag in RX_FQ_STS0 register

1.4.4.2.1.96RX_FQ_RD_ADD_PT7

RX FIFO Queue 7 Read Address Pointer
This register is used only in Continuous Mode.

Address Offset:	0x000004d8	Initial Value: 0x00000000
	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
Bit		VAL
Mode		RW
Initial Value		0x0

Bit 31:0 The SW uses this register to indicate the Data Read Address of the RX message being read to the MH. This address must point to the last word of the RX message considered in the data container. This bit field is relevant only when the MH is configured in Continuous mode. The MH uses this information to ensure enough memory space is available to write the next message. For an initial start, it is mandatory to set VAL[1:0] to 0b11, to avoid RX_FQ_RD_ADD_PT7 register to be equal to the RX_FQ_START_ADDR7 register. Excepted for the initial value, the address value must always be word aligned (32bit), VAL[1:0] must be set to 0b00.

1.4.4.2.1.97TX_FILTER_CTRL0

TX Filter Control register 0

This register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register. The register is accessible in write access in privileged mode only. This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000600																Initial Value:															
	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
Bit												IRQ_EN	EN	CC_CAN	CAN_FD	MODE	MASK								COMB							
Mode												RW	RW	RW	RW	RW	RW								RW							
Initial Value												0x0	0x0	0x0	0x0	0x0	0x0								0x0							

Bit 7:0 When COMB[n] =1 the comparison attached to the reference values (REF_VAL0 and REF_VAL1) or (REF_VAL2 and REF_VAL3) are required to accept a TX message. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 15:8 When MASK[n] =1 the reference values REF_VAL0/1 or REF_VAL2/3 are combined to define a value (REF_VAL0 or REF_VAL2) and a mask (REF_VAL1 or REF_VAL3). Otherwise, the comparison uses the REF_VAL0/1/2/3 bit field as reference value only. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 16 When set to 1 accept on match, otherwise reject on match. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 17 When set to 1 reject CAN-FD messages, otherwise accept them. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 18 When set to 1 reject Classic CAN messages, otherwise accept them. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 19 When set to 1, enable the TX filter for all TX message to be sent. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 20 When set to 1, enable the interrupt tx_filter_irq to be triggered. The interrupt is triggered when a message is rejected. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

1.4.4.2.1.98 TX_FILTER_CTRL1

TX Filter Control register 1

This register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register. The register is accessible in write access in privileged mode only. This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000604	Initial Value: 0x00000000
	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
Bit	FIELD	VALID
Mode	RW	RW
Initial Value	0x0	0x0

Bit 15:0 When VALID[n] = 1 the reference value defined for the TX filter n is valid. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register. The valid reference value used is defined as follow:

VALID[n] is assigned to TX_FILTER_REFVAL0.REF_VAL{n} (n € {0, 1, 2, 3})
VALID[n+4] is assigned to TX_FILTER_REFVAL1.REF_VAL{n} (n € {0, 1, 2, 3})
VALID[n+8] is assigned to TX_FILTER_REFVAL2.REF_VAL{n} (n € {0, 1, 2, 3})
VALID[n+12] is assigned to TX_FILTER_REFVAL3.REF_VAL{n} (n € {0, 1, 2, 3})

Bit 31:16 When FIELD[n] = 1 the TX filter element n is considering SDT, otherwise VCID, to compare with the reference value defined in TX_FILTER_REFVAL0/1/2/3. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register. The reference value to be set is defined as follow:

FIELD[n] is assigned to TX_FILTER_REFVAL0.REF_VAL{n} (n € {0, 1, 2, 3})
FIELD[n+4] is assigned to TX_FILTER_REFVAL1.REF_VAL{n} (n € {0, 1, 2, 3})
FIELD[n+8] is assigned to TX_FILTER_REFVAL2.REF_VAL{n} (n € {0, 1, 2, 3})
FIELD[n+12] is assigned to TX_FILTER_REFVAL3.REF_VAL{n} (n € {0, 1, 2, 3})

1.4.4.2.1.99 TX_FILTER_REFVAL0

TX Filter Reference Value register 0
This register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register. The register is accessible in write access in privileged mode only. This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000608				Initial Value:	0x00000000		
	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				
Bit	REF_VAL3	REF_VAL2	REF_VAL1	REF_VAL0				
Mode	RW	RW	RW	RW				
Initial Value	0x0	0x0	0x0	0x0				

Bit 7:0 Define the reference value 0. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 15:8 Define the reference value 1. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 23:16 Define the reference value 2. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 31:24 Define the reference value 3. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

1.4.4.2.1.100 TX_FILTER_REFVAL1

TX Filter Reference Value register 1

This register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register. The register is accessible in write access in privileged mode only. This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x0000060c				Initial Value:	0x00000000		
	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				
Bit	REF_VAL3	REF_VAL2	REF_VAL1	REF_VAL0				
Mode	RW	RW	RW	RW				
Initial Value	0x0	0x0	0x0	0x0				

Bit 7:0 Define the reference value 0. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 15:8 Define the reference value 1. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 23:16 Define the reference value 2. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 31:24 Define the reference value 3. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

1.4.4.2.1.101 TX_FILTER_REFVAL2

TX Filter Reference Value register 2

This register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register. The register is accessible in write access in privileged mode only. This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000610				Initial Value:	0x00000000		
	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				
Bit	REF_VAL3	REF_VAL2	REF_VAL1	REF_VAL0				
Mode	RW	RW	RW	RW				
Initial Value	0x0	0x0	0x0	0x0				

Bit 7:0 Define the reference value 0. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 15:8 Define the reference value 1. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 23:16 Define the reference value 2. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 31:24 Define the reference value 3. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

1.4.4.2.1.102 TX_FILTER_REFVAL3

TX Filter Reference Value register 3

This register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register. The register is accessible in write access in privileged mode only. This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000614																Initial Value:								0x00000000							
	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
Bit	REF_VAL3								REF_VAL2								REF_VAL1								REF_VAL0							
Mode	RW								RW								RW								RW							
Initial Value	0x0								0x0								0x0								0x0							

Bit 7:0 Define the reference value 0. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 15:8 Define the reference value 1. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 23:16 Define the reference value 2. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 31:24 Define the reference value 3. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

1.4.4.2.1.103 RX_FILTER_CTRL

RX Filter Control register

This register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register. The register is accessible in write access in privileged mode only. This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000680																Initial Value:								0x00000000							
	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
Bit											ANFF	ANMF		ANMF_FQ						THRESHOLD					NB_FE							
Mode											RW	RW		RW						RW					RW							
Initial Value											0x0	0x0		0x0						0x0					0x0							

Bit 7:0 Define the number of RX filter elements that are defined in the local memory. When set to 0, all RX messages are accepted and stored in the

RX FIFO Queue number defined by ANMF_FQ[3:0]. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 12:8 THRESHOLD defines the latest point in time to wait for the result of the RX filtering process., Once this limit is reached, the MH starts fetching an RX descriptor from S_MEM. THRESHOLD value is only used when greater than 0 and ANFF bit set to 1. See chapter "RX Filter" for an explanation how to configure THRESHOLD. When the RX filtering is not providing the result before the threshold of the RX DMA FIFO is reached, the RX message is sent to the default RX FIFO Queue mentioned in the ANMF_FQ[2:0] (only enabled when ANFF set to 1). When the level is over the threshold and the RX filtering result is already known, no action is taken. Threshold is given in number of word of 32bit. This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 18:16 Define the default RX FIFO Queue number (from 0 to 7) when non matching frames are accepted (ANMF = 1) AND/OR when the threshold mechanism is active (ANFF = 1). This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 20 When set to 1, non matching frames are accepted, otherwise rejected. It is mandatory to have the default RX FIFO Queue defined in the ANMF_FQ[2:0] bit field, enabled and started (see RX_FQ_CTRL2 and RX_FQ_CTRL0 registers). This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

Bit 21 When set to 1, frames not filtered in time and over the DMA RX FIFO level defined in THRESHOLD[4:0], are routed to the default RX FIFO Queue (defined by the ANMF_FQ[2:0] bit field). It is mandatory to have the default RX FIFO Queue defined in the ANMF_FQ[2:0] bit field, enabled and started (see RX_FQ_CTRL2 and RX_FQ_CTRL0 registers). This bit field register is only accessible in write mode if the MH is not busy, see BUSY flag in MH_STS register

1.4.4.2.1.104 TX_FQ_INT_STS

TX FIFO Queue Interrupt Status register

Address Offset:	0x00000700																Initial Value:								0x00000000							
	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
Bit									UNVALID																SENT							
Mode									RW																RW							
Initial Value									0x0																0x0							

Bit 7:0 When SENT[n] = 1, a TX message was sent from the TX FIFO Queue n and writing a 1 clears the bit

Bit 23:16 When TX FIFO Queue n loads a TX descriptor with VALID = 0, the bit UNVALID[n] will be set. Writing 1 to UNVALID[n] clears the bit.

1.4.4.2.1.105 RX_FQ_INT_STS

RX FIFO Queue Interrupt Status register

Address Offset:	0x00000704																Initial Value:								0x00000000							
	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
Bit									UNVALID																RECEIVED							
Mode									RW																RW							
Initial Value									0x0																0x0							

Bit 7:0 When RECEIVED[n] = 1, an RX message was received in the RX FIFO Queue n, writing a 1 clears the bit

Bit 23:16 When RX FIFO Queue n loads an RX descriptor with VALID=0, the bit UNVALID[n] will be set. Writing 1 to UNVALID[n] clears the bit.

1.4.4.2.1.106 TX_PQ_INT_STS0

TX Priority Queue Interrupt Status register 0

Address Offset:	0x00000708	Initial Value: 0x00000000
	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
Bit	SENT	
Mode	RW	
Initial Value	0x0	

Bit 31:0 When SENT[n] = 1 TX message was sent from the TX Priority Queue slot n, writing a 1 clears the bit

1.4.4.2.1.107 TX_PQ_INT_STS1

TX Priority Queue Interrupt Status register 1

Address Offset:	0x0000070c	Initial Value: 0x00000000
	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
Bit	UNVALID	
Mode	RW	
Initial Value	0x0	

Bit 31:0 When UNVALID[n] = 1, an invalid RX descriptor is detected while running the TX Priority Queue slot n. Writing a 1 clears the bit. When set to 1, the TX Priority Queue slot n is on hold, waiting for the SW to react. As the TX message is fully defined in system memory before starting the relevant slot, there should not be any invalid TX descriptor interrupts

1.4.4.2.1.108 STATS_INT_STS

Statistics Interrupt Status register

Address Offset:	0x00000710																Initial Value:															
	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
Bit																													RX_UNSUCC	RX_SUCC	TX_UNSUCC	TX_SUCC
Mode																													RW	RW	RW	RW
Initial Value																													0x0	0x0	0x0	0x0

Bit 0 Counter of TX message transmitted successfully has wrapped, writing a 1 clears the bit

Bit 1 Counter of TX message transmitted unsuccessfully has wrapped, writing a 1 clears the bit

Bit 2 Counter of RX message received successfully has wrapped, writing a 1 clears the bit

Bit 3 Counter of RX message received unsuccessfully has wrapped, writing a 1 clears the bit

1.4.4.2.1.109 ERR_INT_STS

Error Interrupt Status register

Address Offset:	0x00000714																Initial Value:															
	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
Bit																												DP_RX_SEQ_ERR	DP_TX_SEQ_ERR	DP_RX_ACK_DO_ERR	DP_RX_FIFO_DO_ER	DP_TX_ACK_DO_ERR
Mode																												RW	RW	RW	RW	RW
Initial Value																												0x0	0x0	0x0	0x0	0x0

Bit 0 When set to 1, a TX acknowledge data overflow is detected, writing a 1 clears the bit

Bit 1 When set to 1, an RX DMA FIFO overflow issue is detected, writing a 1 clears the bit

Bit 2 When set to 1, an RX acknowledge data overflow is detected, writing a 1 clears the bit

Bit 3 When set to 1, a TX sequence issue is detected, writing a 1 clears the bit

Bit 4 When set to 1, an RX sequence issue is detected, writing a 1 clears the bit

1.4.4.2.1.110 SFTY_INT_STS

Safety Interrupt Status register

Address Offset:	0x00000718																Initial Value:															
	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
Bit															ACK_RX_PARITY_ERR	ACK_TX_PARITY_ERR	MEM_SFTY_CE	MEM_SFTY_UE	RX_DESC_CRC_ERR	RX_DESC_REQ_ERR	TX_DESC_CRC_ERR	TX_DESC_REQ_ERR	AP_RX_PARITY_ERR	AP_TX_PARITY_ERR	DP_RX_PARITY_ERR	DP_TX_PARITY_ERR	MEM_AXI_RD_TO_ER	MEM_AXI_WR_TO_ER	DP_PRT_RX_TO_ERR	DP_PRT_TX_TO_ERR	DMA_AXI_RD_TO_ER	DMA_AXI_WR_TO_ER
Mode															RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Initial Value															0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

Bit 0 When set to 1, an AXI write access timeout issue is detected on DMA interface, writing a 1 clears the bit

Bit 1 When set to 1, an AXI read access timeout issue is detected on DMA interface, writing a 1 clears the bit

Bit 2 When set to 1, a TX_MSG timeout issue is detected, writing a 1 clears the bit

Bit 3 When set to 1, an RX_MSG timeout issue is detected, writing a 1 clears the bit

Bit 4 When set to 1, an AXI write access timeout issue is detected on local memory interface, writing a 1 clears the bit

Bit 5 When set to 1, an AXI read access timeout issue is detected on local memory interface, writing a 1 clears the bit

Bit 6 When set to 1, a TX data parity error is detected on datapath, writing a 1 clears the bit

Bit 7 When set to 1, an RX data parity error is detected on datapath, writing a 1 clears the bit

Bit 8 When set to 1, a TX address pointer parity issue is detected, writing a 1 clears the bit

Bit 9 When set to 1, an RX address pointer parity issue is detected, writing a 1 clears the bit

Bit 10 When set to 1, a TX descriptor fetched does not match the one expected, writing a 1 clears the bit

Bit 11 When set to 1, a TX descriptor has a wrong CRC, writing a 1 clears the bit

Bit 12 When set to 1, an RX descriptor fetched does not match the one expected, writing a 1 clears the bit

Bit 13 When set to 1, an RX descriptor has a wrong CRC, writing a 1 clears the bit

Bit 14 When set to 1, an uncorrectable error is detected on the local memory interface

Bit 15 When set to 1, a correctable error is detected on the local memory interface

Bit 16 When set to 1, an acknowledge data parity issue is detected on the TX path

Bit 17 When set to 1, an acknowledge data parity issue is detected on the RX path

1.4.4.2.1.111 AXI_ERR_INFO

DMA Error Information

Address Offset:	0x0000071c																Initial Value:															
	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
Bit																									MEM_RESP		MEM_ID		DMA_RESP		DMA_ID	
Mode																									R		R		R		R	
Initial Value																									0x0		0x0		0x0		0x0	

Bit 1:0 On DMA_AXI interface. Define the AXI ID used when a write or read error response is detected. According to the value, the DMA channel can be identified and so it is possible to define what's the effect of such issue.

Bit 3:2 On DMA_AXI interface. When set to 0b10, the AXI response is SLVERR. When set to 0b11, the response is DECERR. By default, set to 0b00 (OKAY)

Bit 5:4 On MEM_AXI interface. Define the AXI ID used when a write or read error response is detected. According to the value, the DMA channel can be identified and so it is possible to define what's the effect of such issue.

Bit 7:6 On MEM_AXI interface. When set to 0b10, the AXI response is SLVERR. When set to 0b11, the response is DECERR. By default, set to 0b00 (OKAY)

1.4.4.2.1.112 DESC_ERR_INFO0

Descriptor Error Information 0
If the DESC_ERR_INFO0.ADD[31:16] = 0 and DESC_ERR_INFO1.CRC[8:0], DESC_ERR_INFO1.RX_TX and DESC_ERR_INFO1.RC[4:0] are all equal to 0, the faulty descriptor is a TX descriptor fetched from L_MEM

Address Offset:	0x00000720																Initial Value:																0x00000000
	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
Bit	ADD																															
Mode	R																															
Initial Value	0x0																															

Bit 31:0 Descriptor address being used when the error is detected

1.4.4.2.1.113 DESC_ERR_INFO1

Descriptor Error Information 1
When the DESC_ERR_INFO1.CRC[8:0], DESC_ERR_INFO1.RX_TX and DESC_ERR_INFO1.RC[4:0] are all equal to 0, the faulty descriptor is a TX descriptor fetched from L_MEM only if the DESC_ERR_INFO0.ADD[31:16] = 0

Address Offset:	0x00000724																Initial Value:															
	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
Bit								CRC									RX_TX		RC					PQ	IN			FQN_PQSN				
Mode								R									R		R					R	R			R				
Initial Value								0x0									0x0		0x0					0x0	0x0			0x0				

Bit 4:0 Provide the information regarding the RX/TX FIFO Queue number or the TX Priority Queue slot having an issue

Bit 7:5 Provide the instance number defined in RX or TX descriptor logged in

Bit 8 Identify which TX queue is impacted, either the TX Priority Queue (PQ set to 1) or the TX FIFO Queues

Bit 13:9 Provide the information regarding the Rolling Counter defined in RX or TX descriptor impacted

Bit 15 RX descriptor has an issue (RX_TX set to 1), otherwise the same for a TX descriptor

Bit 24:16 CRC value defined in the RX or TX descriptor logged in

1.4.4.2.1.114 TX_FILTER_ERR_INFO

TX Filter Error Information

Address Offset:	0x00000728																Initial Value:															
	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
Bit																											FQN_PQS					FQ
Mode																											R					R
Initial Value																											0x0					0x0

Bit 0 When set to 1, one of the TX FIFO Queues has triggered the TX_FILTER_ERR interrupt, otherwise it is a TX Priority Queue slot

Bit 5:1 Provide the information of the TX FIFO Queue number or TX Priority Queue slot number which has triggered the TX_FILTER_ERR interrupt.

1.4.4.2.1.115 DEBUG_TEST_CTRL

Debug Control register

This register is only accessible in write mode if the Test Mode Key sequence has been performed prior to write. The register is accessible in write access in privileged mode only.

This register is protected by a register bank CRC defined in CRC_REG register.

Address Offset:	0x00000800																Initial Value:															
	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
Bit																						HDP_SEL									HDP_EN	TEST_IRQ_EN
Mode																						RW									RW	RW
Initial Value																						0x0									0x0	0x0

Bit 0 When writing 1, enable the control of the interrupt lines using the INT_TEST0 and INT_TEST1 registers. This bit field register is only accessible in write mode if the Test Mode Key sequence has been performed prior to write.

Bit 1 When writing 1, enable the hardware debug port to monitor MH internal signals. This bit field register is only accessible in write mode if the Test Mode Key sequence has been performed prior to write.

Bit 10:8 Define the set of signals to be monitored on the HDP[15:0] bus signal interface. This bit field register is only accessible in write mode if the Test Mode Key sequence has been performed prior to write.

1.4.4.2.1.116 INT_TEST0

Interrupt Test register 0

This register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set

Address Offset:	0x00000804																Initial Value:															
	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
Bit									RX_FQ_IRQ																TX_FQ_IRQ							
Mode									RW																RW							
Initial Value									0x0																0x0							

Bit 7:0 When writing 1 to TX_FQ_IRQ[n], triggers the interrupt line tx_fq_irq[n], those bits are auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set

Bit 23:16 When writing 1 to RX_FQ_IRQ[n], triggers the interrupt line rx_fq_irq[n], those bits are auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set

1.4.4.2.1.117 INT_TEST1

Interrupt Test register 1

This register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set

Address Offset:	0x00000808																Initial Value:															
	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
Bit												TX_PQ_IRQ	STATS_IRQ	STOP_IRQ	RX_FILTER_IRQ	TX_FILTER_IRQ	TX_ABORT_IRQ	RX_ABORT_IRQ	RX_FILTER_ERR	MEM_TO_ERR	MEM_SFTY_ERR	REG_CRC_ERR	DESC_ERR	AP_PARITY_ERR	DP_PARITY_ERR	DP_SEQ_ERR	DP_DO_ERR	DP_TO_ERR	DMA_CH_ERR	DMA_TO_ERR	RESP_ERR	
Mode												RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	
Initial Value												0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	

Bit 1:0 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set

Bit 2 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set

Bit 3	Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set
Bit 4	Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set
Bit 5	Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set
Bit 6	Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set
Bit 7	Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set
Bit 8	Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set
Bit 9	Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set
Bit 10	Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set
Bit 11	Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set
Bit 12	Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set
Bit 13	Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set
Bit 14	Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set
Bit 15	Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set
Bit 16	Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set

Bit 17 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set

Bit 18 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set

Bit 19 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set

Bit 20 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared. This bit field register is only accessible in write mode if the TEST_IRQ_EN bit in DEBUG_TEST_CTRL is set

1.4.4.2.1.118 TX_SCAN_FC

TX-SCAN first candidates register

This register gives the 4 best candidates evaluated by the TX-Scan. This register gives the first and second highest priority TX descriptor after a TX-Scan. It provides also the third and fourth candidates during a TX-Scan, considering the first and second candidates as already defined by a previous TX-Scan run.

Address Offset:	0x00000810																Initial Value:															
	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
Bit			FQN_PQSN3					FQ_PQ3			FQN_PQSN2					FQ_PQ2			FQN_PQSN1					FQ_PQ1			FQN_PQSN0					FQ_PQ0
Mode			R					R			R					R			R					R			R					R
Initial Value			0x0					0x0			0x0					0x0			0x0					0x0			0x0					0x0

Bit 0 The first candidate evaluated by TX-Scan is either a TX Priority Queue (when set to 1) or a TX FIFO Queue (when set to 0). This bit field is identical to TX_SCAN_BC.FH_PQ bit register

Bit 5:1 The first candidate is coming from either the TX FIFO Queue number N (defined by FQN in TX descriptor) or the TX Priority Queue Slot number M (defined by the PQSN in TX descriptor). The meaning of this bit field depends on the PQ0. This bit field is identical to TX_SCAN_BC.FH_FQN_PQSN bit register

Bit 8 The second candidate evaluated by TX-Scan is either a TX Priority Queue (when set to 1) or a TX FIFO Queue (when set to 0). This bit field is identical to TX_SCAN_BC.SH_PQ bit register

Bit 13:9 The second candidate is coming from either the TX FIFO Queue number N (defined by FQN in TX descriptor) or the TX Priority Queue Slot number M (defined by the PQSN in TX descriptor). The meaning of this bit field depends on the PQ0. This bit field is identical to the TX_SCAN_BC.SH_FQN_PQSN bit register

Bit 16 The third candidate evaluated by TX-Scan is either a TX Priority Queue (when set to 1) or a TX FIFO Queue (when set to 0).

Bit 21:17 The third candidate is coming from either the TX FIFO Queue number N (defined by FQN in TX descriptor) or the TX Priority Queue Slot number M (defined by the PQSN in TX descriptor). The meaning of this bit field depends on the PQ2.

Bit 24 The fourth candidate evaluated by TX-Scan is either a TX Priority Queue (when set to 1) or a TX FIFO Queue (when set to 0).

Bit 29:25 The fourth candidate is coming from either the TX FIFO Queue number N (defined by FQN in TX descriptor) or the TX Priority Queue Slot number M (defined by the PQSN in TX descriptor). The meaning of this bit field depends on the PQ3.

1.4.4.2.1.119 TX_SCAN_BC

TX-SCAN best candidates register

This register gives the first and second highest priority TX descriptor after a TX-Scan

Address Offset:	0x00000814																Initial Value:															
	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
Bit	SH_OFFSET										SH_FQN_PQSN					SH_PQ	FH_OFFSET										FH_FQN_PQSN					FH_PQ
Mode	R										R					R	R										R					R
Initial Value	0x0										0x0					0x0	0x0										0x0					0x0

Bit 0 First highest priority candidate evaluated by TX-Scan. It is either a TX Priority Queue (when set to 1) or a TX FIFO Queue (when set to 0).

Bit 5:1 First highest priority candidate coming from either the TX FIFO Queue number N (defined by FQN in TX descriptor) or the TX Priority Queue

Slot number M (defined by the PQSN in TX descriptor). The meaning of this bit field depends on the FH_PQ.

Bit 15:6 First highest priority candidate offset in multiple of 32bytes (TX descriptor size). This register is relevant only for the TX FIFO Queue. It provides the index of the TX descriptor in the TX FIFO Queue which is in use on the CAN bus. When FH_PQ = 1 it is set to 0.

Bit 16 Second highest priority candidate evaluated by TX-Scan. It is either a TX Priority Queue (when set to 1) or a TX FIFO Queue (when set to 0).

Bit 21:17 Second highest priority candidate coming from either the TX FIFO Queue number N (defined by FQN in TX descriptor) or the TX Priority Queue Slot number M (defined by the PQSN in TX descriptor). The meaning of this bit field depends on the SH_PQ.

Bit 31:22 Second highest priority candidate offset in multiple of 32bytes (TX descriptor size). This register is relevant only for the TX FIFO Queue. It provides the index of the TX descriptor in the TX FIFO Queue which is about to be sent on the CAN bus. When SH_PQ = 1 it is set to 0.

1.4.4.2.1.120 TX_FQ_DESC_VALID

Valid TX FIFO Queue descriptors in local memory

Address Offset:	0x00000818																Initial Value:															
	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
Bit									DESC_NC_VALID																DESC_CN_VALID							
Mode									R																R							
Initial Value									0x0																0x0							

Bit 7:0 When DESC_CN_VALID[n] = 1, the current/next TX descriptor for the TX FIFO Queue n is available in L_MEM

Bit 23:16 When DESC_NC_VALID[n] = 1, the next/current TX descriptor for the TX FIFO Queue is available in L_MEM

1.4.4.2.1.121 TX_PQ_DESC_VALID

Valid TX Priority Queue descriptors in local memory

Address Offset:	0x0000081c																Initial Value:															
	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
Bit	DESC_VALID																															
Mode	R																															
Initial Value	0x0																															

Bit 31:0 When DESC_VALID[n] = 1, the TX descriptor assigned to the slot n in local memory is valid

1.4.4.2.1.122 CRC_CTRL

CRC Control register

Address Offset:	0x00000880																Initial Value:															
	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
Bit																																START
Mode																																W
Initial Value																																0x0

Bit 0 Writing a 1 to this bit triggers the HW CRC check of registers. This action can be done any time for a sanity check

1.4.4.2.1.123 CRC_REG

CRC register

Address Offset:	0x00000884																Initial Value:															0x00000000
	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
Bit	VAL																															
Mode	RW																															
Initial Value	0x0																															

Bit 31:0 CRC value of all the registers protected by CRC. Once done, a write to the START bit in the CRC_CTRL register must be done

1.4.4.3 Local Memory Map (L_MEM Map)

To perform the RX filtering and the TX-SCAN, the MH requires a local memory. This local memory, called L_MEM, is addressable through the MEM_AXI interface.
The MEM_AXI interface can address up to 64KBytes with a 32bit data bus width.

The L_MEM stores all RX filter elements and Header Descriptor for TX FIFO and Priority Queues. When considering TX FIFO Queues, the next TX Header Descriptor are also stored in L_MEM (used for TX-SCAN).

1.4.4.3.1 TX Descriptors

The TX FIFO Queue descriptors are organized into the L_MEM starting at the base address defined in **TX_DESC_MEM_ADD.FQ_BASE_ADDR[15:0]**. Up to 8 memory locations of size 8*32bit, are required to hold the TX Header Descriptor of every TX FIFO Queues.

Every TX FIFO Queue, when active, has its current and next descriptor defined in the L_MEM for the TX-SCAN process. This means, for a given TX FIFO Queue, memory space must be double the size. The current and the next TX Header Descriptor are used for the TX-SCAN.

The TX Priority Queue descriptors are organized in the L_MEM starting at the base address defined in **TX_DESC_MEM_ADD.PQ_BASE_ADDR[15:0]**. Up to 32 memory location of size 8*32bit, is required to hold the TX Header Descriptors of every slot. As there is only one TX message per slot, there is no need to allocate more space.

The TX descriptor elements are organized in 32bit word and so any offset would be a multiple of 8.

Here below is the memory organization of the TX descriptors considering N TX FIFO Queues and M TX Priority Queue Slots:

Memory Base Address	Offset	Name	Bit Field	Description
FQ_BASE_ADDR[15:0]	0x0+0x40*n	TX FIFO Queue n (current/next TX Header Descriptor) (0<= n <N)	Element 0	TX Header Descriptor, see TX descriptor, TX Message and TX FIFO Queue chapters
	0x4+0x40*n		Element 1	
	0x8+0x40*n		Element 2: TS0	
	0xC+0x40*n		Element 3: TS1	
	0x10+0x40*n		Element 4: T0	
	0x14+0x40*n		Element 5: T1	
	0x18+0x40*n		Element 6: T2/TD0	
	0x1C+0x40*n		Element 7: TX_AP/TD1	
	0x20+0x40*n	TX FIFO Queue n (next/current TX Header Descriptor) (0<= n <N)	Element 0	TX Header Descriptor, see TX descriptor, TX Message and TX FIFO Queue chapters
	0x24+0x40*n		Element 1	
	0x28+0x40*n		Element 2: TS0	
	0x2C+0x40*n		Element 3: TS1	
	0x30+0x40*n		Element 4: T0	
	0x34+0x40*n		Element 5: T1	
	0x38+0x40*n		Element 6: T2/TD0	
	0x3C+0x40*n		Element 7: TX_AP/TD1	
PQ_BASE_ADDR[15:0]	0x0+20*m	TX Priority Queue slot m (0<= m <M)	Element 0	TX Header Descriptor, see TX descriptor, TX Message and TX FIFO Queue chapters
	0x4+0x20*m		Element 1	
	0x8+0x20*m		Element 2: TS0	
	0xC+0x20*m		Element 3: TS1	
	0x10+0x20*m		Element 4: T0	
	0x14+0x20*m		Element 5: T1	
	0x18+0x20*m		Element 6: T2/TD0	
	0x1C+0x20*m		Element 7: TX_AP/TD1	

As the L_MEM can be shared across several MH, the SW has some flexibility to allocate TX FIFO/Priority Queue descriptors anywhere and according to the usage of the application. As an example, if only 4 TX FIFO Queues are required with a TX Priority Queue with 16 slots, the expected memory size would be half compared to the maximum configuration possible. It is obvious that this kind of configuration would assume that TX FIFO Queues are continuous, meaning 0, 1, 2 and 3 AND TX Priority Queue slots 0, 1, ... and 15.
It has to be considered, that if more TX FIFO Queue and TX Priority Queue slots are required, more memory space would then need to be allocated. As a matter of fact, if the SW enables all TX FIFO

Queue and TX Priority Queue slots, the worst configuration would be a memory space configured with 8 TX FIFO Queues and 32 TX Priority Queues.

1.4.4.3.2 RX Filter Elements and Ref/Mask Pairs

The filter elements to be parsed are stored in the L_MEM on a 32bit word. The global setting of the RX Filter is defined by the **RX_FILTER_CTRL** register and will apply to all filter elements. Several filter elements can be defined (where n is from value 0 to 255) with up to m reference/mask pair (where m is from 0 to 255). The number of elements is defined in the **RX_FILTER_CTRL.NB_FE[7:0]** bit field register, the value 0 being assigned to no RX filters. Thus, only 255 RX filter elements can be defined for RX messages.

The Reference (REFm) and Mask (MSKm) pairs are defined after the full list of RX Filter Elements as defined below.

Memory Base Address	Offset (1<=n<=255) (0<=m<=255)	Name (1<=n<=255) (0<=m<=255)	Description
BASE_ADDR[15:0]	0x0	FE0	Define the RX filter element 0
	-	-	-
	0x4*n-1	FEn-1	Define the RX filter element n-1
BASE_ADDR[15:0]+0x4*n	0x0	REF0	RX Filter Reference value 0
	0x4	MSK0	RX Filter Reference mask 0
	-	-	-
	0x0+0x8*m	REFm	RX Filter Reference value m
	0x4+0x8*m	MSKm	RX Filter Reference mask m

As the L_MEM can be shared across several Message Handlers, the SW has some flexibility to allocate RX filter elements and reference/mask pairs anywhere and according to the usage of the application. As a memory space of 64Kbyte is addressable, the start address of those elements is defined in the **RX_FILTER_MEM_ADD. BASE_ADDR[15:0]** bit field register.

1.4.5 Functional Description

The MH can manage concurrently up to 8 TX FIFO Queues, up to 8 RX FIFO Queues and up to 32 slots defined in a TX Priority Queue.

The Message Handler is using the principle of linked list to define RX and TX FIFO Queues, as well as the TX Priority Queue.

The TX messages are managed by TX descriptors which define the TX message header information and the address of its payload. The payload buffer can be defined in any memory location. More information provided by chapter TX Descriptor.

The RX messages are written to the memory based on the information defined in RX descriptors and/or configuration registers. The RX message data can be stored in any memory location. More information provided by chapter RX Descriptor.

The RX FIFO Queue can support Classical CAN, CAN FD, and CAN XL frame format.
The TX FIFO Queues and TX Priority Queue Slots can support Classic, CAN FD and CAN XL frame format.
The TX Message Handler processes the TX messages while the RX Message Handler takes care of the RX messages.
Both share the Descriptor Message Handler to get their TX and RX descriptors respectively. This module also updates the status at the TX/RX FIFO Queues or TX Priority Queue when a transfer is completed. A dedicated sub-module in the Descriptor Message Handler is assigned to the TX path and one for the RX path, they can run concurrently.

The RX/TX FIFO Queues and TX Priority Queue can be fully defined in E_MEM when the overall system latency is low. This means, RX message data, TX message payload data and TX/RX descriptors can be allocated to the same external memory. For high system latency, it is essential that the RX/TX descriptors are fetched from SRAM (low access time) while still leaving the RX message data and TX message payload data in the E_MEM.

The selection of the highest priority TX message and the RX message filtering are done locally using the L_MEM. Therefore, the highest priority message to be sent is defined in a shorter time.
Regarding the RX filtering, the RX filter elements are fetched from the L_MEM to reduce the processing time to accept or reject an RX message before a new one comes in.

The MH can drive only one Protocol Controller using the TX_MSG and RX_MSG interfaces.

1.4.5.1 TX Message Handler

The TX Message Handler is in charge of TX FIFO Queues and TX Priority Queue management. Therefore, the TX Message Handler requests the TX descriptors whenever required, arbitrates the TX descriptors according to their IDs and selects the high priority TX message to be sent to the PRT.

Finally, once a TX descriptor is selected and the PRT is winning the arbitration on CAN bus, it fetches the payload data assigned to that descriptor from the S_MEM.

The internal arbitration on TX descriptors is called TX-SCAN to avoid a conflict with the arbitration done on the CAN bus.

A TX filter is put in place to ensure only the relevant TX messages will be sent through the CAN bus.

1.4.5.1.1 Block Diagram

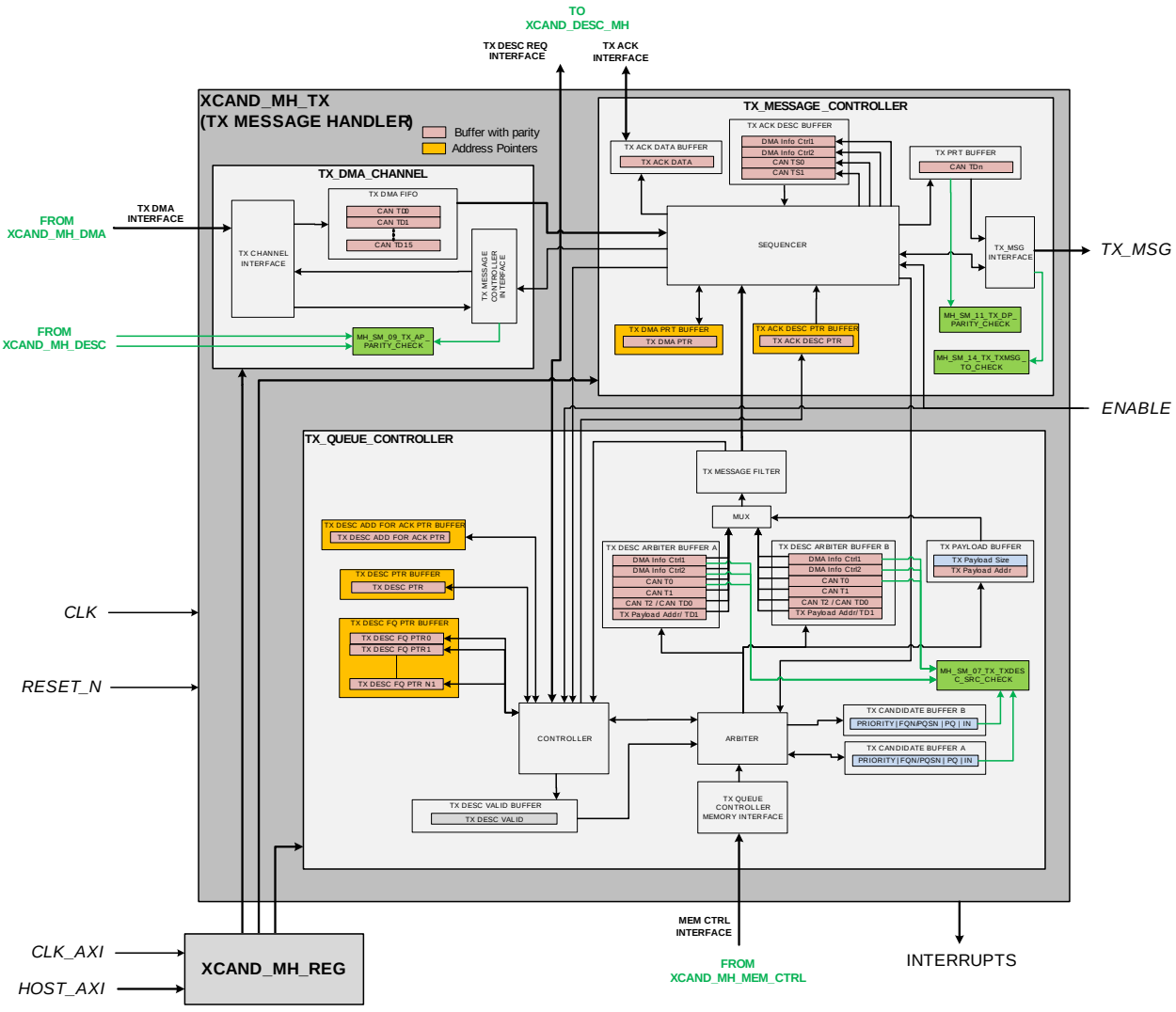

Figure: TX Message Handler block diagram

1.4.5.1.2 Block Description

Several blocks are used to manage the TX message TX FIFO Queues and the TX Priority Queue

1.4.5.1.2.1 TX DMA CHANNEL INTERFACE:

This block interfaces the DMA MESSAGE HANDLER to send read commands to the S_MEM. It will also buffer payload data in a local TX DMA FIFO before transferring the data to the TX MESSAGE CONTROLLER. The size of the payload data for a CAN frame can be in the range of 8 bytes for Classical CAN and up to 2048 bytes for CAN XL.
Every TX descriptor defines the size of the data to be executed. Only one DMA transfer request is performed per TX descriptor. Every information related to the data transfer is set by the TX MESSAGE CONTROLLER. The TX DMA FIFO size is set to two maximum burst lengths to allow continuous

execution of data transfers. As soon as the TX DMA FIFO has enough space to load a new burst, the DMA MESSAGE HANDLER will initiate a new fetch from the S_MEM. Only one data transfer can be executed at a time. As a matter of fact, when the first defined data transfer is finished, meaning the data to be read are inside the TX DMA FIFO, a second data transfer can be started. The buffered data from the first transfer are used for continuous data transfer to the TX MESSAGE CONTROLLER while waiting for the second DMA transfer.
Only the address pointer *ADDR_PTR*[31:0] and the *SIZE*[10:0] (size of the transfer) are required to fetch the payload data.
The other data transfer parameters are static and defined using control registers.
The TX DMA CHANNEL can accept only one data transfer definition at a time in the TX DMA PTR BUFFER, so one data transfer can be performed at a time.

1.4.5.1.2.2 TX MESSAGE CONTROLLER:

This block is in charge of sequencing the TX message data to the TX_MSG interface. Two sources of data are used to build the TX message. The first data comes from the TX descriptor, which contains the header and the first payload words of the CAN frame. This TX descriptor comes from the TX QUEUE CONTROLLER and is provided by the ARBITER.
Once the TX descriptor is executed, the address pointer defined in the descriptor is used to fetch further payload data from the S_MEM thanks to the TX DMA CHANNEL.

The TX MESSAGE CONTROLLER is in charge of managing new TX descriptors when several descriptors are used for one TX message. Any new TX message to be sent is solely provided by the ARBITER.

As all TX messages are managed by the TX MESSAGE CONTROLLER, once a message is sent successfully or not to the PRT, an acknowledge descriptor is provided to the DESCRIPTOR MESSAGE HANDLER to be written back to the first descriptor of the TX message. If some issues are detected, the current message is cancelled and all the traffic from the S_MEM is aborted. Once done, a new TX message must be already provided by the ARBITER.

The PRT signalizes via *ENABLE* whether it requires message handling or not. When this signal goes low, the MH must stop its current activities. This means the TX FIFO Queues and TX Priority Queue are put on hold as well as all the relevant traffic from and to the S_MEM must be aborted.

1.4.5.1.2.3 TX QUEUE CONTROLLER:

This block manages the TX FIFO Queues and the TX Priority Queue as well as the TX-SCAN. As soon as a TX FIFO Queue is started, and/or a TX Priority Queue slot is valid, the TX QUEUE CONTROLLER will fetch the appropriate TX descriptor from the S_MEM. Those descriptors are stored in the L_MEM for further processing. The TX descriptors (only part of it) are fetched from the L_MEM and analyzed to find out the TX message having the highest priority.
The one selected is stored locally for the TX MESSAGE CONTROLLER to be read. This block computes the address to read the next TX descriptor for every running TX FIFO Queue, once the current TX

message is selected by the TX-SCAN. The block also manages the active slot from the TX Priority Queue when a new one being declared.
All relevant information for the TX MESSAGE CONTROLLER is provided by this block.
The TX filter uses configuration registers to select between TX messages to be sent to the CAN bus and TX messages to be discarded.

When the TX-Scan (selection of the TX message with the highest priority) is done, the selected TX descriptor is read from the L_MEM. To ensure that it is the one already selected, some TX descriptor bit fields are checked against the expected value stored locally by the TX-Scan. In case one of the bit fields, listed below, does not match a *TX_DESC_REQ_ERR* signal is triggered to the system:

- The IN (instance number)
- The FQN (TX FIFO queue number) if PQ = 0
- The PQSN (TX Priority Queue slot number) if PQ = 1
- The PQ (Priority Queue flag)
- The Priority Value assigned to the TX message

1.4.5.2 RX Message Handler

The RX Message Handler is in charge of the RX FIFO Queues. Every RX FIFO Queue uses a linked list of RX descriptors to identify the exact location in S_MEM to store the message. The RX Message Handler requests the RX descriptor whenever required, e.g., when RX filter result of an accepted incoming RX Message becomes available.
The RX filter identifies any incoming RX messages and determines whether it must be rejected (not stored) or accepted (stored into one of the RX FIFO queues, defined by the RX filter).

1.4.5.2.1 Block Diagram

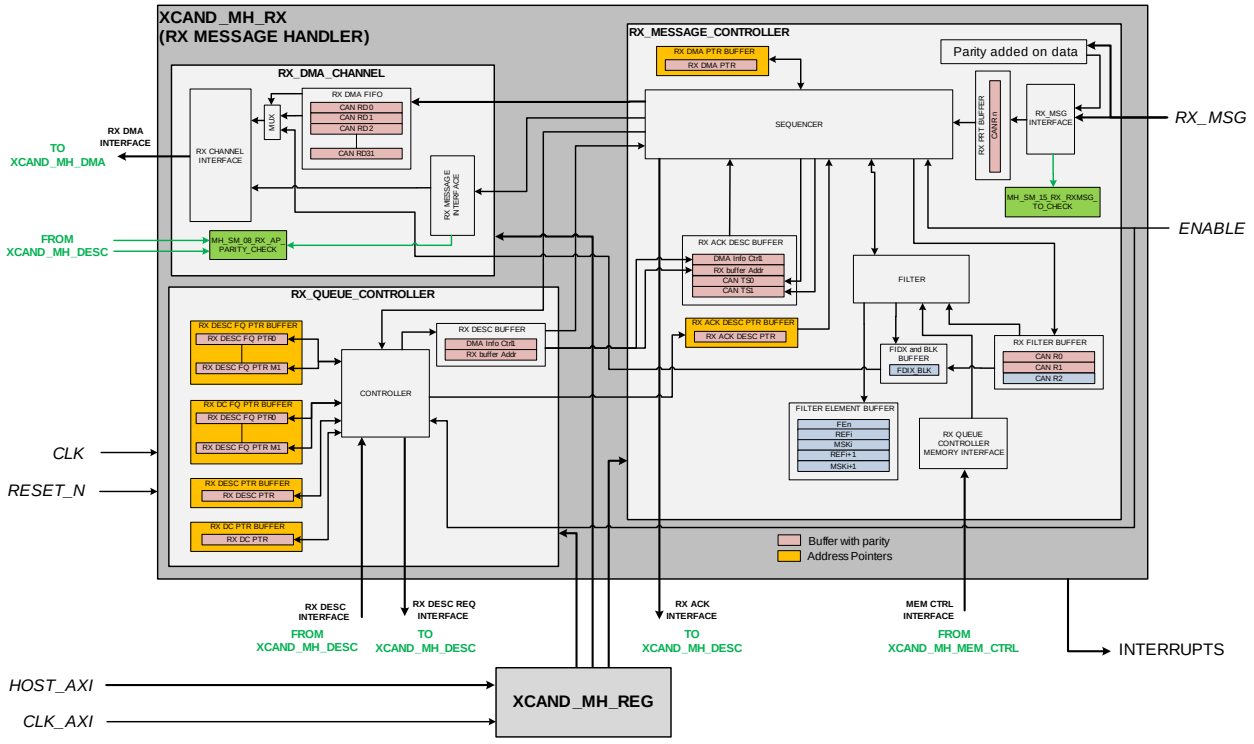

Figure: RX Message Handler block diagram

1.4.5.2.2 Block Description

Several blocks are used to manage the RX FIFO Queues:

1.4.5.2.2.1 RX DMA CHANNEL

This block interfaces the DMA MESSAGE HANDLER to send write commands to the S_MEM. It will also buffer the RX message data in a local RX DMA FIFO before sending the data to the S_MEM. The size of the payload data for a CAN frame can be the size of 8 bytes for Classical CAN and up to 2048 bytes for CAN XL.

Every RX descriptor of the same RX FIFO Queue has a fixed buffer size to hold data. The size of the overall transfers is stored locally to identify how many descriptors are required for the RX message and what the size of each DMA data transfer is.

As a matter of fact, when the RX message exceeds the maximum buffer pointed by the current descriptor, one or several DMA data transfers are executed. In other words, there are as many DMA data transfer as RX descriptors per RX message.

Only the address pointer *ADDR_PTR[31:0]* and the *SIZE[10:0]* (size of the transfer) are required to fetch the payload data. The other data transfer parameters are static and are defined in control registers.

The RX DMA CHANNEL can accept only one data transfer definition in the RX DMA PTR BUFFER, so one data transfer can be performed at a time.

1.4.5.2.2.2 RX MESSAGE CONTROLLER

This block is in charge of sequencing the RX message data from the RX_MSG interface to the RX_DMA_CHANNEL and to filter the incoming messages (refer to the RX Filter chapter for more details).

The RX message is managed by the RX MESSAGE CONTROLLER. Once a message is received successfully, an acknowledge descriptor is provided to the DESCRIPTOR MESSAGE HANDLER to be written back to the first descriptor of the RX message. This first descriptor is used along the process of receiving a message and is the only one which is acknowledged and holds the header data.
If an error was detected, the current message will be cancelled and the storage to the S_MEM will be aborted. Once done, a new RX message can be processed, and the RX descriptors of the previously aborted message are reused.

To avoid duplication of buffers, the data from the PRT is stored directly into the RX DMA FIFO without waiting for the result of the filter. Once the result of the filter is known (RX message accepted or not accepted), the CAN data being received is stored in the S_MEM or discarded.

The PRT signalizes via *ENABLE* whether it is active and requires message handling or not. When this signal is going low, the MH stops current activities. This means that the RX FIFO queues are put on hold as well as the traffic from and to the S_MEM will be aborted.

1.4.5.2.2.3 RX QUEUE CONTROLLER

This block manages the RX FIFO Queues and keeps track of the write pointers to use for each of them. As soon as an RX FIFO Queue is started, the RX QUEUE CONTROLLER is allowed to request descriptors from the DESCRIPTOR MESSAGE HANDLER. The descriptor to be used is stored into the local RX DESC BUFFER and is the result of a request to the DESCRIPTOR MESSAGE HANDLER when the RX FIFO Queue is identified by the Filter. This block also computes the address to read the next RX descriptor for every RX FIFO Queues running, once used. All the relevant information to write data to the S_MEM or to generate an interrupt when receiving a message is provided to the RX MESSAGE CONTROLLER.
In case that several descriptors are required for one message, the RX QUEUE CONTROLLER can request the next descriptor from the DESCRIPTOR MESSAGE HANDLER as soon as RX MESSAGE CONTROLLER has taken over the current descriptor.

1.4.5.3 Descriptor Message Handler

The Descriptor Message Handler is in charge of providing RX descriptors from the S_MEM, used by the RX MESSAGE HANDLER, respective TX descriptors used by the TX MESSAGE HANDLER.

As soon as an RX or a TX message was completed, it provides the acknowledge data and message header to the dedicated first descriptor in the S_MEM.

This sub-module only manages RX/TX descriptors fetched and acknowledged on request from the TX MESSAGE HANDLER and the RX MESSAGE HANDLER.

As the RX and TX and Acknowledge paths are fully concurrent, it would be up to the DMA controller (managing the traffic from/to the S_MEM) to decide which request to serve first.

As the CAN bus is unidirectional, there should be a low collision rate on the AXI bus interface on the same channel.

The parallel processing of TX/RX descriptors will decouple functions between the two paths. Such approach relaxes the constraints on those two concurrent data flows, considering use cases where both are active at the same time. Furthermore, while receiving a CAN Frame, TX descriptors can be fetched from the S_MEM on request or while executing RX FIFOs. This approach lowers the complexity of use case management.

Regarding the acknowledge of descriptors, the same strategy is used, i.e., the acknowledge path does not interfere with the RX and TX data path.

Any configuration register is defined into the main register bank of the Message Handler.

1.4.5.3.1 Block Diagram

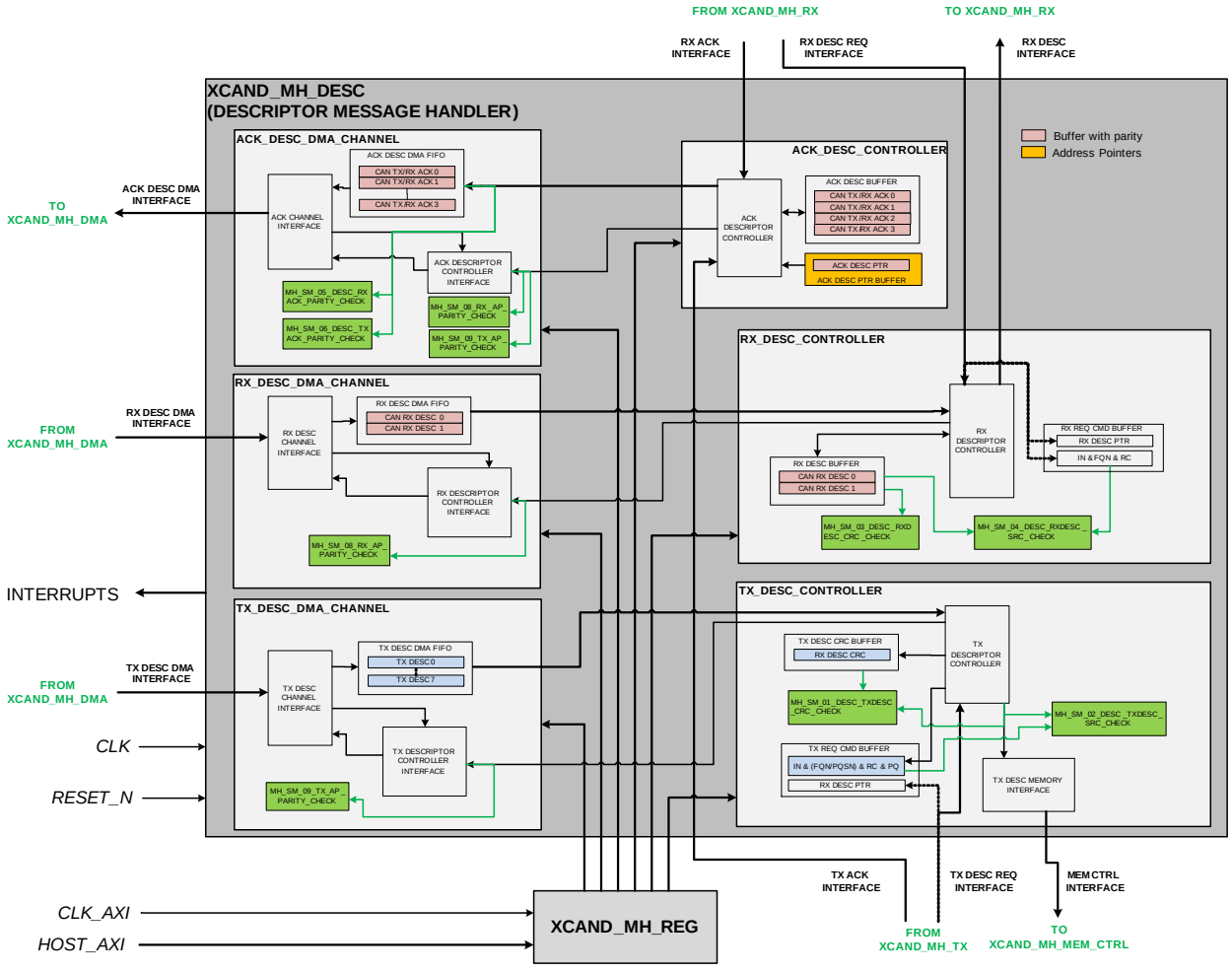

Figure: Descriptor Message Handler block diagram

1.4.5.3.2 Block Description

1.4.5.3.2.1 TX_DESC_CONTROLLER

This block stores read descriptor requests from the TX_MESSAGE_HANDLER and sends them to the TX_DESC_DMA_CHANNEL. It can accept up to two requests when there is a need to pre-fetch TX descriptors.

To provide the request to the TX_DESC_DMA_CHANNEL, the block sends only the address of the TX descriptor *ADDR_PTR[31:0]* (the size of the TX descriptor is always identical). Other control signals manage the handshaking. On top of those information, an abort signal is provided to stop the current data transfer on the DMA channel, when requested by the TX_MESSAGE_HANDLER.

Once a TX descriptor is provided by the TX_DESC_DMA_CHANNEL, several checks are performed to ensure the correctness of the descriptor and its validity. These checks are (in the order):

- The VALID bit in the TX descriptor is checked to ensure descriptor is valid. The following check is performed only if the VALID bit equals 1

- A CRC check is done on the TX descriptor. In case a CRC error is detected, a CRC error is triggered to the system using the *TX_DESC_REQ_ERR* signal. The following check is performed only if there is no CRC issue

- The instance number **IN[2:0]**, either the TX FIFO Queue number **FQN[3:0]** or the TX FIFO Queue slot number **PQSN[4:0]**, the rolling counter **RC[4:0]** bit fields of the TX descriptor and the Priority Queue bit **PQ** are checked against the expected values from the request (see TX descriptor definition chapter for more detail on those bit fields). In case that one of the bit fields does not match, a *TX_DESC_REQ_ERR* signal is triggered to the system

Whatever the result of the checks done on the TX descriptor, it is always written to the L_MEM. Doing so, the wrong TX descriptor can be read from the L_MEM, if required for debug purpose.

To store the TX descriptor, a write access is performed to the L_MEM through the memory controller interface. As the size of the TX descriptor to write does not change, the number of words to be written is identical for all descriptors. As soon as a TX descriptor is checked and no issue is identified, it is written to the L_MEM and a notification is sent to the TX_MESSAGE_HANDLER.

The TX descriptor from the S_MEM is stored locally for filtering. Once stored and accepted, it is written to the L_MEM. In case a Header Descriptor is rejected, the *TX_FILTER_IRQ* is triggered to the system. The TX MESSAGE CONTROLLER is notified that the requested TX descriptor is rejected and will not be provided. Refer to the TX Filter chapter for a detailed description.

1.4.5.3.2.2 TX_DESC_DMA_CHANNEL

This block interfaces the DMA_CONTROLLER to send read commands to the S_MEM. It will also hold the TX descriptors in a local DMA FIFO before sending the data to the TX_DESC_CONTROLLER when available and complete. As the TX descriptor has a fixed size (8 words of 32bit), the data transfers, that are executed by the DMA channel, will always be the same. Only the address pointer *ADDR_PTR[31:0]* is required to fetch the TX descriptor. The other transfer parameters are static and are stored in control registers. As the received FIFO can accept only one TX descriptor, only one data transfer can be performed at a time. There is no check performed by this block as everything is done by the TX_DESC_CONTROLLER which hold the read request definition. More details provided by the DMA CONTROLLER chapter.

1.4.5.3.2.3 RX_DESC_CONTROLLER

This block is in charge of storing read descriptor requests from the RX_MESSAGE_HANDLER and to send them to the RX_DESC_DMA_CHANNEL. It is possible to accept up to two requests when there is a need to pre-fetch RX descriptors for large payload data defined in RX messages.

To provide the request to the RX_DESC_DMA_CHANNEL, this block sends the address of the RX descriptor *ADDR_PTR[31:0]*. Other control signals manage the handshaking. On top of those information, an *ABORT* signal is provided to stop the current data transfer on the DMA channel when requested by the RX_MESSAGE_HANDLER.

Once an RX descriptor is provided by the RX_DESC_DMA_CHANNEL, several checks are performed to ensure the correctness of the descriptor. These checks are (in the order):

- The VALID bit in the RX descriptor is checked to ensure descriptor is valid. The following check is performed only if the VALID bit equals 0
- A CRC check is done on the RX descriptor. When the CRC is valid, the RX descriptor is sent to the RX_MESSAGE_HANDLER, otherwise a CRC error is triggered to the system using the *RX_DESC_REQ_ERR* signal. The following check is performed only if there is no CRC issue
- The instance number **IN[2:0]**, the RX FIFO Queue number **FQN[3:0]** and the rolling counter **RC[4:0]** bit fields of the RX descriptor are checked against the expected value mentioned in the request (see RX descriptor definition chapter for more detail on those bit fields). In case one of the bit fields does not match, an RX_*DESC_REQ_ERR* signal is triggered to the system

1.4.5.3.2.4 RX_DESC_DMA_CHANNEL

This block interfaces the DMA_CONTROLLER to send read commands to the S_MEM. It will also hold the RX descriptors in a local DMA FIFO before sending the data to the RX_DESC_CONTROLLER when available and complete. As the RX descriptors have the same size (2 words of 32bit), the data transfer to be executed by the DMA channel will always be the same. Only the address pointer *ADDR_PTR[31:0],* to fetch the RX descriptor, is required. The other data transfer parameters are static and defined using control registers. As the received FIFO can accept only one RX descriptor, only one data transfer can be performed at a time. There is no check performed by this block as everything is done by the RX_DESC_CONTROLLER holding the read request definition. For more details on the DMA_CONTROLLER interface, see the relevant chapter.

1.4.5.3.2.5 ACK_DESC_CONTROLLER:

This block manages the RX_MESSAGE_HANDLER and TX_MESSAGE_HANDLER request when an RX or TX descriptor being executed needs to be acknowledged. As soon as the RX_MESSAGE_HANDLER has completed its execution using one RX descriptor, the relevant information (transfer status and errors mainly) of that transfer must be sent back to the first descriptor. To do so, the RX_MESSAGE_HANDLER and TX_MESSAGE_HANDLER will send a request to the ACK_DESC_CONTROLLER to write acknowledge data into the respective Header Descriptor.
The ACK_DESC_CONTROLLER can only accept data when the ACK DESC DMA FIFO has enough data to store it. If this DMA FIFO cannot accept the data, it will hold any request from either RX_MESSAGE_HANDLER and/or TX_MESSAGE_HANDLER. Acknowledge data are build and stored in

the RX_MESSAGE_HANDLER and TX_MESSAGE_HANDLER. This way, any updates along the reception or transmission of a TX message will automatically be done locally on the sub-module. As the CAN bus is unidirectional, there should be no conflict regarding RX and TX descriptors being acknowledged at the same time. The only exception would appear when the PRT is set in loopback mode. As soon as the ACK DESC DMA FIFO in ACK_DESC_DMA_CHANNEL provides the right FIFO level to receive one burst of data, the ACK_DESC_CONTROLLER will write those data and will push the address pointer of those data. Despite that RX and TX acknowledge requests may not occur at the same time, the higher priority is always given to the RX path. The ACK_DESC_CONTROLLER will always start writing the acknowledge data (always 4x32bit) to the DMA FIFO in ACK_DESC_DMA_CHANNEL whatever the request source is, either TX_MESSAGE_HANDLER or RX_MESSAGE_HANDLER. At last, it will write the address pointer of that descriptor triggering at the same time a new DMA data transfer. The option, to provide a priority signal to define the urgency of the writing, exists.

1.4.5.3.2.6 ACK_DESC_DMA_CHANNEL

This block interfaces the DMA_CONTROLLER to send write commands to the S_MEM. It also holds bursts to be sent over the interconnect into a local DMA FIFO before triggering the DMA_CONTROLLER to send it to the S_MEM.

As the acknowledge data for RX and TX descriptors has a fixed size (4 words of 32bit), the data transfer to be executed by the DMA channel will always be the same. Only the address pointer to write the burst is required as well as a priority signal to manage the urgency of the request. The other data transfer parameters are statics and are provided by control registers. As the transmit DMA FIFO can accept only one burst, one transfer can be performed at a time. More details are provided by the chapter DMA_CONTROLLER.

1.4.5.4 DMA Message Handler

The DMA CONTROLLER reads and writes bursts of data from and to the S_MEM through its AXI4 master interface *DMA_AXI* (compliant to AMBA 4 ARM Ltd protocol, see [5]). In that sense the DMA CONTROLLER manages request commands from sub-module that is in charge of sending/receiving TX/RX messages as well as fetching RX/TX descriptors.

It is in charge of providing data to the sub-module which is responsible to send TX messages (TX MESSAGE HANDLER) as well as to the sub-module that manages the RX and TX descriptors (DESCRIPTOR MESSAGE HANDLER). It manages all data from a received RX message (RX MESSAGE HANDLER) as well as writes back information into RX/TX descriptors when required (DESCRIPTOR MESSAGE HANDLER).

1.4.5.4.1 Block Diagram

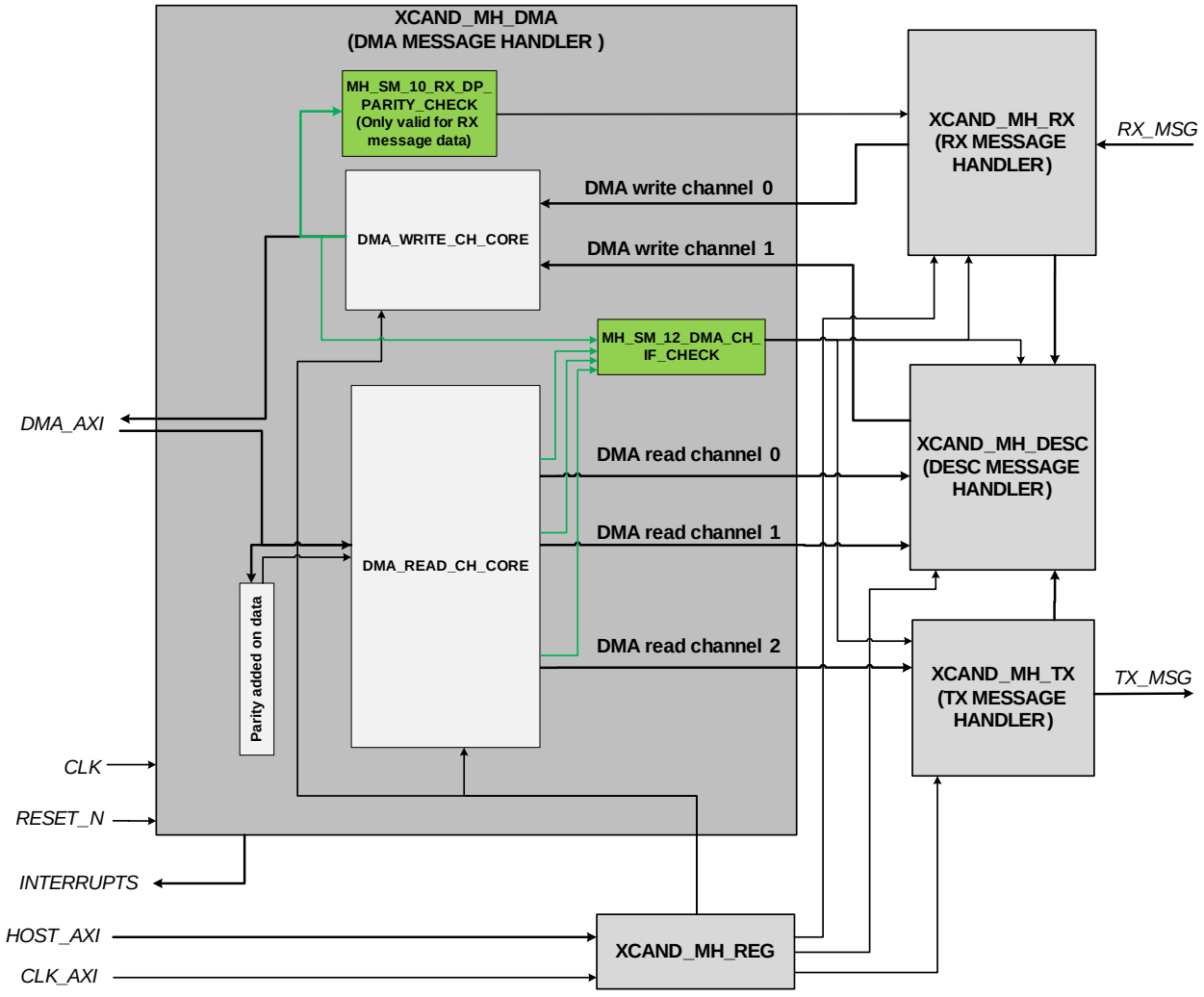

Figure: DMA Message Handler block diagram

1.4.5.4.2 Block Description

The DMA build in the XCAND_MH_DMA block has a static configuration, once the SW has written the registers, they must not be changed excepted if all DMA channels are stopped.

An arbitration process will take place to define which request command is to be served first.
As several concurrent read and write accesses can be foreseen, refer to
AXI_PARAMS.AR_MAX_PEND[1:0] and **AXI_PARAMS.AR_MAX_PEND[1:0]** bit field registers.

To maximize the AXI throughput, whatever the number of data transfer to be done, the DMA Controller ensures the usage of the maximum burst length whenever possible. To do so, the DMA Controller is always trying to generate a burst length for the first transfer to get an aligned address burst size for the next data to be transferred (maximize the usage of maximum burst size for transfers).

The *RESP_ERR[1:0]* interrupts are used to trigger the system for any bus error, when reading or writing the S_MEM and L_MEM.

Before starting any transfer, a DMA read/write channel must be enabled. The **TX_FQ_CTRL2.ENABLE[n]**, **RX_FQ_CTRL2.ENABLE[n]** and **TX_PQ_CTRL2.ENABLE[n]** bit field registers are used to identify when the DMA channels are required. If none of those enable bit are set to 1, no data transfer can occur.

The DMA is intended to:

- Write all received RX message data coming from the PRT to the S_MEM at a defined address location (specified into RX descriptors). This traffic does concern only the RX MESSAGE HANDLER
- Write the acknowledge data of TX messages back to the relative TX descriptor. This traffic is owned by the DESCRIPTOR MESSAGE HANDLER
- Write the acknowledge data of RX messages back to the appropriate RX descriptor. This traffic is owned by the DESCRIPTOR MESSAGE HANDLER
- Read TX descriptors where the TX message header is defined with some other relevant information like the address pointer of the payload data. This traffic is owned by the DESCRIPTOR MESSAGE HANDLER
- Read RX descriptors according to the RX message being filtered to identify which location to write the received data. This traffic is owned by the DESCRIPTOR MESSAGE HANDLER
- Read TX message payload data from the S_MEM when the corresponding message header is winning the CAN bus arbitration. This traffic does concern only the TX MESSAGE HANDLER

1.4.5.4.2.1 DMA_WRITE_CH_CORE:

This block is in charge of:

- Writing data to the S_MEM and to have those transfers compliant to the AXI4 AMBA protocol
- Providing the appropriate write burst length for a maximum system bus efficiency according to the number of data to be sent
- Reading the relevant amount of data from a defined DMA write channel through the read FIFO interface
- Arbitrating among the different DMA write commands of those channels
- Stopping any AXI data transfer any time without locking the AXI write system bus interface

The DMA_WRITE_CH_CORE stores and sends all write commands to the S_MEM. As soon as a write command is granted, the required data is fetched from the read FIFO interface of the corresponding channel and is written to the AXI write system bus interface.

A classic read FIFO interface is provided at the block interface to avoid embedded data FIFOs. This kind of implementation allows to scale the data FIFO assigned to any DMA write channel without having to modify the DMA controller. Only the level of the FIFO to be read must be provided to ensure a proper handshaking. The read FIFO interface is defined as a 32bit data bus width with a read enabled and a FIFO level to ensure enough data are present in the FIFO to perform a new burst.

Once a command is received from a DMA write channel, the arbitration process takes care of the right command to execute.

Any write command selected by the arbiter must only be issued by a sub-module if all the relevant data of the burst are present in the local FIFO of the sub-module.

As long as the DMA FIFO level is not empty, AXI write commands will be issued according to the write outstanding value set in the **AXI_PARAMS.AW_MAX_PEND[1:0]** bit register.

It is not allowed to insert wait state in between data read from the FIFO interface.

1.4.5.4.2.2 DMA_READ_CH_CORE:

This block is in charge of:

- Reading data from the S_MEM and to have those transfers compliant to the AXI4 AMBA protocol
- Providing the appropriate read burst length for a maximum system bus efficiency according to the number of data to be fetched

- Writing the relevant amount of data to a defined DMA read channel through the write FIFO interface
- Arbitrating among the different DMA read command of those channels
- Stopping any AXI data transfer any time without locking the AXI read system bus interface

The DMA_READ_CH_CORE stores and sends all read commands to the S_MEM. As soon as a read command is granted, the required data is fetched from the AXI write system bus interface and is written to the read FIFO interface of the corresponding channel.

A classic write FIFO interface is provided at the block interface to avoid embedded data FIFOs. This kind of implementation allows to scale the data FIFO assigned to any DMA read channel without having to modify the DMA controller. Only the level of the FIFO to be written must be provided to ensure a proper handshaking. The write FIFO interface is defined as a 32bit data bus width with a write enabled and a FIFO level to ensure enough storage is present in the FIFO to receive a new burst.

A read command from the DMA read channel would need to define all the relevant information to describe the read data transfer to be executed.

Once a command is received from a DMA read channel, the arbitration process will take care of the right command to be executed.

As long as the DMA FIFO level is not full, AXI read commands will be issued according to the read outstanding value set in the **AXI_PARAMS.AR_MAX_PEND[1:0]** bit register.

It is not allowed to insert wait state in between data written to the FIFO interface.

1.4.5.4.3 Data Transfer Mode

Several data transfer type can be defined:
No Transfer: When the register **AXI_PARAMS.AW_MAX_PEND[1:0]** is set to 0, no AXI write transfer is executed. There is the option to have the MH fully active and running without the need of an external memory to receive RX messages. Acknowledges will not be written, so this mode is considered for debug purpose only. When the **AXI_PARAMS.AR_MAX_PEND[1:0]** is set to 0, no read access is performed and without TX/RX descriptor read, the MH will be waiting forever.

The **AXI_PARAMS.AW_MAX_PEND[1:0]** and **AXI_PARAMS.AR_MAX_PEND[1:0]** can set the maximum number of read/write outstanding commands on the *DMA_AXI* interface.

1.4.5.4.4 Data Transfer Description

1.4.5.4.4.1 Address bus

The DMA is able to address up to 4Gbyte memory space (*DMA_AXI_AWADDR[31:0]* and *DMA_AXI_ARADDR[31:0]*) but in order to support SoC with bus addresses higher than 32bit, the **AXI_ADD_EXT** register can be used to extend the AXI address up to 64bit. The AXI addresses for read and write transaction is then build as:

- *DMA_AXI_AWADDR[63:0]* = **AXI_ADD_EXT[31:0]** & address from embedded MH DMA engine (32bit)
- *DMA_AXI_ARADDR[63:0]* = **AXI_ADD_EXT[31:0]** & address from embedded MH DMA engine (32bit)

1.4.5.4.4.2 Burst size

The maximum number of bytes to transfer in each data transfer is fixed and set to 4. Any read or write transfer always uses 32bit.

When considering TX message for instance, the payload data being defined as byte must be 4byte aligned when read from the S_MEM.

For the RX message, if data to be written to S_MEM is not properly aligned (CAN frames are byte aligned) padding bytes are added to complete the last word (4byte). The padding bytes are set to 0x00.

As a consequence, the write strobe signals are not managed by the DMA CONTROLLER as all 4 bytes are always written.

1.4.5.4.4.3 Burst length

The DMA CONTROLLER for the AXI read and write transfers supports INCR burst lengths from 1 to 8, considering an AXI 32bit data bus width. The DMA_AXI_AWLEN[3:0] and DMA_AXI_ARLEN[3:0] are sized to support a maximum burst length of 16 despite only 8 is possible. To be fully compliant with the AXI4 AMBA protocol [5] the DMA_AXI_AWLEN[7:4] and DMA_AXI_ARLEN[7:4] are considered as 0b0000.

The DMA Controller will always try to align its burst address to make full benefit of the maximum allowed burst length. The address burst value must always be 32byte aligned to ensure the maximum burst length (8x32bit). Whatever the data transfer mode, the DMA engine will reduce (if needed) the size of the first burst to align the address to the maximum burst length. Depending on the amount of data to be transferred, the last burst can be shorter.

It is important to optimize the access to the S_MEM, especially if a low number of data transfers is performed. As an example, if a data transfer of 12x32bit needs to be executed and the start address is 32byte aligned, it will result in two burst 8x32bit and 4x32bit. In case the start address is not aligned, and the worst scenario is assumed, this can lead to 3 bursts 3x32bit and 8x32bit and 1x32bit or 2x32bit and 8x32bit and 2x32bit or 1x32bit and 8x32bit and 3x32bit.

In case a high latency is expected in the SoC, it is essential to limit the number of burst and to make sure that, whenever it is possible, to align the start address to the maximum burst size.

The DMA CONTROLLER provides a variable burst length of data, according to the sub-module command requests.
The burst lengths from/to sub-modules connected to the DMA CONTROLLER are defined based on the data type of information to be used.

Here below are the expected burst lengths from/to the sub-modules:

- TX MESSAGE HANDLER: This sub-module does read the TX payload data from the S_MEM through the DMA read channel 2. The maximum burst length is limited to 8x32bit. There is no write access from this sub-module.

- RX MESSAGE HANDLER: This sub-module writes the RX message data to the S_MEM through the DMA write channel 1. The maximum burst length is limited to 8x32bit. There is no read access from this sub-module.

- DESCRIPTOR MESSAGE HANDLER: This sub-module performs a fixed burst read of 8x32bit to read TX descriptors from the S_MEM through the DMA read channel 2. A fixed burst length of 2x32bit is used instead to read RX descriptors through the DMA read channel 0. To acknowledge any transfer from and to the CAN bus, a fixed burst length of 4x32bit is written back to either the RX descriptor for RX message or to the TX descriptor for TX message.

1.4.5.4.4.4 Outstanding

In order to support read and write outstanding commands and to limit the FIFO size, the maximum burst length is limited to 8x32bit. The maximum outstanding expected at the *DMA_AXI* interface is programmable, see **AXI_PARAMS.AW_MAX_PEND[1:0]** and **AXI_PARAMS.AR_MAX_PEND[1:0]** bit field register. Up to 3 outstanding can be specified for read and write transactions. Even if set to the maximum value, the maximum number of outstanding performed by the MH will depend on many parameters like the system latency, the CAN Bus bit rate, the MH and PRT clock ratio, ...

1.4.5.4.4.5 Burst type

The only burst type supported is the burst incrementing INCR.
The WRAP/FIXED burst type is not supported.

1.4.5.4.4.6 Multi-region

The DMA controller AXI4 system bus interface does not support multiple region interfaces, see [5] for more details.

1.4.5.4.4.7 Memory attributes

The memory attributes for the read or write accesses to memory are Normal, Non-modifiable (Non-cacheable in AXI3) and Non-bufferable. No read-allocate nor No Write-allocate are expected on this interface and would be set to 0.
This means *DMA_AXI_AWCACHE[3:0]* and *DMA_AXI_ARCACHE[3:0]* are set to 0b0000.

As a reminder, Non-bufferable means (See [5] for more details):

- The write response must be obtained from the destination.
- Read data must be obtained from the destination.
- Transactions are Non-modifiable
- Read and write transactions from the same ID to addresses that overlap must remain ordered.

As a reminder, Non-modifiable means:

- A Non-modifiable transaction must not be split into multiple transactions or merged with other transactions.
- In a Non-modifiable transaction, the parameters AxADDR, AxSIZE, AxLEN, AxBURST and AxPROT must not be changed.

1.4.5.4.4.8 Access permissions

It is considered that any access is always defined as Data, Non-secure and the operating mode is Unprivileged, see [5] for more details. As a consequence, the DMA_AXI_ARPROT[1] and DMA_AXI_AWPROT[1] are set to 1. Those settings cannot be changed by SW. This means MEM_AXI_A(W/R)PROT[2:0] is set to 0b010.

1.4.5.4.4.9 Transaction ID

The DMA CONTROLLER generates the ID of every burst access based on the number of channels defined. It provides a way to track on the system bus which DMA channel is doing the access at any time.

For the AXI read interface, the *DMA_AXI_ARID[1:0]* defines the channel number as follow:
2'b00 => RX descriptor fetch from S_MEM
2'b01 => TX descriptor fetch from S_MEM
2'b10 => TX data payload read from S_MEM
For the AXI write interface, the *DMA_AXI_AWID[0]* defines the channel number as follow:
1'b0 => TX/RX descriptor acknowledge to S_MEM
1'b1 => RX message data write (payload and header) to S_MEM

1.4.5.5 TX Descriptor

TX descriptors are used for the TX FIFO Queues and the TX Priority Queue. They can be fetched with one AXI burst, as the overall size is only 8x32bit.
Many bit fields are common, but some are different between TX FIFO Queue and TX Priority Queue. Details are provided in the following table.
Further information is provided by the chapter TX Message Header Definition.

1.4.5.5.1 TX Priority Queue Descriptor Overview

Table: TX Priority Queue Descriptor Overview

TX PRIORITY QUEUE DESCRIPTOR	31	30	29	28	27	26	25	[24:16]	15	14	13	12	11	[10:9]	[8:4]	3	2	1	0
DMA Info Ctrl 1	VALID	**HD (set to 1) (Message Header)**	WRAP	**NEXT (set to 0)**	IRQ (Interrupt)	**PQ (set to 1)**	**Not Used (set to 0)**	CRC[8:0]	PQSN[4:0] (Priority Queue Slot Number)					**Not Used (set to 0)**	RC4[:0] (Rolling Counter)	STS[3:0] (TX Message Status)			
DMA Info Ctrl 2	**Not Used (set to 0)**					PLSRC	SIZE[9:0] (TX Buffer size)		IN[2:0] (Instance Number)			**Not Used (set to 0)**	**TDO[9:0] (set to 0)**						**Not Used (set to 0)**
TS0	TS0[31:0] (TimeStamp[31:0])																		

TS1	TS1[31:0] (TimeStamp[63:32])
T0	T0[31:0] (TX Message Header Information)
T1	T1[31:0] (TX Message Header Information)
T2 / TD0	T2[31:0] / TD0[31:0] (TX Message Header Information / First TX Data Payload)
TX_AP / TD1	TX_AP[31:0] / TD1[31:0] (TX Payload Data Address Pointer / Second TX Data Payload)

	Managed by SW and HW

1.4.5.5.2 TX FIFO Queue Descriptor overview

Table: TX FIFO Queue Descriptor Overview

TX FIFO QUEUE DESCRIPTOR	31	30	29	28	27	26	25	[24:16]	15	14	13	12	11	10	9	[8:4]	3	2	1	0
DMA Info Ctrl 1	VALID	**HD (set to 1) (Message Header)**	WRAP	**NEXT (set to 0)**	IRQ (Interrupt)	**PQ (set to 0)**	END	CRC[8:0]	FQN[3:0] (FIFO Queue Number)				**Not Used (set to 0)**		**Not Used (set to 0)**	RC4[:0] (Rolling Counter)	STS[3:0] (TX Message Status)			
DMA Info Ctrl 2	**Not Used (set to 0)**					PLSRC	SIZE[9:0] (TX Buffer size)		IN[2:0] (Instance Number)			**Not Used (set to 0)**	**NHDO[9:0] (set to 1)**							**Not Used (set to 0)**
TS0	TS0[31:0] (TimeStamp[31:0])																			
TS1	TS1[31:0] (TimeStamp[63:32])																			
T0	T0[31:0] (TX Message Header Information)																			
T1	T1[31:0] (TX Message Header Information)																			

T2 / TD0	T2[31:0] / TD0[31:0] (TX Message Header Information / First TX Data Payload)
TX_AP / TD1	TX_AP[31:0] / TD1[31:0] (TX Payload Data Address Pointer / Second TX Data Payload)

	Managed by SW and HW

1.4.5.5.3 TX Descriptor Description

Element Number	Bit field	Name	Managed by	Description/Constraints
0	[31]	VALID	SW/MH	Valid: The SW must set this bit to 1 to define a TX descriptor is valid for the MH. When the descriptor has been fully used, the MH will clear this bit when writing the acknowledge data information back to this descriptor. This update occurs only when the HD bit is set to 1. In case the descriptor is fetched when this bit is set to 0, an interrupt *TX_FQ_IRQ* is triggered to the system for the TX FIFO queue n having this descriptor.
	[30]	HD	SW only	Must be set to 1
	[29]	WRAP	SW only	Wrap: When set to 1 the next message descriptor is the one declared at the initial start address of the TX FIFO Queue (First Descriptor). This bit provides a way to the SW to keep the next TX message continuous in a memory buffer if less space is available at the end of a data container
	[28]	NEXT	SW only	Must be set to 0
	[27]	IRQ	SW only	Interrupt: when set to 1 an interrupt is triggered to the system when the descriptor execution is complete, meaning when the TX message has been sent to the CAN bus
	[26]	PQ	SW only	TX Priority Queue: when set to 1, the TX descriptor belongs to the TX Priority Queue TX FIFO Queue: must be set to 0
	[25]	END	SW only	For the TX FIFO Queue: when set to 1 the TX FIFO Queue defined is ending, it means, it is set as inactive. Once done, the TX FIFO Queue can be reprogrammed and started

Element Number	Bit field	Name	Managed by	Description/Constraints
				For the TX Priority Queue: must be set to 0
	[24:16]	CRC[8:0]	SW only	CRC: this CRC is computed by the SW for the current TX descriptor. It must consider all elements assuming this bit field as set to 0. Any CRC error is triggering an interrupt to the system. The CRC is not evaluated if the MH_SFTY_CTRL.TX_DESC_CRC_EN bit is set to 0.
	[15:12]	FQN[3:0]	SW only	TX FIFO Queue: define the TX FIFO Queue number allocated to this TX descriptor. Despite being set to 4bit, only the FQN[2:0] bit range is used
		PQSN[4:1]	SW only	TX Priority Queue: define the TX FIFO Queue slot number allocate to this descriptor
	[11]	reserved	SW only	TX FIFO Queue: must be set to 0
		PQSN[0]	SW only	TX Priority Queue: define the TX FIFO Queue slot number allocate to this descriptor
	[10:9]	Not used	SW only	Must be set to 0
	[8:4]	RC[4:0]	SW only	Rolling Counter: use to track the order of TX descriptor fetched when a TX FIFO Queue or a TX Priority Queue slot is running. TX FIFO Queue: The first TX descriptor in a TX FIFO Queue must have the RC[4:0] set to 5'b00000 (before first start). This value must be incremented for every new TX descriptor up to 5'b11111 and then back to 5'b00000, and so on. If a TX FIFO Queue is circular, meaning the FIFO restarts at the first TX descriptor, the RC[4:0] must be updated accordingly based on the RC[4:0] defined and executed in the last TX descriptor of the TX FIFO Queue. TX Priority Queue: This bit field must be set to 5'b00000 as the default value for the TX Header descriptor defined in the slot.
	[3:0]	STS[3:0]	MH only	Status: gives the status of the TX message transmitted. The MH writes back only the Header Descriptor (HD bit set to 1) for status report. The SW must always set it to 0 0'b0000: none 0'b0001: message sent successfully 0'b0010: message not sent after a number of trials 0'b0011: message skipped due to HFI 0'b0100: message rejected by TX filter 0'b0101: reserved

Element Number	Bit field	Name	Managed by	Description/Constraints
				0’b0110: reserved 0’b0111: reserved 0’b1000: reserved 0’b1001: reserved 0’b1010: reserved 0’b1011: reserved 0’b1100: reserved 0’b1101: reserved 0’b1110: reserved 0’b1111: message acknowledge data with parity error
1	[31:27]	Not Used	SW only	Must be set to 0
	[26]	PLSRC	SW only	Payload Source: This bit provides to the MH the information about the need to fetch payload data in the data container when executing only a TX Header Descriptor. When set to 1: the TX descriptor is attached to a data container which would need to be accessed and the bit field SIZE[9:0] defines the number of TX data to send for this descriptor. For CAN XL, as no payload data can be defined in TX descriptor, this bit is always set to 1 for CAN XL. For CAN FD, this bit is set to 1 when the payload data is greater than 4bytes. When set to 0: the payload data defined in the data container are not required. Therefore, the TX descriptor includes all data payload. For the Classical CAN, all payload data are always included, this bit must always be set to 0. In case of CAN FD, it would be set to 0 only when the payload data is less or equal to 4bytes. Nevertheless, the bit field SIZE[9:0] still defines the number of payload data to send per TX descriptor
	[25:16]	SIZE[9:0]	SW only	Define the buffer size in word (32bit) for the given TX descriptor to transmit to the PRT. As an example, a payload from 1 to 4 bytes requires SIZE to be set to 1. As only 32bit read accesses are performed the buffer size containing the payload must be 32bit aligned. When set to 0, there is no payload data attached to the TX descriptor (only valid for Classical CAN/CAN FD without payload or a Classical CAN remote frame) For CAN XL no data is defined in TX descriptor. The MH replies only on the address pointer defined in element 7 to fetch payload data from S_MEM. For CAN FD:

Element Number	Bit field	Name	Managed by	Description/Constraints
				• SIZE > 1: The copy of the first data payload (aligned on 32bit) is required in element 6. The address pointer in element 7 is used to fetch the payload data from S_MEM. • SIZE = 1: The copy of the first data payload (aligned on 32bit) is required in element 6. In case it is less than 4bytes, just pad with 0s. The address pointer in element 7 is not used. Nevertheless, it is required to have it set to the address of the payload data in S_MEM • SIZE=0: Elements 7 and 6 are not used •
	[15:13]	IN[2:0]	SW only	Instance Number: define the X_CAN instance number using that descriptor. This bit field is relevant if several X_CAN are running concurrently. It provides a way to detect descriptor fetch issue between instances. The value defined must be equal to the one defined in the MH_CFG.INST_NUM bit field register.
	[12]	Not Used	SW only	Must be set to 0
	[11:2]	TDO[9:0]	SW only	For the TX Priority Queue: must be set to 0.
		NHDO[9:0]	SW only	For the TX FIFO Queue: must be set to 1.
	[2:0]	Not used	SW only	Must be set to 0
2	[31:0]	TS0[31:0]	MH only	Timestamp 0: LSB of the 64bits timestamp of the successfully sent TX message (only valid when HD bit is set to 1)"
3	[31:0]	TS1[31:0]	MH only	Timestamp 1: MSB of the 64bits timestamp of the successfully sent TX message (only valid when HD bit is set to 1)"
4	[31:0]	T0[31:0]	SW only	Define the TX message header information, see TX message header definition chapter
5	[31:0]	T1[31:0]	SW only	Define the TX message header information, see TX message header definition chapter
6	[31:0]	TD0[31:0]	SW only	Classical CAN and CAN FD: define the first payload of the TX message
		T2[31:0]	SW only	CAN XL: Defined the TX message header information, see TX message header definition chapter
7	[31:0]	TD1[31:0]	SW only	Classical CAN with payload greater equal to 4byte: define the last payload data of the TX message for the Classical CAN (in case payload data is greater than 4bytes).
		TX_AP[31:0]	SW only	CAN XL and CAN FD (with payload greater than 4bytes): Address pointer to fetch the TX message payload data

Element Number	Bit field	Name	Managed by	Description/Constraints
				for CAN FD and CAN XL frames. For CAN FD frames with more than 4 bytes this bit field is, nevertheless, mandatory. As the address pointer must be 32bit aligned the two LSB will not be considered and so must be set to 0 all time. In case the TX_AP is not used it must be set to 0

Table: TX Descriptor description

Here is the list of the required elements for the various TX descriptor definitions to be managed by the SW or the MH:

	SW to write information to MH	SW to read information from MH
Element Number	Header Descriptor	Header Descriptor
0	Mandatory	Mandatory
1	Mandatory	Mandatory
2	NA	Mandatory
3	NA	Mandatory
4	Mandatory	NA
5	Mandatory	NA
6	Mandatory	NA
7	Optional	NA

Table: TX Descriptor Element managed by SW

	MH to write information to SW	MH to read information from SW
Element Number	Header Descriptor	Header Descriptor
0	Mandatory	Mandatory
1	Mandatory	Mandatory
2	Mandatory	NA
3	Mandatory	NA
4	NA	Mandatory
5	NA	Mandatory
6	NA	Mandatory
7	NA	Optional

Table: TX Descriptor Element Managed by MH

1.4.5.5.4 TX Descriptor CRC Computation

A dedicated CRC is computed for every TX descriptor. When a CRC error is detected, the *DESC_ERR* interrupt signal is triggered. This way, the data transfer setting and description, up to the DMA engine, are fully protected.

The CRC covers all the relevant data, meaning the 247bit data (8*32bit - 9) in the TX descriptor considering the CRC bit field in the descriptor as equal to 0b000000000. The CRC is part of the Element Number 0.

The CRC (CRC-9_167) is computed assuming the following elements in sequence:
Element Number 0[31:25] & 0b000000000 & Element Number 0[15:0]
Element Number 1[31:0]
Element Number 2[31:0] set to 32'b0
Element Number 3[31:0] set to 32'b0
Element Number 4[31:0]
Element Number 5[31:0]
Element Number 6[31:0]
Element Number 7[31:0]
The Koopman representation of the polynomial CRC-9_167 is used to protect TX descriptors:
CRC-9_167 = ($x^9 + x^7 + x^6 + x^3 + x^2 + x + 1$) ⁑ (CRC polynomial in implicit "+1" hex format, meaning the trailing "+1" is omitted from the polynomial number)

Using the **MH_SFTY_CTRL.TX_DESC_CRC_EN** bit register, the SW can decide to disable this check for all the TX descriptors fetched from S_MEM or L_MEM.

Here below is the pseudo code to compute the CRC for a TX/RX descriptor:
The word_table[] is the array of 32bit element defined previously (in the order they are listed):

```
static bit[8:0]   rem9      = 9'h1FF;
static bit[8:0]   rem9_old  = 9'h1FF;
static bit[8:0]   poly      = 9'h167;
static bit[8:0]   crc9;

// This algorithm is indirect
// initialize CRC shift register

rem9 = 9'h1FF;
foreach (word_table[i]) begin
   for (int j = 31; j >= 0; j--) begin
```

```
      // to decide whether reduction with polynomial will be required based on MSB before shift
      rem9_old = rem9;

      // shift out MSB of CRC
      rem9 = rem9 << 1;
      rem9[0] = word_table[i][j];

      // perform reduction if required
      if (rem9_old[8]) rem9 = rem9 ^ poly;
    end
end

// processing 9 0s more
repeat(9) begin
    // to decide whether reduction with polynomial will be required based on MSB before shift
    rem9_old = rem9;

    // shift out MSB of CRC
    rem9 = rem9 << 1;
    rem9[0] = 0;

    // perform reduction if required
    if (rem9_old[8]) rem9 = rem9 ^ poly;
end
crc9 = rem9;
```

1.4.5.5.5 TX Descriptor Errors

When a TX descriptor error is detected, the relevant information is logged in the **DESC_ERR_INFO1** register. Furthermore, the source address of the faulty TX descriptor is logged in the **DESC_ERR_INFO0** register. This would help the SW to identify potential root causes when such error occurs. The **DESC_ERR_INFO1.RX_TX** bit register is set to 0 when a TX descriptor gets an error.

1.4.5.6 TX Message Header Definition

The TX descriptor contains i.e., the TX message header. The header data structure depends on the CAN Frame Format (Classical CAN, CAN FD, CAN, XL) to be used for this message on the CAN Bus. It can be controlled by the header bits **R0.FDF**, **R0.XLF** and **R0.XTD**. The following tables describe the three data structures used for the headers.

Table: Classical CAN TX Header definition

Tn	Bits	Name	Description/Constraints
T0	[31]	FDF	FD Format
	[30]	XLF	XL Format
	[29]	XTD	Extended Identifier
	[28:18]	BaseID [28:18]	Base ID
	[17:0]	ExtID [17:0]	Extended ID
T1	[31]	Reserved	Not Applicable
	[30]	FIR	Fault Injection Request
	[29:27]	Reserved	Not Applicable
	[26]	RTR	Remote Transmission Request
	[25:20]	Reserved	Not Applicable
	[19:16]	DLC[3:0]	Data Length Code
	[15:0]	Reserved	Not Applicable

Note: Classical CAN frames (CBDF, CEDF, CBRF, CERF) require **T0.FDF** = **0** and **T0.XLF** = 0. The header consists of T0 and T1.

Table: CAN FD TX Header definition

Tn	Bits	Name	Description/Constraints
T0	[31]	FDF	FD Format
	[30]	XLF	XL Format
	[29]	XTD	Extended Identifier
	[28:18]	BaseID [28:18]	Base ID
	[17:0]	ExtID [17:0]	Extended ID
T1	[31]	Reserved	Not Applicable
	[30]	FIR	Fault Injection Request
	[29:27]	Reserved	Not Applicable
	[26]	Must be set to 0	Not Applicable
	[25]	BRS	Bit Rate Switch
	[24:21]	Reserved	Not Applicable
	[20]	ESI	Error State Indicator
	[19:16]	DLC[3:0]	Data Length Code
	[15:0]	Reserved	Not Applicable

Note: CAN FD frames (FBDF, FEDF) require **T0.FDF** = 1 and **T0.XLF** = 0. The header consists of T0 and T1.

Table: CAN XL TX Header definition

Tn	Bits	Name	Description/Constraints
T0	[31]	FDF	FD Format
	[30]	XLF	XL Format

Tn	Bits	Name	Description/Constraints
	[29]	XTD	Extended Identifier
	[28:18]	Priority ID[28:18]	Priority identifier
	[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
	[29:27]	Reserved	Not Applicable
	[26:16]	DLC-XL[10:0]	Data Length Code with CAN XL encoding
	[15:0]	Reserved	Not Applicable
T2	[31:0]	AF[31:0]	Acceptance Field

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

1.4.5.7 RX Descriptor

The RX descriptor definition for the RX FIFO is defined in table below. Only 4x32bit are required to define an RX descriptor. As a matter of fact, the overall RX descriptor can be fetched with one burst. Some bit field elements are defined in a separate table for the sake of simplicity.

1.4.5.7.1 RX FIFO Queue Descriptor Overview (Normal Mode)

Table: RX FIFO Queue Descriptor Overview (Normal Mode)

RX FIFO QUEUE DESCRIPTOR (Normal Mode)	**31**	**30**	**29**	**28**	**27**	**26**	**25**	**[24:16]**	**[15:12]**	**[11:9]**	**[8:4]**	**[3:0]**
DMA info Ctrl 1	VALID	HD (Message Header)	**Not Used (set to 0)**	NEXT	IRQ	**Not Used (set to 0)**		CRC[8:0]	FQN[3:0] (RX FIFO Queue Number)	IN[2:0] (Instance Number)	RC[4:0] (Rolling Counter)	STS[3:0] (TX Message Status)
RX_AP	RX_AP[31:0] (RX Address Pointer)											
TS0	TS0[31:0] (TimeStamp[31:0])											

TS1	TS1[31:0] (TimeStamp[63:32])

	Managed by SW and HW

1.4.5.7.2 RX FIFO Queue Descriptor Overview (Continuous Mode)

Table: RX FIFO Queue Descriptor Overview (Continuous Mode)

RX FIFO QUEUE DESCRIPTOR (Normal Mode)	31	30	29	28	27	26	25	[24:16]	[15:12]	[11:9]	[8:4]	[3:0]
DMA info Ctrl 1	VALID	HD (Message Header) (always set to 1)	Not Used (set to 0)	NEXT (always set to 0)	IRQ	Not Used (set to 0)		CRC[8:0]	FQN[3:0] (RX FIFO Queue Number)	IN[2:0] (Instance Number)	RC[4:0] (Rolling Counter)	STS[3:0] (TX Message Status)
RX_AP	RX_AP[31:0] (RX Address Pointer)											
TS0	TS0[31:0] (TimeStamp[31:0])											
TS1	TS1[31:0] (TimeStamp[63:32])											

	Managed by SW and HW

1.4.5.7.3 RX Descriptor Description

Element Number	Bit field	Name	Managed by	Description/Constraints
0	[31]	VALID	SW/MH	Valid: The SW must set this bit to 0 to define a RX descriptor is pointing to a valid data container. As soon as the RX descriptor is executed the MH will set this bit to 1 to indicate to the SW valid data written to the S_MEM. In case the RX descriptor is fetched with this bit set to 1 and interrupt *RX_FQ_IRQ* is triggered to the system for the RX FIFO Queue having this non valid descriptor. The SW must clear this bit only when all the RX message

Element Number	Bit field	Name	Managed by	Description/Constraints
				data attached have been read
	[30]	HD	MH only	Message header: when set to 1 the RX descriptor is defined as containing the header of the RX message. Any other RX descriptor, if several descriptors are used for the same RX message, will contain only payload data. In Continuous Mode HD is always set to 1 as only one RX descriptor is used per RX message
	[29]	reserved	SW only	Must be set to 0
	[28]	NEXT	MH only	Next: Set to 1 by the MH to indicate in the RX Header descriptor that more than one descriptor is used for the RX message. This information is only mentioned in the Header Descriptor, the RX Trailing Descriptors are not modified. This allows the SW to acknowledge only the RX Header Descriptor for any RX messages. In Continuous Mode NEXT is always set to 0 as only one RX descriptor is used per RX message
	[27]	IRQ	SW only	Interrupt: when set to 1, an interrupt is triggered to the system when the descriptor execution is complete and a correctly received RX message was written to it. This interrupt can provide to the SW, a synchronization point to monitor the RX FIFO Queue execution
	[26:25]	Not Used	SW only	Must be set to 0
	[24:16]	CRC[8:0]	SW only	CRC: this CRC is computed by the SW for the current RX descriptor. It must consider all elements assuming this bit field as set to 0. Any CRC error is triggering an interrupt to the system. The CRC is not evaluated if the **MH_STS.RX_DESC_CRC_EN** bit is set to 0.
	[15:12]	FQN[3:0]	SW only	RX FIFO Queue number: define the RX FIFO Queue number allocated to this RX descriptor
	[11:9]	IN[2:0]	SW only	Instance Number: define the X_CAN instance number using that descriptor. This bit field is relevant if several X_CAN are running concurrently. It provides a way to detect descriptor fetch issue between instances. The value defined must be equal to the one defined in the **MH_CFG.INST_NUM** bit field register.
	[8:4]	RC[4:0]	SW only	Rolling Counter: use to track the order of RX descriptor fetched when a RX FIFO Queue is running. When a RX FIFO Queue is started for the first time, its First RX descriptor must have the RC[4:0] set to 5'b00000. This value must be incremented for every new RX descriptor

Element Number	Bit field	Name	Managed by	Description/Constraints
				up to 5'b11111 and then back to 5'b00000 and so on. Even if a wrap occurs at the end of the RX FIFO Queue (circular RX FIFO Queue), the first RX descriptor of that FIFO must be updated with the correct RC[4:0] value. Thus, the First RX descriptor RC[4:0] value needs to be updated by incrementing the value defined in the previous descriptor. To always have RC[4:0] = 5'b00000 for the First RX descriptor (in case of circular RX FIFO Queue), the RX FIFO Queue size must be a multiple of 32 RX descriptor
	[3:0]	STS[3:0]	MH only	**Status:** gives the status of the RX message received. This bit field is written back by the MH when the descriptor has been completed. This bit field must be set to 0 by SW. 0'b0000: none 0'b0001: message received successfully 0'b0010: message received but not filtered 0'b0011: reserved 0'b0100: reserved 0'b0101: reserved 0'b0110: reserved 0'b0111: reserved 0'b1000: reserved 0'b1001: reserved 0'b1010: reserved 0'b1011: reserved 0'b1100: reserved 0'b1101: reserved 0'b1110: reserved 0'b1111: message acknowledge data with parity error
1	[31:0]	RX_AP	SW/MH	**Normal Mode:** the SW defines the address of the RX data container to write RX data **Continuous Mode:** The SW must set this bit field to 0 as default value. The MH writes this field with the address pointer to find the RX message attached to the RX descriptor. Only the RX Header Descriptor is having this bit field updated, with the RX message address in the data container. This address must be 32bit aligned, the two LSB bits are assumed to be always 0
2	[31:0]	TS0[31:0]	MH only	**Timestamp 0:** LSB of the 64bits timestamp of the

Element Number	Bit field	Name	Managed by	Description/Constraints
				successfully received RX message (only valid when HD bit is set to 1)"
3	[31:0]	TS1[31:0]	MH only	**Timestamp 1:** MSB of the 64bits timestamp of the successfully received RX message (only valid when HD bit is set to 1)"

Table: RX Descriptor description

Here is the list of the required elements for the various RX descriptor definitions to be managed by the SW or the MH:

	SW to write information to MH	SW to read information from MH	
Element Number	RX Descriptor	Header Descriptor	Trailing Descriptor
0	Mandatory	Mandatory	Mandatory in Normal mode
1	Mandatory in Normal mode NA in Continuous mode (must be set to 0)	Mandatory	Mandatory in Normal mode
2	NA (must be set to 0)	Mandatory	NA (must be equal to 0)
3	NA (must be set to 0)	Mandatory	NA (must be equal to 0)

Table: Element managed by SW

	MH to write information to SW		MH to read information from SW
Element Number	Header Descriptor	Trailing Descriptor	RX Descriptor
0	Mandatory	Not updated	Mandatory
1	Not updated in Normal mode Mandatory in Continuous mode	Not updated	Mandatory in Normal mode NA in Continuous mode
2	Mandatory	Not updated	NA
3	Mandatory	Not updated	NA

Table: Element managed by MH

When the Element Number is mentioned as NA, the assumed default value must be 0.

1.4.5.7.4 CRC Computation

A dedicated CRC is computed for every RX descriptor. When a CRC error is detected, the *DESC_ERR* interrupt signal is triggered. This way, the data transfer setting and description, up to the DMA engine, are fully protected.

The CRC covers all the relevant data, meaning the 55bit data in the RX descriptor considering the CRC bit field in the descriptor as equal to 0b000000000. The CRC is part of the Element Number 0.

The CRC (CRC-9_167) is computed assuming the following elements in sequence:
Element Number 0[31:25] & 0b000000000 & Element Number 0[15:0]
Element Number 1[31:0]
The Koopman representation of the polynomial CRC-9_167 is used to protect RX descriptors:
CRC-9_167 = $(x^9 + x^7 + x^6 + x^3 + x^2 + x + 1)$ * (CRC polynomial in implicit "+1" hex format, meaning the trailing "+1" is omitted from the polynomial number)

Using the **MH_SFTY_CTRL.RX_DESC_CRC_EN** bit register, the SW can decide to disable this check for all the TX descriptors fetched from S_MEM.

The Pseudo code of indirect CRC algorithm is available in TX Descriptor chapter under CRC computation section.

1.4.5.7.5 RX Descriptor Errors

When a RX descriptor error is detected, the relevant information is logged in the **DESC_ERR_INFO1** register. Furthermore, the source address of the faulty RX descriptor is logged in the **DESC_ERR_INFO0** register. This would help the SW to identify potential root causes when such error occurs. The **DESC_ERR_INFO1.RX_TX** bit register is set to 1 when a RX descriptor gets an error.

1.4.5.8 RX Message Header Definition

Messages received from the CAN Bus are stored in the S_MEM, each consisting of a header followed by the payload. The header data structure depends on the CAN Frame Format (Classical CAN, CAN FD, CAN, XL) used for this message on the CAN Bus. It can be identified by the header bits **FDF** and **XLF**. The following tables describe the three data structures used for the headers, consisting of the words R0, R1 and R2.

Rn	Bits	Name	Source	Description/Constraints
R0	[31]	FDF	CAN	FD Format
	[30]	XLF	CAN	XL Format
	[29]	XTD	CAN	Extended Identifier
	[28:18]	BaseID [28:18]	CAN	Base ID
	[17:0]	ExtID [17:0]	CAN	Extended ID
R1	[31:27]	na	na	reserved
	[26]	RTR	CAN	Remote Transmission Request

Rn	Bits	Name	Source	Description/Constraints
	[25:20]	na	na	reserved
	[19:16]	DLC[3:0]	CAN	Data Length Code
	[15:11]	na	na	reserved
	[10]	FAB	MH	Filter Aborted: when set to 1, the RX filtering process was ending before completing with no match
	[9]	BLK	MH	Black List: When set to 1, the RX message filtered belongs to a blacklist
	[8]	FM	MH	Filter Match: When set to 1 one of the filter elements (defined by FIDX[7:0]) has detected a match
	[7:0]	FIDX[7:0]	MH	Filter index: provide the information of the filter index which has been triggered
R2	[31:0]	na	na	reserved

Table: Classical CAN RX Header definition
Note: Classical CAN frames (CBDF, CEDF, CBRF, CERF) can be identified by **R0.FDF** = 0 and **R0.XLF** = 0.

Rn	Bits	Name	Source	Description/Constraints
R0	[31]	FDF	CAN	FD Format
	[30]	XLF	CAN	XL Format
	[29]	XTD	CAN	Extended Identifier
	[28:18]	BaseID [28:18]	CAN	Base ID
	[17:0]	ExtID [17:0]	CAN	Extended ID
R1	[31:26]	na	na	reserved
	[25]	BRS	CAN	Bit Rate Switch
	[24:21]	na	na	reserved
	20	ESI	CAN	Error State Indicator
	[19:16]	DLC[3:0]	CAN	Data Length Code
	[15:11]	na	na	reserved
	[10]	FAB	MH	Filter Aborted: when set to 1, the RX filtering process was ending before completing with no match
	[9]	BLK	MH	Black List: When set to 1, the RX message filtered belongs to a blacklist
	[8]	FM	MH	Filter Match: When set to 1 one of the filter elements (defined by FIDX[7:0]) has detected a match
	[7:0]	FIDX[7:0]	MH	Filter index: provide the information of the filter index which has been triggered
R2	[31:0]	na	na	reserved

Table: CAN FD RX Header definition
Note: CAN FD frames (FBDF, FEDF) can be identified by **R0.FDF** = 1 and **R0.XLF** = 0.

Rn	Bits	Name	Source	Description/Constraints
R0	[31]	FDF	CAN	FD Format
	[30]	XLF	CAN	XL Format
	[29]	na	na	reserved
	[28:18]	Priority ID[28:18]	CAN	Priority identifier
	[17]	RRS	CAN	Remote Request Substitution
	[16]	SEC	CAN	Simple Extended Content
	[15:8]	VCID[7:0]	CAN	Virtual CAN Network ID
	[7:0]	SDT[7:0]	CAN	SDU Type
R1	[31:27]	na	na	reserved
	[26:16]	DLC-XL[10:0]	CAN	Data Length Code with CAN XL encoding
	[15:11]	na	na	reserved
	[10]	FAB	MH	Filter Aborted: when set to 1, the RX filtering process was ending before completing with no match
	[9]	BLK	MH	Black List: When set to 1, the RX message filtered belongs to a blacklist
	[8]	FM	MH	Filter Match: When set to 1 one of the filter elements (defined by FIDX[7:0]) has detected a match
	[7:0]	FIDX[7:0]	MH	Filter index: provide the information of the filter index which has been triggered
R2	[31:0]	AF[31:0]		Acceptance Field

Table: CAN XL RX Header definition
Note: CAN XL frames (XLFF) could be identified by **R0.FDF** = 1 and **R0.XLF** = 1.

1.4.5.9 TX Message

For a better understanding while reading this chapter, read the TX descriptor chapter first.
A TX message is defined using one TX descriptor and a TX data container where the payload data buffer is defined.

The Header Descriptor (or the only one, in case of one descriptor per message) holds the header data information and for some CAN protocols, the data payload of the message. Such descriptor also provides some additional information to the MH: the interrupt to be triggered, where to write acknowledge data, where to fetch TX message data, etc.

A TX data container is a general term to name the memory space allocated by the SW. This data container is used to hold the payload data buffer. In most of the cases, this TX data container would be identical to the data buffer size to transmit, avoiding the loss of memory space.

A specific TX descriptor is used for the TX FIFO Queue and for the TX Priority Queue due to the structure of the two different implementations.

In order to optimize the fetch of the TX descriptor as well as data payload, a maximum burst length of 8x32bit is used.

The buffer size which can be defined in a TX descriptor can go up to 2048byte. This way, a TX message can be defined using a single TX descriptor and one data buffer.

As the maximum efficiency is reached when using the maximum burst length, it is highly recommended to define a data buffer size aligned on the maximum burst length.

If the TX payload data is not a multiple of the burst length, the remaining data in the data container won't be read. Nevertheless, the embedded DMA controller will use the maximum burst length to read the payload whenever possible and will adapt the latest burst length to complete its transfer. Only the relevant data are read from S_MEM when smaller than the maximum burst length.

The address pointer used to fetch the payload data is always 32bit, despite that payload data is byte aligned.
Every TX descriptor holding the header of the TX message, once a TX message is transmitted, is acknowledged for status, error reporting and timestamping.

Here below are the different types of messages according to CAN protocols.

1.4.5.9.1 Single TX descriptor Usage

A TX message can be defined using one single TX descriptor. This kind of choice requires to have the complete payload data defined in one data container in the S_MEM. In case of Classical CAN, the complete Classical CAN message is embedded in the TX descriptor. This means no payload buffer is required for Classical CAN messages. The NEXT bit in TX descriptor must be set to 0. In case of a TX Priority Queue, the TX descriptor TDO bit field must be set to 0. For the TX FIFO Queues, the NHDO is set to a value equal to 1 in order to define the next TX header descriptor.

For the TX Priority Queue and TX FIFO Queue the same description below applies.
Classical CAN with up to 8byte payload
As the Classical CAN payload data is only 8byte, it can be defined completely in the TX descriptor (see TD0 and TD1). There is no need to define an address pointer to a payload buffer in that case. Despite a data container is mentioned, it is not used. This is to align with the other description in the next sections.

This approach provides a single and simple way to send any Classical CAN TX message in a straightforward manner. Using the T0, T1, TD0 and TD1 in the TX descriptor, the overall Classical CAN message can be defined, refer to the TX descriptor chapter for more details.

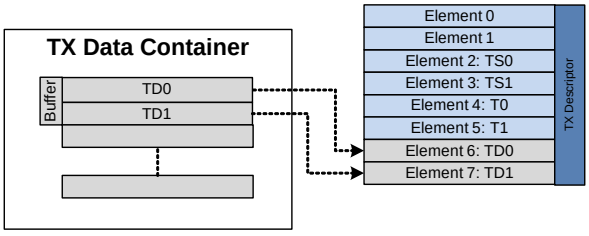

Figure: Classical CAN TX message with 8byte payload (single descriptor)

CAN FD

As the CAN FD protocol can provide up to 64byte, it is mandatory to define an address pointer to read the payload data from the S_MEM when the size is greater than 4byte. The first payload data defined in the payload data buffer also needs to be defined in the TX descriptor. For high latency system, the time to fetch the payload data, once the arbitration process is complete, can lead to a potential underrun. To solve this issue, TD0 is declared in the TX descriptor. By the time TD0 is sent through the CAN bus, the payload data will be read from the S_MEM. This approach avoids prefetching the payload data before having the arbitration result and to throw away the complete burst when arbitration is not successful. The TDO from the first read burst access will then be skipped.

The address pointer points to the buffer holding the overall payload data as depicted in figure below. In case only 4byte payload data is required, there would be no need to define the address pointer (must be set to 0). For payload data above 4byte an address pointer is required. The minimum data container size is either 32byte (data payload lower or equal to 32byte) or 64byte (data payload greater than 32byte).

The size of the buffer to be fetched is always 32bit aligned. When the data payload is lower than a multiple of 32bit, padding is expected and will be discarded by the PRT.

Using the T0, T1, TD0 and the TX_AP fields, the overall CAN FD TX message can be defined, refer to the TX descriptor chapter for more details.

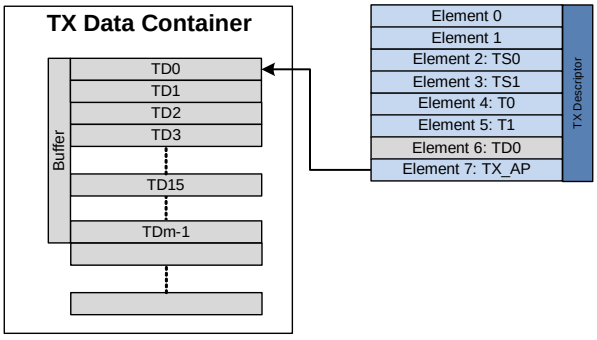

Figure: CAN FD TX message with more than 4byte payload (single descriptor)

CAN XL

As the CAN XL header information requires 3 words of 32bit, there is no payload data defined in the TX descriptor. T2 is required only when the arbitration on the CAN bus is successful, giving time for the MH to read the payload data from the S_MEM and to avoid the need of prefetching data.
Using the T0, T1, T2 and the TX_AP fields, the overall CAN XL TX message can be defined, refer to the TX descriptor chapter for more details.
The size of the buffer to be fetched is always 32bit aligned. When the data payload is lower than a multiple of 32bit, padding is expected and will be discarded by the PRT.

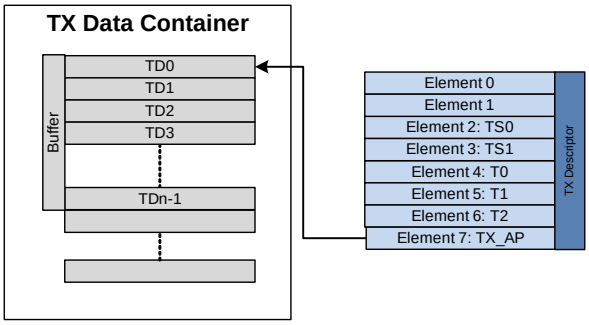

Figure: CAN XL TX message (single descriptor)

1.4.5.10 RX Message in Normal Mode

Prior to reading this chapter, read the RX descriptor chapter first.
In order to receive RX messages, an RX descriptor is required to define how the MH must behave and where to write the RX data in Normal mode.
Those RX descriptors are attached to RX FIFO Queues which are selected according to the RX filtering rules. It means, RX descriptors are concatenated and read in sequence.

An RX data container is a general term to name the memory space allocated by the SW. This data container is used to hold the RX message data. In most of the cases, this RX data container would not be fully filled with data, maximum data payload being different for CAN protocols.

Every RX descriptor is assigned to a data container to write incoming data to the S_MEM. The RX data container size is a multiple of the maximum burst lengths supported, 8x32bit with a maximum of 4064byte (127*32byte) and a minimum of 32byte. This granularity does provide some flexibility to address several RX messages sizes with only one data container. As defined previously, if an RX data container is smaller than an RX message, several RX descriptors will be assigned to that message.

Compared to the TX message, the header and the payload of the RX message are written together to the S_MEM. This approach gives the flexibility to pass address pointers of the overall message to the application and to avoid copies.

If the payload data does not cover a multiple of the burst length, some data won't be written to the data buffer in the container. The embedded DMA controller will use the maximum burst length

whenever possible to write header and payload and will adapt the latest burst length to complete its transfer.

The address pointer used to write the RX message is always 32bit aligned despite payload data is byte aligned.
The size of the data container defined into the RX descriptor is fixed for a given RX FIFO Queue and for all the RX descriptors of that queue. The smaller the size of data buffer the less RX descriptors a message would require.

As the data container is defined anywhere into the S_MEM, the SW can decide to allocate all the data containers into a continuous way in the S_MEM to ensure, the RX message is not split over several location. It will ease the reading of RX messages and simplify the management of data buffers, see RX FIFO Queue chapter for more details.

The NEXT bit defined into the RX Header Descriptor provides the information to the SW that one or several RX descriptors are used. On top of it, the RX Header Descriptor of an RX message will have the HD bit set to 1 to indicate that the data container got the header of the message.

Only the RX Header Descriptor holding the header data is acknowledged when an RX message is received. This way, despite receiving the timestamp at the end of the data received, it will be written with the header and status reporting.

Here below are the different types of RX messages according to the CAN protocol and some different structures when using one or several RX descriptors.

1.4.5.10.1 Single RX Descriptor

With this structure, the size of the data container defined by the RX descriptor must be large enough to hold the maximum payload size of the expected RX message to receive.

Classical CAN
As depicted in the figure below, the Classical CAN header and payload data can be directly written into a 32byte data container (N = 1). If such data buffer size is defined, then several RX descriptors would be required to support CAN FD (3 RX descriptors) or CAN XL (65 RX descriptors). It is important to note that, according to the RX message size received, it may be possible to have message data written in a bigger data container as every RX FIFO Queue defines its own data container size.

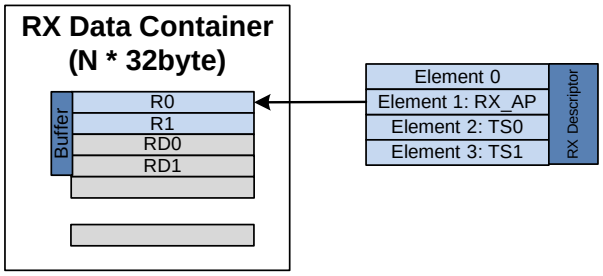

Figure: Classical CAN RX message (single descriptor)

CAN FD

Compared to the Classical CAN, a larger data buffer is required to hold up to 64byte of payload data and the header message data. In this case, a data container of 96byte (N = 3) is allocated to support CAN FD frame format. There will be no issue regarding Classical CAN message as it would fit entirely into the same data container. Doing so, the CAN XL message can be supported but would require up to 22 RX descriptors.

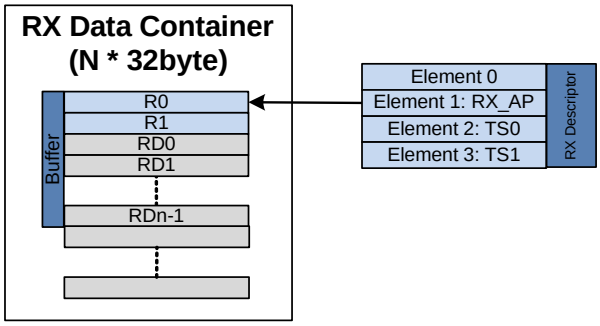

Figure: CAN FD RX message (single descriptor)

CAN XL

To ensure that only one RX descriptor points to one CAN XL RX message, it is possible to allocate a data container size of more than 2048byte (N=65). With this setting, all the different CAN protocols are covered with a single data container per RX descriptor. However, quite some memory space is lost in the data container (when configure to support CAN XL payload size) when receiving Classical CAN or CAN FD messages. To solve this issue, multiple RX descriptors can be used, see next chapter.

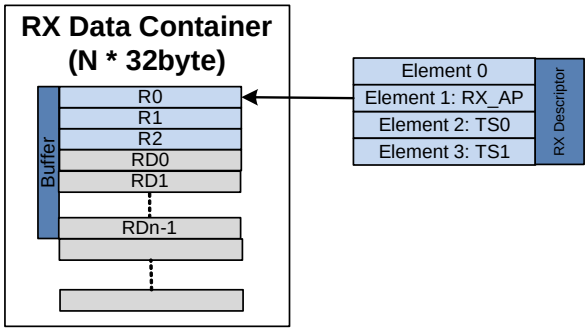

Figure: CAN XL RX message (single descriptor)

1.4.5.10.2 Multiple RX Descriptor

To optimize the memory usage, regardless of the payload size of the RX message received, several RX descriptors can be assigned to one RX message. Doing so, the RX message is written in several data containers. When one is full, the RX message data is going to the next one. As depicted in the figure below, for a given size of data container (constant per RX FIFO Queue), the RX message can be written anywhere into the S_MEM. The MH takes care of filling the right data container with the RX message data whenever required. As a fixed memory allocation is defined per RX descriptor, the RX message data may be spread over several data containers and RX descriptors (depends on RX message payload data).

The figure below shows three RX descriptors and their assigned data container to hold the entire RX message. If a data container has a size of 96byte (N=3) and a CAN XL message payload of 270byte is received, then the RX message is depicted in figure below. Although the CAN XL message, in this example, is split over several RX descriptors, this configuration allows to support Classical CAN and CAN FD with only one RX descriptor.

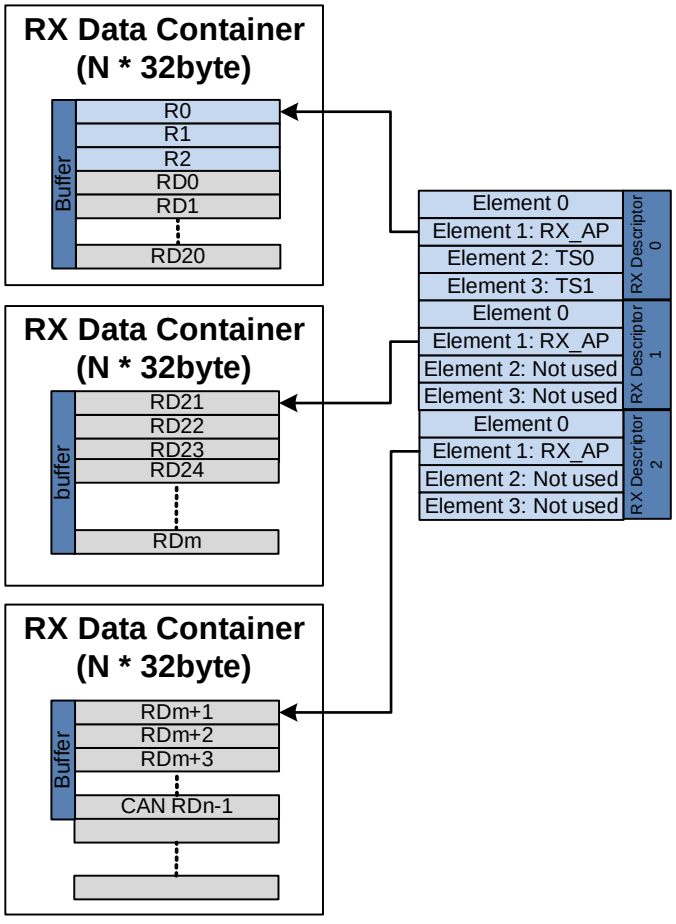

Figure: RX message (multiple descriptors)

1.4.5.11 RX Message in Continuous Mode

Prior to reading this chapter, read the RX descriptor chapter first.
For a better understanding of that section please also read the RX Message in Normal mode chapter.
In the Continuous mode, the RX messages, instead of being split over several data containers (see RX Message in Normal mode chapter), are merged in the same big data container one after the other.
As depicted below only a single data container is defined per RX FIFO Queue and one RX descriptor is used per RX message.

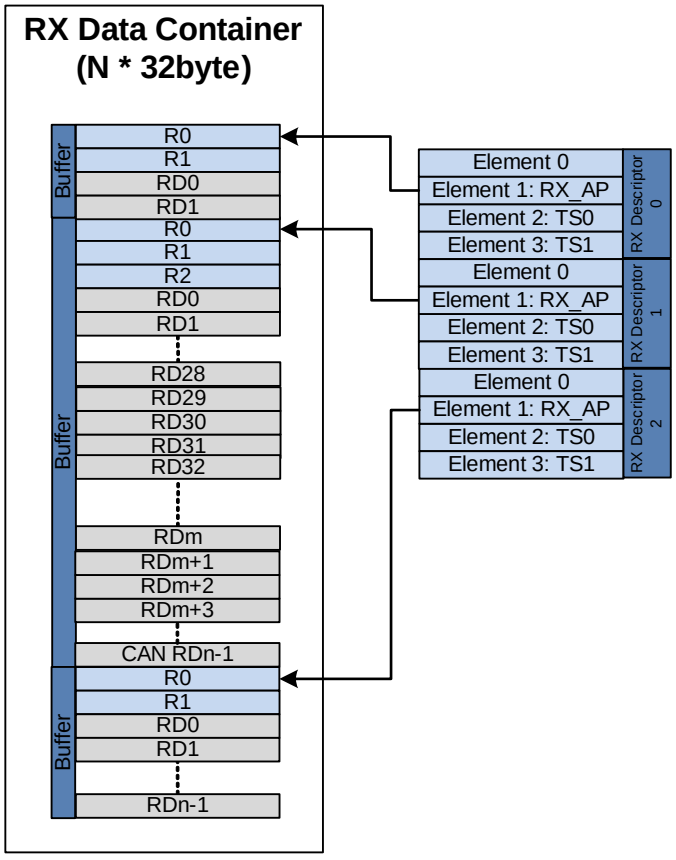

Figure: RX message (Continuous Mode)
The Continuous mode makes use of the already defined RX descriptor list to support the SW management of RX messages, see RX message Normal mode for more details.
It is important to note that the Continuous mode applies to all RX FIFO Queues when set. There is no option to make it available only to some queues.

The RX descriptors are attached to a defined RX FIFO Queue. The RX FIFO Queue, to write the RX message, is defined according to the RX filtering rules, see RX Filter chapter for more details. Once the RX FIFO Queue is identified, the latest RX descriptor (meaning the current one) is fetched from S_MEM. As the RX descriptors in a given FIFO Queue are concatenated, they will be read in sequence up to the end of the RX message.

Every RX descriptor is assigned to only one RX message in this large data container. This data container size is a multiple of the maximum burst lengths supported, 8x32bit with a maximum of 131040 byte (4095*32byte) and a minimum of 32byte. Every RX FIFO Queue has its own data container (defined by a start address and a size).

The header and the payload of the RX message are written one after the other to the S_MEM. This approach gives the advantage to have the complete RX message data available in one place in the S_MEM. A copy of the RX message in the S_MEM can then be easily defined by a start address and a size.

Whenever it is possible, the embedded DMA controller uses the maximum burst length to write header and payload data and will adapt the latest burst length to complete its transfer.

The address pointer, used to write the RX message, is always 32bit aligned despite payload data is byte aligned.

The data container is defined anywhere in the S_MEM. Being defined as a 32bit address pointer, it can be defined in a 4G byte memory area.

The NEXT bit defined in the RX Descriptor will never be set, as only one RX Descriptor is used per RX message. The RX Descriptor pointing to the RX message has the HD bit set to 1 as it is a Header Descriptor. No Trailing Descriptors are used for such mode, only one TX descriptor is required, and it is a Header Descriptor.

Only the RX Header Descriptor holding the header data is acknowledged when an RX message is received, considering the Normal mode. The same applies for the Continuous mode, see RX message in Normal mode chapter for more details.

Compared to the Normal mode, there is no trade-off to consider regarding the different CAN protocol payload data size. As the RX messages are written in a row, no loss of memory is expected in the data container assigned to an RX FIFO Queue.

This mode will also ensure that RX data are always linearly and continuously written in the S_MEM. At the beginning of the reception of an RX message header, a check is performed to ensure the RX data to be received can fit entirely in the data container. As the RX message cannot be written at the bottom and at the top of the data container, the MH will go to the start address of the data container before writing the first data. This would provide to the SW an easy way to perform memory copy, as one start address, and a size can define the overall RX message.

1.4.5.12 Descriptor Acknowledgement

For the TX and RX paths, the MH is providing data information back to the RX and TX Header Descriptor.

To do so, some place holders are defined in the RX and TX descriptors for the MH to write RX and TX message status, timestamping and error reporting.

As the CAN bus is not a full duplex interface, there shouldn't be any collision on the acknowledge of RX and TX descriptors, with the exception of the PRT when in loopback mode. In such mode, all TX messages transmitted by the MH are send back by the PRT to the MH, refer to the PRT chapter for detailed description of the loopback.

The process of acknowledgement is completely separated from the reception or transmission of a CAN frame, a dedicated DMA channel is reserved for such purpose.

1.4.5.12.1 RX Descriptor

For the RX path, one or several RX descriptors can be used to hold the complete RX message. Once an RX message is received successfully, an acknowledgement is written back to the Header Descriptor when the message is completed. If several descriptors are used per message (Trailing Descriptors), they are not changed by the MH. If the data container assigned to the RX descriptor is sized in such a way that any RX message can fit in entirely, then every RX descriptor (in fact Header descriptors in that case) will be acknowledged.

If several RX descriptors are used to store the RX message and an issue occurs while processing the message, all RX descriptors already used are then released for the next RX message.

Here below is the list of bit fields used by the MH to provide the acknowledgement information to the RX Header descriptor, see RX Descriptor chapter for details:

- VALID: The MH expected this bit to be set to 0 by the SW to ensure the data container is ready to be written again. This bit is written by the MH to 1, when an RX message is received successfully, and the data are available in S_MEM. It is true only for the RX descriptor holding the RX message header data

- TS0[31:0] and TS1[31:0]: The MH writes the 64bit timestamp (TS0 and TS1) in the RX descriptor when the RX message data is received successfully. Only the RX descriptor holding the data will have this bit field updated.

- NEXT: As soon as more than one RX descriptor is required for an RX message, the MH sets this bit to 1. Only the RX Header descriptor will have this bit field updated. The Trailing descriptors are not updated. When using the Continuous mode, this bit is always set to 0.

- HD: In case several RX descriptors are used to define an RX message, the MH sets this bit to 1 to identify which descriptor has the message header embedded into its data buffer.

- STS[3:0]: This bit field gets updated by the MH for any usage of RX descriptor. It provides information on the status of the RX message and on any related issue while RX FIFO Queues are running

1.4.5.12.2 TX Descriptor

For the TX path, once a TX message is processed, the TX Header Descriptor is written back with the relevant information. The list of conditions to trigger an acknowledge is defined below:

- Message sent successfully
- Message rejected by the TX filter (see TX Filter chapter)
- Message discarded after several re-transmission
- Message rejected by the PRT (see HFI codeword in PRT chapter)

Here below is the list of bit fields used by the MH to provide the acknowledgement information to the TX descriptor, see TX Descriptor chapter for details:

- VALID: The MH expected this bit to be set to 1 by SW to ensure the data buffer is ready to be sent, only the TX descriptors having this bit set to 1 are accepted and executed. This rule applies for every TX descriptor with or without the header data (when TX message is split over several descriptors). When the last data defined by this TX descriptor has been sent over the CAN bus, it will be set back to 0 by the MH only to the TX descriptor holding the header data.

- TS0[31:0] and TS1[31:0]: When the TX message data is sent successfully, a 64bit timestamp is written back into to the TX descriptor holding the header data

- STS[3:0]: This bit field gets updated by the MH for any usage of TX descriptor. It provides information on the status of the TX message and on any related issue while TX FIFO Queues are running

1.4.5.13 TX FIFO Queue

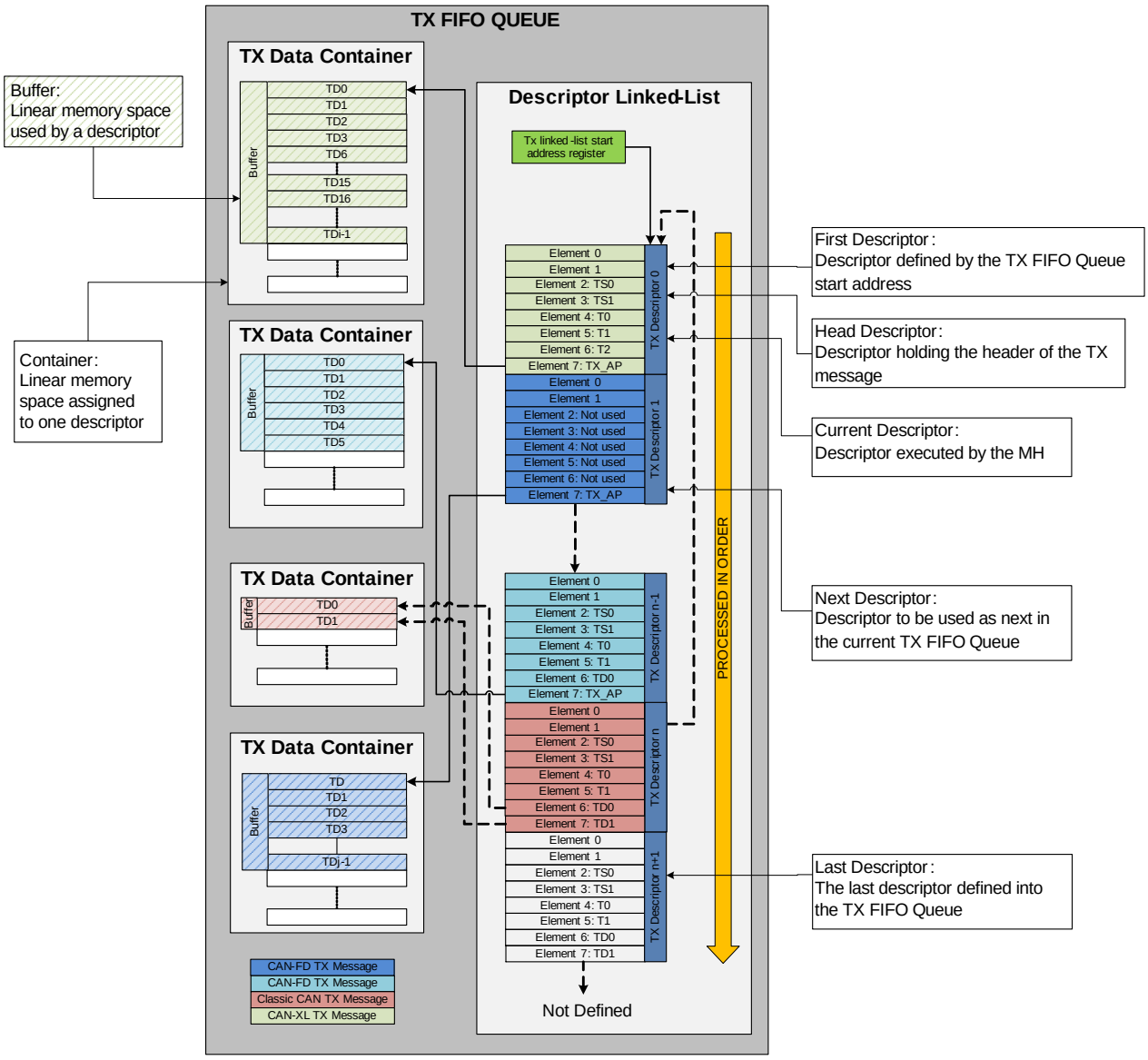

Figure: TX FIFO Queue description

Up to 8 TX FIFO Queues can be defined and managed by the MH.

When the SW wants to configure N TX FIFO Queues, only the queue number from 0 to N-1 can be used.

A TX FIFO Queue is a list of TX messages to be sent in order to the PRT.

Each one being fully independent from the others, the SW can declare and add new messages to any of the FIFO Queue without stopping the execution of the others or the current one. In this sense, the TX FIFO Queues can be enabled or disabled individually. An abort mechanism is provided to stop and flush each TX FIFO Queue individually.

Prior to launch any TX FIFO Queue, the MH must be started (**MH_CTRL.START** written to 1 will drive the **MH_STS.BUSY** bit status to 1). To start the TX FIFO Queue n, write 1 to the **TX_FQ_CTRL0.START[n]**. Before launching a TX FIFO Queue n, it must be enabled by setting the **TX_FQ_CTRL2.ENABLE[n]** bit to 1. Once enabled and started, there is no way to disable it while it is

running without a defined procedure. Instead, the abort bit **TX_FQ_CTRL1.ABORT[n]** provides a way to stop a TX FIFO Queue n running and to ensure a safe stop and flush of ongoing data. For more detail on starting and stopping TX FIFO Queues, refer to the Application Information chapter.
To ensure no dead lock can occur at start, the *ENABLE* signal from the PRT must be set to 1, to allow any TX FIFO Queue to start. This signal status can be monitored in the **MH_STS.ENABLE** bit field.

There is also nothing preventing the SW to declare and run a TX FIFO Queue with a defined list of TX messages, assuming an interrupt at the end of the TX FIFO Queue execution.

However, the TX FIFO Queue can be used as a circular buffer when the Last Descriptor defines a wrap to the First Descriptor (WRAP bit set to 1 in TX descriptor). Doing so, the SW can add new messages in an endless manner over time.

The mechanism, used to manage TX FIFO Queues, is based on the concept of linked list. Any TX FIFO Queue is defined using a linked list of TX descriptors and data buffers to read TX message payload from the S_MEM.

A linked list is made of descriptors, where a descriptor is defined by several data elements of the same size, the element is a 32bit word. Each element provides some information or would define some actions to perform. A descriptor is built by the SW but will be read and executed by the MH.

Every TX descriptor is of the same size, pointing to a data buffer and also to the next descriptor, as depicted in the figure above. The TX descriptors are continuous in memory (to ease and simplify implementation). Therefore, it is not required to declare or use a bit field to mention the position of the next descriptor as it is implicit.

The linked list is started by fetching the First Descriptor in the list, once it is fully read, it is executed and the data buffer assigned to it, is read. Other actions can be defined into the element data like triggering an interrupt or setting a flag. The linked list is processed one descriptor at a time, once a descriptor is complete, the next one is fetched into the list and the process repeats itself. The process keeps going up to the Last Descriptor of the linked list and from this point in, may end or may wrap to the first descriptor in a circular buffer mode.

Every TX FIFO Queue defines its own order of TX messages to be send to the CAN bus, but as several queues are running concurrently, an arbitration process is performed between queues to select the highest priority message. Every TX message is filtered to ensure only the required ones can be sent. The SW builds those queues with messages and the MH takes care of sending them whenever appropriate.

For the TX FIFO Queues, data buffers hold the payload data of the TX message while the descriptor defines header information. In some cases, the first payload data may also be part of the Head Descriptor.

To give a status report and some information like timestamping, the MH is also able to write back some elements in the TX descriptors. Not all of them are written back but only the one having the Header Descriptor data are updated.

It is possible to wrap at the top of the TX FIFO Queue any time but with the following constraints:

- The WRAP bit must be set in the Header Descriptor to identify where the next TX message is located

The descriptors are mainly defined on SRAM as they drive the actions to be taken. Nothing prevents the SW to declare and use the E_MEM instead but it can slow down the execution and may create real time issues. However, the data buffers can be either in E_MEM, which is usually the case, or in SRAM. As a matter of fact, if the next descriptor cannot be fetched before the relevant data are fully read or written, the linked list execution speed would depend on the data access time. To solve this issue, outstanding reads are performed to hide the system latency whenever possible.

TX Data Containers can be defined at any location in S_MEM. But for performance reason and to optimize the burst access, it is highly recommended to have the TX buffer 32byte aligned. Those containers are considered as memory space that is allocated by the Memory Management Unit to store buffer. Once a message is sent, the container can be deallocated, so the memory space is released for further usage.

As soon as the TX FIFO Queue is started, the MH will fetch the First Descriptor and store it to L_MEM. When the TX descriptor is available in L_MEM, it will be part of the arbitration process. As long as the TX message defined by this TX descriptor is not sent to the CAN bus, it will remain for all the arbitration runs. When it is sent successfully, the next TX descriptor of that TX FIFO Queue is fetched automatically.

The MH will proceed with all TX FIFO Queues in the same way. As the TX message to be sent is based on its priority, the TX FIFO Queues will run at a different rate up to the point that all TX messages are sent successfully.

If the Last Descriptor of a TX FIFO Queue sets the END bit, the MH will end the FIFO execution as soon as the TX message defined is transmitted successfully on the CAN bus and the TX descriptor acknowledge is written to the S_MEM.

Up to 1023 TX descriptors can be defined for a TX FIFO Queue. When the maximum number of TX descriptor defined for a TX FIFO Queue is reached, the MH wraps automatically to its initial start address to fetch the next TX descriptor. Despite this default behavior, it is still possible at any time for the SW to mention a wrap using the WRAP bit in the Header Descriptor.

When the END bit is not set for the last TX descriptor, the TX FIFO Queue is considered as endless, and any new TX descriptor can be appended to the already defined last descriptor. To allow such way of working, the last descriptor must always be not valid (VALID bit set to 0). This is very important as the detection of the non-valid TX descriptor triggers an interrupt to the system to declare that the TX

FIFO Queue is stopped. It would then be up to the SW to append a new TX descriptor and restart the TX FIFO Queue.

If for some reasons a TX FIFO Queue has an error, it is still possible to abort the execution of that TX FIFO Queue. When such action is performed, the TX FIFO Queue will be considered as active as long as the current data transfer assigned to a TX descriptor is not finished. This means, the TX descriptor is not considered for the arbitration anymore, so no more fetches are done, and the TX FIFO Queue is set inactive.

Any safety issue related to a TX descriptor executed by a TX FIFO Queue will stop it right away. The TX FIFO Queue is declared as no more valid and is stopped. To identify such issue, some interrupts are triggered to the system, *TX_CRC_ERR* and *TX_SFTY_STS*. Despite that the faulty TX FIFO queue is stopped, the others will keep going.

If a message has reached maximum number of re-transmissions or has declared an invalid header format, the message is skipped and the next one in the TX FIFO Queue is considered instead. The error mentioning such skip is written back to the report status bit field in the TX Header Descriptor.

In a context where a TX descriptor provides the definition of one TX message, the next TX message is the next TX descriptor, an offset of 1 (1x32byte) is required.

A TX FIFO Queue n is controlled and monitored using several registers and bit registers:

- The **TX_FQ_START_ADD{n}** (n € {0, 1, 2, ..., 7}) register to define the start address of the TX FIFO Queue n
- The **TX_FQ_CTRL0.START[n]** (n € {0, 1, 2, ..., 7}) register to launch the TX FIFO Queue n
- The **TX_FQ_SIZE{n}** (n € {0, 1, 2, ..., 7}) register to define the maximum number of TX descriptor for the TX FIFO Queue n before looping back to the initial start address

- The **TX_FQ_ADD_PT{n}** (n € {0, 1, 2, ..., 7}) register to monitor the current address pointer of the TX FIFO Queue n

- The **TX_DESC_ADD_PT** register to monitor the current address pointer
- The **TX_FQ_CTRL1.ABORT[n]** (n € {0, 1, 2, ..., 7}) bit register to abort the execution of the TX FIFO Queue n
- The **TX_FQ_CTRL2.ENABLE[n]** (n € {0, 1, 2, ..., 7}) bit register to enable the TX FIFO Queue n prior to use it
- The **TX_FQ_INT_STS.SENT[n]** and **TX_FQ_INT_STS.UNVALID[n]** (n € {0, 1, 2, ..., 7}) bit registers to identify respectively, a message is transmitted, and an invalid TX descriptor is detected

- The **TX_FQ_STS0.BUSY[n]** and **TX_FQ_STS.STOP[n]** (n € {0, 1, 2, ..., 7}) bit registers to know respectively, the status of the TX FIFO Queue n, busy (TX FIFO Queue is active) and stopped or running

- The **TX_FQ_STS1.ERROR[n]** and **TX_FQ_STS.UNVALID[n]** (n € {0, 1, 2, ..., 7}) bit registers to identify the root cause of the TX FIFO Queue being stopped, an error is detected, or an invalid TX descriptor is detected

A TX FIFO Queue is being controlled for any issue using common bit registers when receiving interrupts:

- The **SFTY_INT_STS.TX_DESC_CRC_ERR** and **SFTY_INT_STS.TX_DESC_REQ_ERR** bit registers to identify respectively, any CRC issue on TX descriptor running in the TX FIFO Queue n and non-expected TX descriptor

- The **ERR_INT_STS.DP_TX_ACK_DO_ERR** bit register to identify overflow on TX ACK data path for the TX FIFO Queues
- The **ERR_INT_STS.DP_TX_SEQ_ERR** bit register to identify if an issue occurs on the TX_MSG interface

1.4.5.13.1 Basic Mode

The SW defines one TX descriptor per TX message payload data. Thus, a TX message would be:

- One TX descriptor to provide the complete header information
- One TX data container to hold the complete TX message payload data (only required for CAN FD and CAN XL when payload data is over 8byte)

Data containers holding the TX payload buffer can be declared anywhere in the S_MEM despite being attached to only one TX descriptor, as depicted in the figure below.

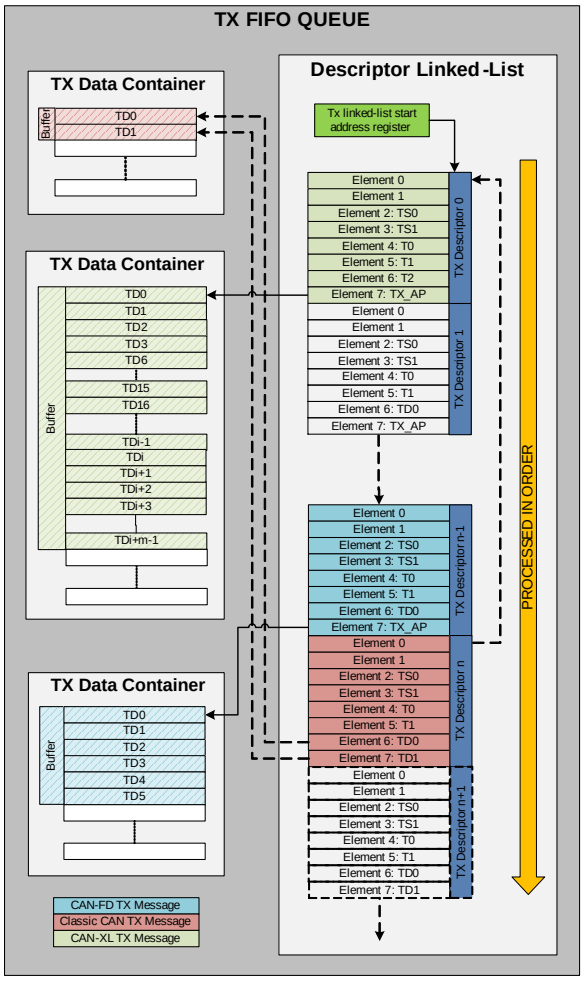

Figure: TX FIFO Queue (Basic Mode)
This approach provides less constraints on system as only one TX descriptor needs to be fetched per TX message. It would be much more efficient in terms of performance and memory allocation, regarding linked list descriptors.

The only constraint for such configuration would be the memory space allocated to the payload data. As the CAN XL can support up to 2048 byte, the size of the data container to hold the complete payload can be quite large.

1.4.5.14 TX Priority Queue

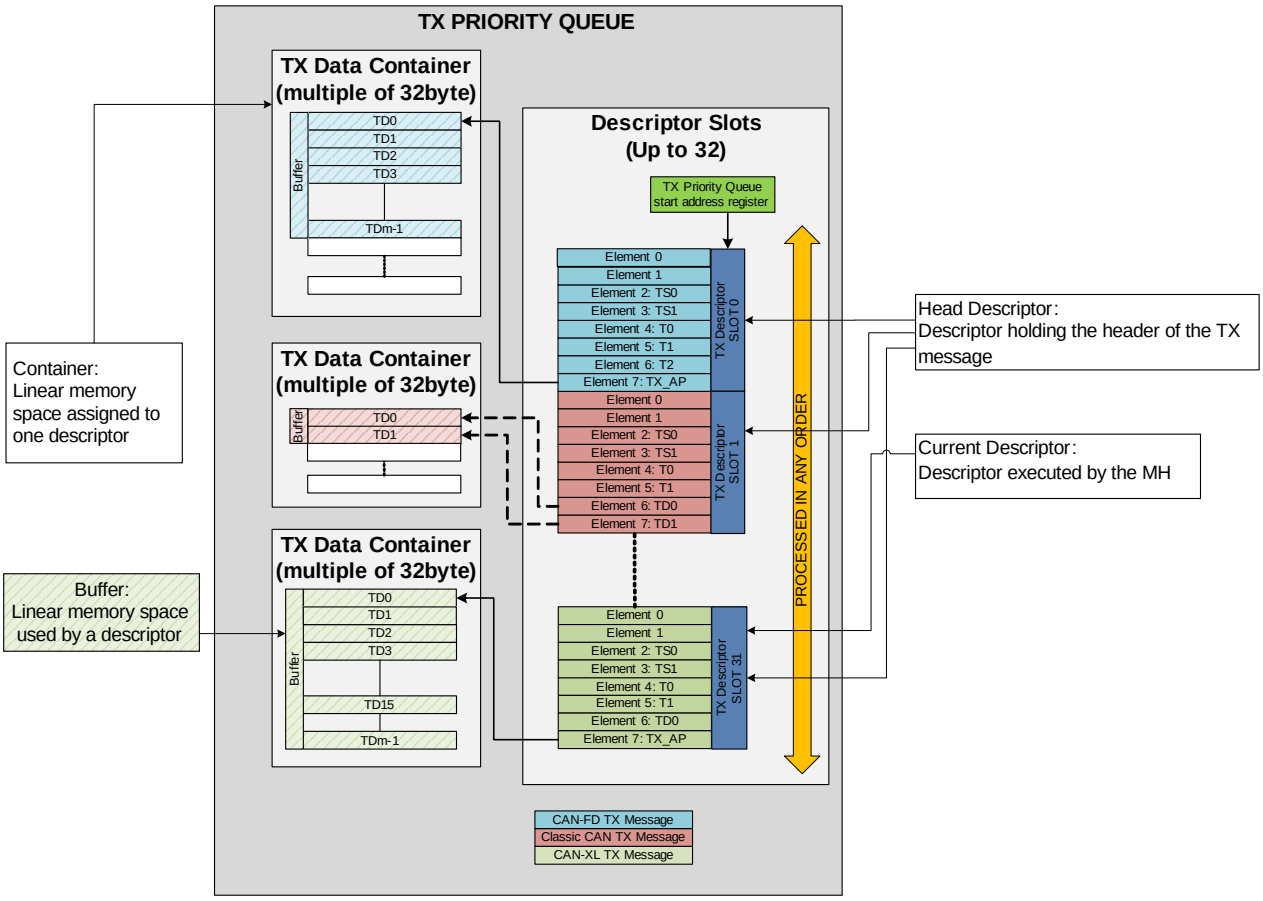

Figure: TX Priority Queue description

This kind of queue does not behave as the TX FIFO Queue, but the way messages are defined and how the MH is reading the descriptor are identical.

A TX Priority Queue can be configured with a maximum of 32 slots.
When the SW wants to configure N TX Priority Queue slots, only the slot number from 0 to N-1 can be used.
Every slot is assigned one TX message from a SW point of view. Every slot can be enabled/disabled individually leaving the option to define any number of active slot or none in the SW. Compared to the TX FIFO Queue, there is no order of execution. Any message defined in the TX Priority queue can be selected and executed in any order, only the highest priority message is selected first. Those messages are evaluated against the one currently in use in all TX FIFO Queues.

The same principle is used to define a TX message, meaning some TX descriptors and TX data buffers to define a message. Like the TX FIFO Queues, data buffers hold the payload data of the TX message while descriptor defines header information. In some cases, the first payload data may also be part of the descriptor.

To give a status report and some information like timestamping, the MH writes back some bit field in the TX Header Descriptor. This way the SW is able to track back TX messages sent and identify those with issues.

The TX Priority Queue is using the same data path as the one defined and implemented for the TX FIFO Queue.
The main difference between the two queues is:

- The Priority Queue is managed in any order. As soon as a slot over the 32 is available, a new message can be defined

- The SW needs to trigger the MH to consider a new message in a slot
- The SW needs to read status register to identify which message has been sent

Prior to launch any TX FIFO Queue slot, the MH must be started (**MH_CTRL.START** written to 1 will drive the **MH_STS.BUSY** bit status to 1). To start the TX FIFO Queue slot n, write 1 to the **TX_PQ_CTRL0.START[n]**. Before launching a TX FIFO Queue slot n, it must be enabled by setting the **TX_PQ_CTRL2.ENABLE[n]** bit to 1. Once enabled and started, there is no way to disable it while running without a defined procedure. Instead, the abort bit **TX_PQ_CTRL1.ABORT[n]** provides a way to stop a TX FIFO Queue slot n running and to ensure a safe stop and flush of ongoing data. For more detail on starting and stopping TX Priority Queue slots, refer to the Application Information chapter.
To ensure that no dead lock can occur at start, the *ENABLE* signal from the PRT must be high to allow any TX FIFO Queue to start. This signal status can be monitored in the **MH_STS.ENABLE** bit field.

As soon as one or several slots of the TX Priority Queue are started, the MH will fetch the relevant TX descriptors defined at those locations and stores them in L_MEM. When the TX descriptors are available in the L_MEM, they will be part of the arbitration process. As long as the TX messages defined by those TX descriptors are not sent to the CAN bus, they will remain for all the arbitration runs. It is important to note the TX FIFO Queue's messages are also part of this arbitration process.

When the TX message is sent successfully on the CAN bus and TX descriptor acknowledge is written to S_MEM, the TX Priority Queue slot is released and set inactive. A *TX_PQ_IRQ* interrupt can be triggered to the SW when the TX descriptor acknowledge is written to the S_MEM. The other option would be to poll the corresponding status bit register, to identify when the transfer has completed. This last approach requires much more CPU time compared to the interrupt one.

As the TX message to select is based on an arbitration process, the TX Priority Queue execution will run at a different rate compared to the TX FIFO Queues. If TX messages are defined into the TX Priority and have highest priority, they will go between TX FIFO Queues. The SW can add new messages at any time when a slot is available.

If for some reason a TX Priority Queue slot n needs to be stopped, it is still possible to abort the execution of that slot. When such action is performed, the TX Priority Queue slot n will be considered as no more active. If the TX message assigned to this slot is already in progress to the CAN bus or has been selected as the next message to be sent, it won’t be cancelled. By using a register status, it is possible to identify if the slot aborted has been done before or after the sending of the TX message.

Any safety issue related to a TX descriptor executed by a TX Priority Queue slot is declared as no more valid. This means, it will not be part of the arbitration process with the other slots but won't prevent the others to be executed. To identify such issue a *TX_DESC_CRC_ERR* is sent to the system. Despite this TX Priority Queue slot is stopped, the others will keep going with their own TX descriptors.

If some message doesn't go through for the two following reasons, maximum number of restarts reached or invalid header format, the message is discarded. The error status being detected is written back to the TX descriptor holding the header data.

There is a way to keep track of the TX descriptors used for a given TX FIFO Queue, refer to the Trace and Debug chapter.

A TX Priority Queue is controlled and monitored using several registers and bit registers:

- The **TX_PQ_START_ADD** register to define the start address of the TX Priority Queue
- The **TX_PQ_CTRL0.START[n]** (n € {0, 1, 2, ..., 31}) bit register to launch the TX Priority Queue slot n
- The **TX_PQ_CTRL1.ABORT[n]** (n € {0, 1, 2, ..., 31}) bit register to abort the execution of the TX Priority Queue slot n
- The **TX_PQ_CTRL2.ENABLE[n]** (n € {0, 1, 2, ..., 31}) bit register to enable the TX Priority Queue slot n prior to use it
- The **TX_DESC_ADD_PT** register to monitor the current address pointer
- The **TX_PQ_STS0.BUSY[n]** (n € {0, 1, 2, ..., 31}) bit register to know the status of the TX Priority Queue slot n, either busy (TX Priority Queue Slot is having a TX message to send) or not busy (Slot is no more active for reasons like message sent, safety issue, ...)
- The **TX_PQ_STS1.SENT[n]** (n € {0, 1, 2, ..., 31}) bit register to know the status of the TX message assigned to the TX Priority Queue slot n, either sent (TX message assigned to slot n is sent) or not sent (potential reasons are safety issue, max re-transmission counter reached, ...)
- The **TX_PQ_INT_STS0.SENT[n]/ TX_PQ_INT_STS1.SENT[n]** and **TX_PQ_INT_STS0.UNVALID[n]/ TX_PQ_INT_STS1.UNVALID[n]** (n € {0, 1, 2, ..., 31}) bit register to identify respectively, a message is transmitted, or an invalid TX descriptor is detected

A TX Priority Queue is being controlled for any issue using common bit registers:

- The **SFTY_INT_STS.TX_DESC_CRC_ERR** and **SFTY_INT_STS.TX_DESC_REQ_ERR** bit registers to identify respectively, any CRC issue on TX descriptor running in the TX FIFO Queue n and non-expected TX descriptor
- The **ERR_INT_STS.DP_TX_ACK_DO_ERR** bit register to identify overflow on TX ACK data path for the TX FIFO Queues

- The **ERR_INT_STS.DP_TX_SEQ_ERR** bit register to identify if an issue occurs on the TX_MSG interface

1.4.5.15 RX FIFO Queue in Normal Mode

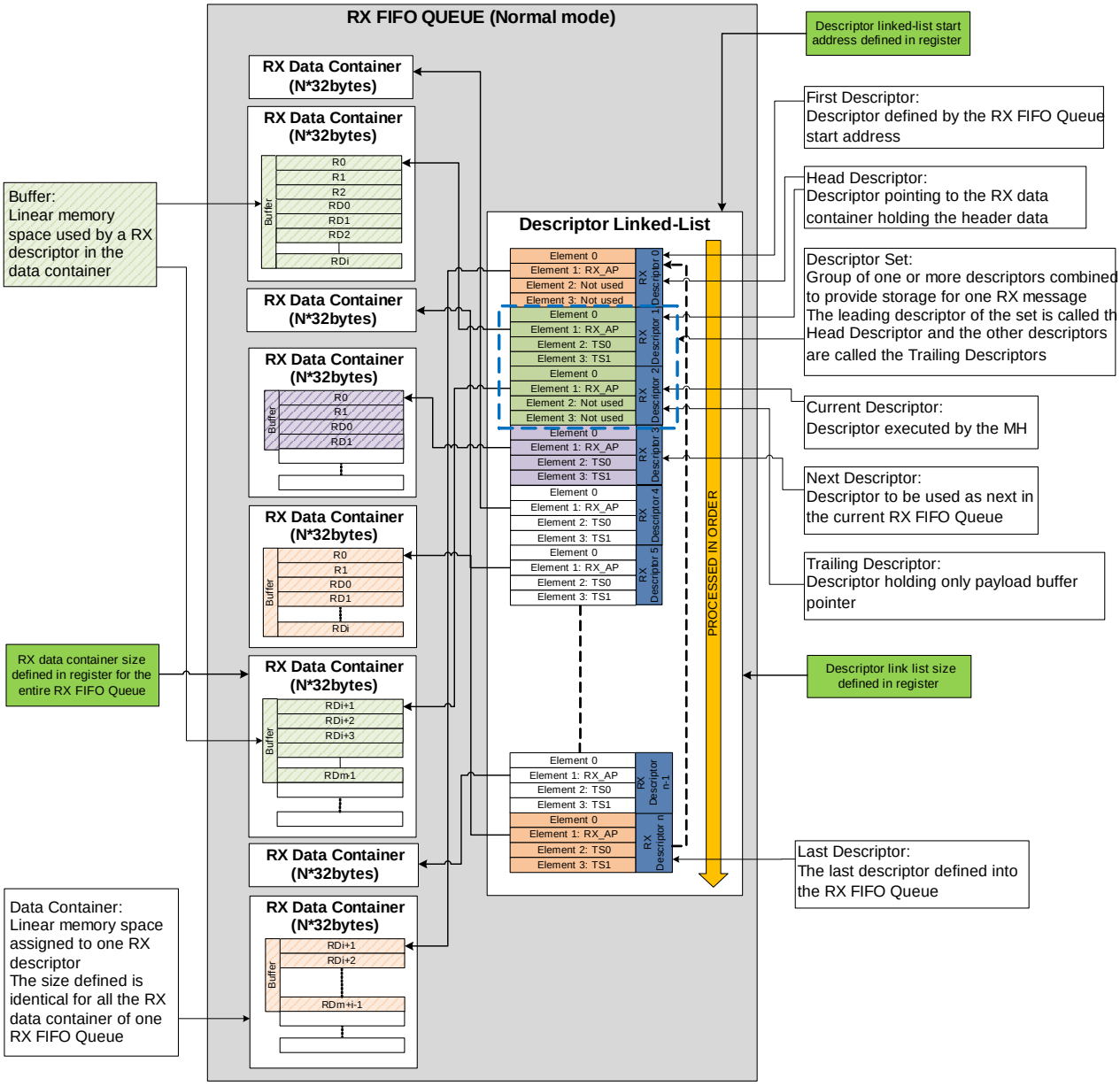

Figure: RX FIFO Queue description (Normal mode)

Up to 8 RX FIFO Queues can be defined and managed by the MH.

When the SW wants to configure N RX FIFO Queues, only the queue number from 0 to N-1 can be used.

An RX FIFO Queue is a list of RX descriptors pointing to an RX data container to store the RX messages received by the PRT.

The RX filtering rules, programmed by the SW, define if a message is rejected or accepted. In case it is accepted, it defines which RX FIFO Queue receives the message. If a message is rejected, it won't appear in any of the FIFOs. Each one being fully independent from the others, the MH appends new RX message as they arrive on the CAN Bus.

The mechanism to manage RX FIFO Queues is based on the concept of linked list. Any RX FIFO Queue uses a linked list of RX descriptors and RX data containers. Those containers are used to write the RX message data to the S_MEM and have fixed size over the entire RX FIFO Queue. A different size can be defined per RX FIFO Queue but must always be a multiple of 32byte.

The size of the data container is programmable to store small or large RX message payload data, if required. Up to **RX_FQ_SIZE{n}.DC_SIZE[6:0]** * 32byte data container size can be defined per RX descriptor in an RX FIFO Queue n. As the size is programmable per RX FIFO Queue, it is then possible to limit the memory footprint according to the expected message to be received.

A linked list is made of descriptors, where a descriptor is defined by several data elements of the same size, the element is 32bit word. Each element provides some information or would define some actions to perform. A descriptor is built by the SW but will be read and executed by the MH. Every descriptor is of the same size, pointing to a data container and also to the next descriptor. The link between descriptors is just a fixed offset, as they are continuous in memory (to ease and simplify implementation). Therefore, it is not required to declare or use a bit field to mention the position of the next descriptor as it is implicit (dashed lines). As data containers have a fixed size and the RX message received may change in size, several descriptors can be required.

As messages are received in a continuous way, the RX FIFO Queue are used in a circular buffer mode. This means when the Last Descriptor is reached, the MH will consider the First Descriptor as the next descriptor. The Last Descriptor is defined by the size of the RX FIFO Queue and the start address of the RX FIFO Queue.

An RX filter in the MH is used to accept or reject RX messages. If a message is accepted, it is then sent to a defined RX FIFO Queue. The RX filter builds those queues over time with messages based on the filtering result. It is up to the SW to read them in time.

The RX filter observes all the incoming RX messages to identify the right RX FIFO Queue. Once defined, the first RX descriptor attached to the selected RX FIFO Queue is fetched and used to write the incoming data to the S_MEM. As soon as the incoming RX data increases above the limit of the data buffer pointed by the current RX descriptor, a new one is fetched to keep going. This process repeats up to last RX data received.

The MH will proceed with all the RX FIFO Queues the same way. As the RX FIFO Queue selected depends on the RX filtering result, the RX FIFO Queues will be filled up at a different rate.

Prior to launching any RX FIFO Queue, the MH must be started (**MH_CTRL.START** written to 1 will drive the **MH_STS.BUSY** bit status to 1). To start the RX FIFO Queue n, write 1 to the

RX_FQ_CTRL0.START[n]. Before launching an RX FIFO Queue n, it must be enabled by setting the **RX_FQ_CTRL2.ENABLE[n]** bit to 1. Once enabled and started, there is no way to disable it while running without a defined procedure. Instead, the abort bit **RX_FQ_CTRL1.ABORT[n]** provides a way to stop an RX FIFO Queue n running and to ensure a safe stop and flush of ongoing data. For more detail on starting and stopping RX FIFO Queues, refer to the Programming Guidelines chapter.
It is essential to configure and start the relevant RX FIFO Queues before starting the PRT. When the MH is not started and so no RX FIFO Queues are started, the MH will not accept any RX data, leading to a PRT data overflow.

Each RX FIFO Queue can be managed individually, the SW can decide to enable or disable any queue according to the way RX messages must be managed. Once the RX filter is defined and the PRT is receiving messages, any change on the RX FIFO Queue setting is not possible. There is still a mechanism to abort and flush an RX FIFO Queue while others are running.

Once a linked list is started and an RX message needs to be written inside, the first descriptor in the list is read. It is executed and the data buffer assigned to it, is written. Other actions can be defined into the element data like triggering an interrupt or setting flags. The linked list is processed one descriptor at a time, if more RX descriptors are required for a given message, the next one into the list is fetched and the process repeats itself. The process keeps going up to the last descriptor of the linked list, a wrap will occur automatically at this time

If the size of the container is small, the RX message with few payload data may fit in but a larger one would require several descriptors and containers. This approach optimizes the memory usage as the number of containers used is very close to the effective size of the RX message received. However, such strategy requires more descriptors and Data Containers.

If we consider the other way round, a large data container avoids splitting data buffers and limits the number of descriptors. The main disadvantage would be the usage of more memory per descriptor.

This is up to the SW to find the best trade-off according to the CAN protocol and to the application required. There is also the option to size the data container for every RX FIFO Queue differently, leaving some flexibility of optimization.

Before receiving any RX message, the RX FIFO Queues must be started. In case some messages are received and the RX FIFO Queue to write data is not active the RX message is rejected and an *RX_ABORT_IRQ* interrupt is triggered to the system.

To give a status report and some information like timestamping, the MH is also able to write back some elements in the RX or TX descriptors. Not all of them are written back but only the one having the header data defined.

The same remark regarding TX descriptors and TX data buffer location into memory applies for the RX descriptors and data buffers.

The SW must always ensure that some RX descriptors in the RX FIFO Queue are always valid (VALID bit set to 0). In case an RX descriptor is not valid, the RX FIFO Queue n is stopped and an interrupt *RX_FQ_IRQ* is sent to the system. If the system provides a valid RX descriptor and restarts the RX FIFO Queue n in time, the RX message may be written into memory, otherwise the message is rejected and the interrupt *RX_FQ_IRQ* is triggered to the system.

Up to 1023 RX descriptors can be defined for an RX FIFO Queue. The size of the RX FIFO Queue is defined such a way when the Last Descriptor is reached, the MH wraps automatically to its initial start address to get the First Descriptor.

If for some reasons an RX FIFO Queue has an error, it is still possible to abort the execution of that FIFO Queue. When such action is performed, the RX FIFO Queue will be considered as active as long as the current data transfer assigned to an RX descriptor is not finished. This means no more fetches are done and the RX FIFO Queue is set inactive.

Any issue related to an RX descriptor executed by an RX FIFO Queue will stop it right away. To identify such issue, some interrupts are triggered to the system, *RX_DESC_CRC_ERR* or *RX_DESC_REQ_ERR*. Despite this RX FIFO Queue is stopped, the others will keep going through their own RX descriptors.

An RX FIFO Queue is controlled and monitored using several registers and bit registers:

- The **RX_FQ_START_ADD{n}** (n € {0, 1, 2, ..., 7}) bit register to define the start address of the RX FIFO Queue n
- The **RX_FQ_CTRL0.START[n]** (n € {0, 1, 2, ..., 7}) bit register to launch the RX FIFO Queue n
- The **RX_FQ_CTRL1.ABORT[n]** (n € {0, 1, 2, ..., 7}) bit register to abort the execution of the RX FIFO Queue n
- The **RX_FQ_CTRL2.ENABLE[n]** (n € {0, 1, 2, ..., 7}) bit register to enable the RX FIFO Queue n prior to use it
- The **RX_FQ_SIZE{n}.MAX_DESC** and **RX_FQ_SIZE{n}.DC_SIZE** (n € {0, 1, 2, ..., 7}) bit register to define respectively, the maximum number of RX descriptor before looping back to the initial start address and the Data Container size for the RX FIFO Queue n

- The **RX_FQ_ADD_PT{n}** (n € {0, 1, 2, ..., 7}) register to monitor the current address pointer of the RX FIFO Queue n

- The **RX_FQ_STS0.BUSY[n]** and **RX_FQ_STS0.STOP[n]** (n € {0, 1, 2, ..., 7}) bit registers to know respectively, the status of the RX FIFO Queue n, busy (RX FIFO Queue is active) and stopped or running or not started

- The **RX_FQ_STS1.ERROR[n]** and **RX_FQ_STS1.UNVALID[n]** (n € {0, 1, 2, ..., 7}) bit registers to identify the root cause of the RX FIFO Queue being stopped, an error is detected, or an RX descriptor is invalid

- The **RX_FQ_INT_STS.RECEIVED[n]** and **RX_FQ_INT_STS.UNVALID[n]** (n € {0, 1, 2, ..., 7}) bit registers to identify respectively, a message is received, and an invalid RX descriptor is detected

An RX FIFO Queue is being controlled for any issue using common bit registers:

- The **SFTY_INT_STS.RX_DESC_CRC_ERR** and **SFTY_INT_STS.RX_DESC_REQ_ERR** bit registers to identify respectively, any CRC issue on RX descriptor running in the RX FIFO Queue n and non-expected RX descriptor

- The **ERR_INT_STS.DP_RX_ACK_DO_ERR** bit register to identify overflow on RX ACK data path for the RX FIFO Queues
- The **ERR_INT_STS.DP_RX_FIFO_DO_ERR** bit register to identify overflow on RX DMA FIFO for the RX FIFO Queues
- The **ERR_INT_STS.DP_RX_SEQ_ERR** bit register to identify if an issue occurs on the RX_MSG interface

1.4.5.15.1 Fragmented Data Container

The RX Data Container can be defined into any location and so an RX message is split across several area in the S_MEM. With such approach, an address pointer is given to the application for any RX message data, no copy is performed. A new Data Container is then allocated to replace the one being sent to the application. It is important to note that in case of an RX message is received into several Data Containers, several address pointers will need to be provided. It is assumed that Data Containers that belongs to the same message can only be released once all RX buffer data have been read.

When the MH has executed the Last Descriptor (the descriptor defined at the latest position in the Descriptor linked list), it wraps automatically to the First descriptor automatically.

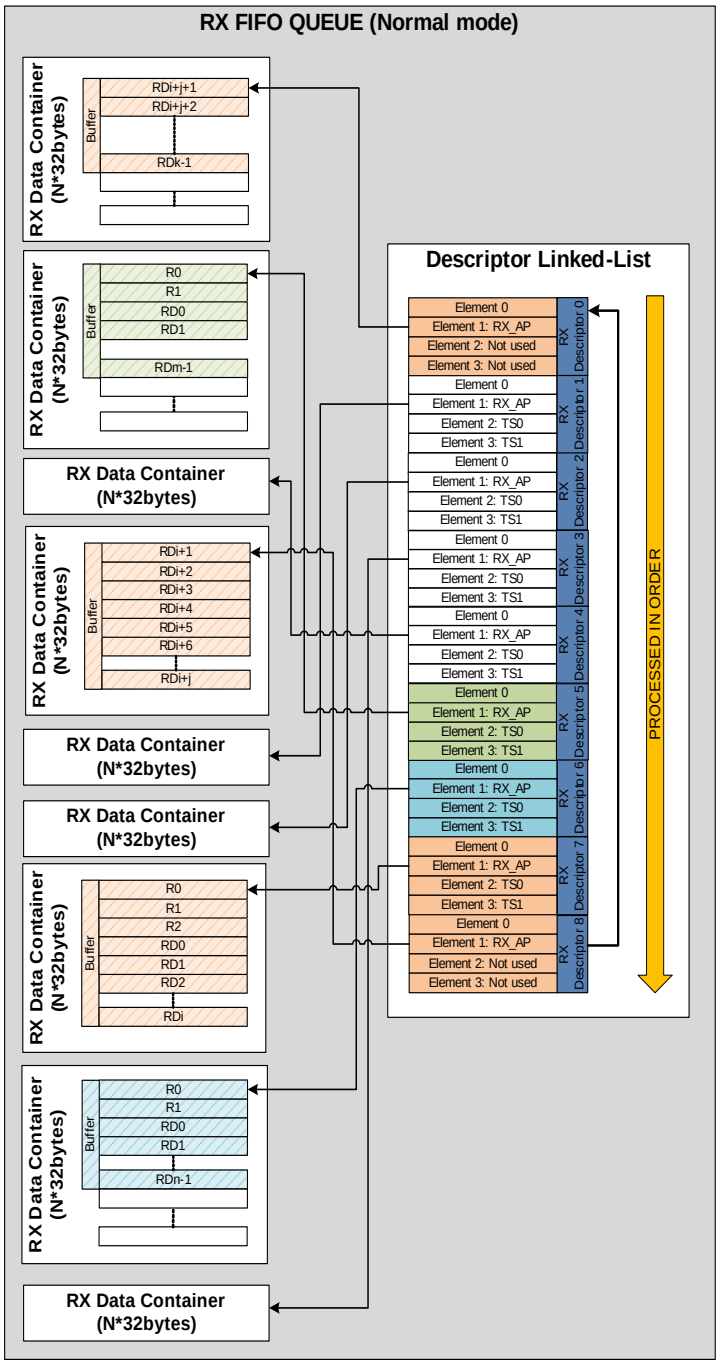

Figure: RX FIFO Queue in Normal Mode (Fragmented data containers)

In some cases, the RX message can then be split across RX descriptor being at the top and at the bottom of an RX FIFO Queue. This mode does make use of all the RX descriptors defined in a given RX FIFO Queue. It may happen that the SW would prefer to rely on linear RX message, having in mind a linear organization of data containers in S_MEM, see Continuous data container chapter.

1.4.5.15.2 Continuous Data Container

As the RX data containers of the same message are spread to different location in L_MEM, it won't be easy for the SW to read the entire message. To get around this issue, the SW can decide to declare the RX Data Container in a linear memory area and to have them continuous to each other as depicted below.

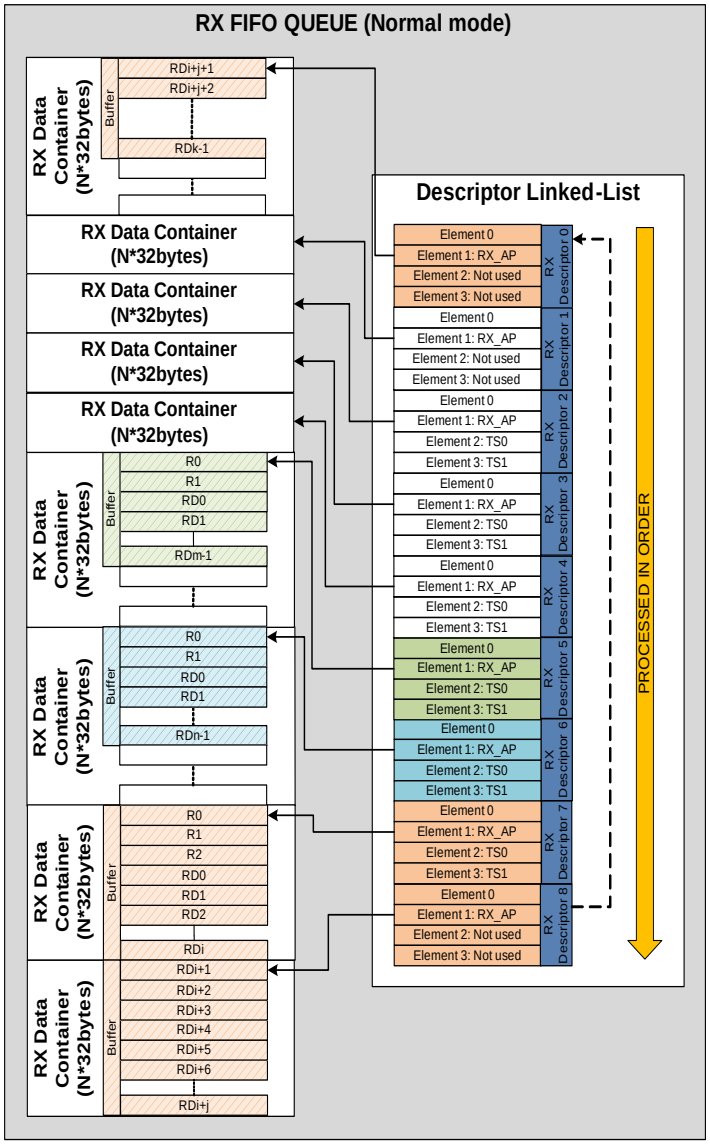

Figure: RX FIFO Queue in Normal Mode (Continuous Data Containers)

This way of managing the RX message will have the main advantage to provide an RX message written in a linear memory area, despite being split in several data containers. The current RX message will not be aligned right at the end of a data container. Thus, there will be free memory space in between RX messages but this may be acceptable for a SW point of view.

The only exception would be when the Last Descriptor is used, and the message size exceed the Data Container. The MH wraps and uses the First Descriptor to keep going with the current RX message data. In this particular scenario the RX message data is split over the top and the bottom of the data container and the same applies for the linked list holding the RX descriptor. This is normal behavior, and it must not be an issue for the SW to read the RX message following the RX descriptor list from bottom to top.

This configuration provides a pseudo linearity for the RX messages in S_MEM, excepted at the borders. Doing so, the SW would need to perform a copy of the RX message data to free the memory area for the new incoming messages. Such configuration does not require any update on address pointer in the RX descriptors. Only the VALID bit needs to be written by the SW to acknowledge the reading of the RX message and the update of the read address pointer register.

1.4.5.16 RX FIFO Queue in Continuous Mode

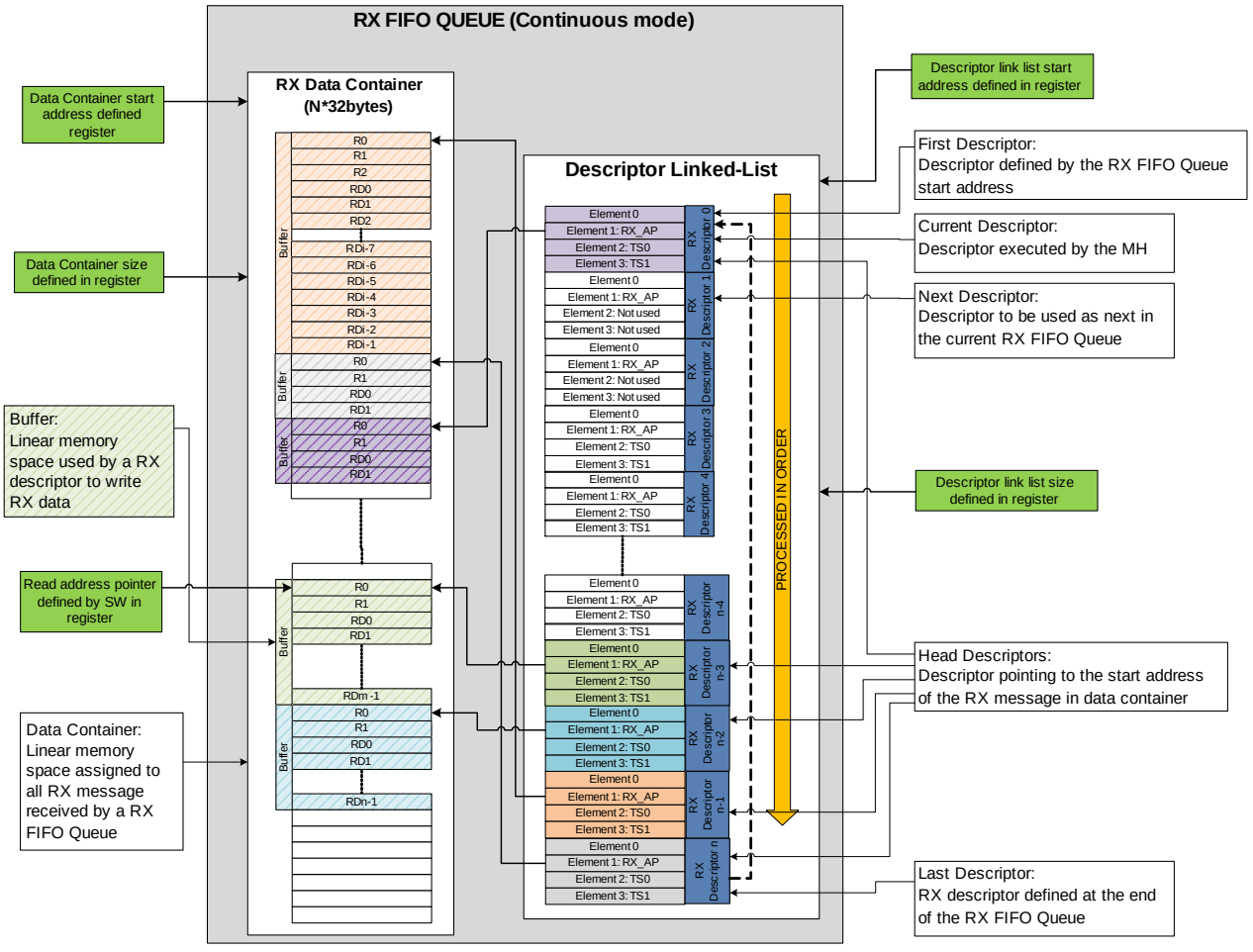

Figure: RX FIFO Queue in Continuous mode

The same principle, that is defined in the RX FIFO Queue in Normal mode, applies for the RX descriptors in this Continuous mode. The way they are used, managed, and defined remain the same, see RX FIFO Queue in Normal mode chapter.
The main difference comes from the structure of the RX message data being stored in the S_MEM.

An RX FIFO Queue when in Continuous mode is a list of RX descriptors pointing to a large and single data container to store all the RX messages received by the PRT.

The RX filtering rules, programmed by the SW, define if a message is rejected or accepted. In case it is accepted or rejected, it is decided which RX FIFO Queue receives the message. If a message is rejected, it won't appear in any of the FIFOs. Each one is fully independent from the others. The MH appends to RX FIFO Queues new RX message as they arrive on the CAN Bus. The RX filter builds those queues over time with messages based on the filtering results. It is up to the SW to read them in time.

The mechanism to manage RX FIFO Queues is based on the concept of linked list. Any RX FIFO Queue when in Continuous mode is defined using a linked list of RX descriptors and a large data container.

As messages are received in a continuous way, the RX FIFO Queue are used in a circular buffer mode. This means, when the Last Descriptor is reached, the MH will consider the First Descriptor as the next descriptor. The Last Descriptor is defined by the size of the RX FIFO Queue and the start address of the RX FIFO Queue.

The RX filter observes all incoming RX messages to identify the right RX FIFO Queue. Once defined, the current RX descriptor attached to the selected RX FIFO Queue is fetched and used to define the new incoming RX message data.

The MH will proceed in the same way with all the RX FIFO Queues. As the RX FIFO Queue selected depends on the RX filtering result, the RX FIFO Queues will be filled up at a different rate.

Every RX FIFO Queue can be managed individually, SW can decide to enable or disable any queue according to the way RX messages must be managed. Once the RX filter is defined and the PRT is receiving messages, any change on the RX FIFO Queue setting is not possible. There is still a mechanism to abort and flush an RX FIFO Queue while others are running.

Once an RX FIFO Queue is started and an RX message needs to be written in its corresponding data container, the First descriptor in the descriptor list is read. It is executed and the initial start address of the data container assigned to it. The linked list processes one descriptor at a time every time a new RX message is received by the same RX FIFO Queue. The process continues up to the Last descriptor of the linked list before making a wrap.
Other actions can be defined in the RX descriptor, like triggering an interrupt or setting flags.

The MH writes messages as they arrive, to avoid overwriting. The SW needs to write the address value of the current message being read to a MH register. Therefore, the MH can compute the exact memory left to be used by the new RX message.

The size of the data container is programmable to store numerous CAN XL messages, if required. Up to **RX_FQ_SIZE{n}.DC_SIZE[11:0]** * 32byte data container size can be defined for an RX FIFO Queue n. As the size is programmable per RX FIFO Queue, it is then possible to limit the memory footprint according to the expected message to be received.

Before receiving any RX message, the RX FIFO Queues must be started. In case some messages are received and the RX FIFO Queue to write data is not active, the RX message is rejected and an *RX_ARBORT_IRQ* interrupt is triggered to the system.

To give a status report and some information like timestamping, the MH is also able to write back some elements in the RX or TX descriptors. Not all of them are written back but only the one having the header data defined.

The same remark, regarding TX descriptors and TX data buffer location into memory, applies for the RX descriptors and data buffers.

The SW must always ensure that some RX descriptors in the RX FIFO Queue are always valid (VALID bit set to 0). In case an RX descriptor is not valid, the RX FIFO Queue n is stopped and an interrupt *RX_FQ_IRQ* is sent to the system. If the system provides a valid RX descriptor and restarts the RX FIFO Queue n in time, the RX message may be written into memory, otherwise the message is rejected and the interrupt *RX_FQ_IRQ* is triggered to the system.
In the Continuous mode, one RX descriptor is assigned to one message, thus, once the SW has read the message in the data container, the VALID bit can be set to 0 right away. The SW must indicate to the MH, using the **RX_FQ_RD_ADD_PT{n}** (n € {0, 1, 2, ..., 7}) registers, the address pointer value of the last word read from S_MEM. The MH uses this information to estimate if the incoming message data can be written safely in the data container.

Up to 1023 RX descriptors can be defined for an RX FIFO Queue. The size of the RX FIFO Queue is defined such that when the Last Descriptor is reached, the MH wraps to its initial start address to fetch the First Descriptor. As the RX FIFO Queue size for the RX descriptors is defined at the first time and cannot be changed once the RX FIFO Queue is started, the MH wraps automatically to keep going.

If, for some reasons, an RX FIFO Queue has an error, it is still possible to abort the execution of that FIFO Queue. When such action is performed, the RX FIFO Queue will be considered as active, as long as the current data transfers, assigned to an RX descriptor, are not finished. This means, there is no pending transaction attached to the RX FIFO Queue, this includes the RX descriptor acknowledge when a RX message is received. The *RX_FQ_IRQ* interrupt is triggered to the system, if enable, once the last RX message is received by the aborted RX FIFO Queue n.

Any issue related to an RX descriptor that is executed by an RX FIFO Queue will stop it right away. To identify such issue, some interrupts are triggered to the system, *RX_DESC_CRC_ERR* or *RX_DESC_REQ_ERR*. Despite of having this RX FIFO Queue stopped, the other ones will keep going through their own RX descriptors.

An RX FIFO Queue is controlled and monitored using several registers and bit registers:

- Refer to the list of registers already defined for the Normal mode
- The **RX_FQ_DC_START_ADD{n}** (n € {0, 1, 2, ..., 7}) register to be written by the SW to indicate to the MH the read address pointer in the data container for the RX FIFO Queue n

- The **RX_FQ_STS2.DC_FULL[n]** (n € {0, 1, 2, ..., 7}) bit register to identify the root cause of the RX FIFO Queue being stopped, there is no space left on the S_MEM to write new RX data. This issue may occur only if the MH is set to Continuous Mode.

An RX FIFO Queue is being controlled for any issue using common bit registers:

- Refer to the list of registers already defined for the Normal mode

1.4.5.17 TX FIFO Queue Data Flow

The SW defines the TX descriptors for every TX FIFO Queues to be used and declares the TX data buffers assigned to those TX descriptors.

As soon as the TX FIFO Queues are started, the TX MH will process and fetch all the relevant TX descriptors and will store them in the L_MEM for arbitration.

Only the TX message with the highest priority ID is sent first. Those messages will compete against the one defined in the TX Priority Queue. Only the TX descriptor holding the header data is written back with status information of the data transfer and a timestamp. As soon as a TX descriptor, from a TX FIFO Queue, is in use, the next one will be uploaded. The TX MESSAGE HANDLER manages the request for a new descriptor on its own, whenever this is required.

The following data flow describes how the TX FIFO Queues are running in parallel.
Here are the different steps when a TX message is selected and/or used:
Step 1: After transmitting a TX message from the TX FIFO queue n, the TX MESSAGE HANDLER will send a request to the DESCRIPTOR MESSAGE HANDLER for the next TX descriptor from that queue.

Step 2: The relevant TX descriptor of the TX FIFO Queue is fetched by the DESCRIPTOR MESSAGE HANDLER and is written to the L_MEM.

Step 3: As soon as the new TX descriptor is completely written into the L_MEM, an arbitration run is performed. This arbitration will identify, which TX descriptor has the highest priority, looping through the current TX descriptor for every TX FIFO Queues and through all the slots of the TX Priority Queue declared as active. Once the two first candidates (Priority Queue slot number or TX FIFO queue number) are defined, they are loaded in the TX MESSAGE HANDLER.

Step 4: The TX MESSAGE HANDLER then tries to upload the TX message with the highest priority locally. If a TX message is in progress, the TX MESSAGE HANDLER will wait for the end of the current transmission to read the complete TX descriptor from the L_MEM. If nothing prevents the upload of

the next descriptor, it will be done right away. As soon as the TX descriptor is locally stored, the first TX message data are sent to the PRT. The TX MESSAGE HANDLER will wait for the PRT to know if it has won the arbitration process. As long as no new TX descriptor changes the arbitration result, the selected TX descriptor remains in the TX MESSAGE HANDLER for further arbitrations. As soon as the TX message is winning the arbitration, all the data contained into the TX descriptor are sent to the PRT.

Step 5: The payload data assigned to the TX descriptor is fetched from the S_MEM.
Step 6: If the TX message is sent successfully on the CAN bus, an acknowledge request is sent to the DESCRIPTOR MESSAGE HANDLER with the status and information of the transfer. The DESCRIPTOR MESSAGE HANDLER writes back the acknowledge of that descriptor in the S_MEM. When the DESCRIPTOR MESSAGE HANDLER has finished writing the TX descriptor, an interrupt *TX_FQ_IRQ[n]* for the TX FIFO Queue n may be triggered to the system.

As all TX FIFO Queue are processed the same way, the data flow of only one TX FIFO Queue is depicted in figure below with the reference number for each step.

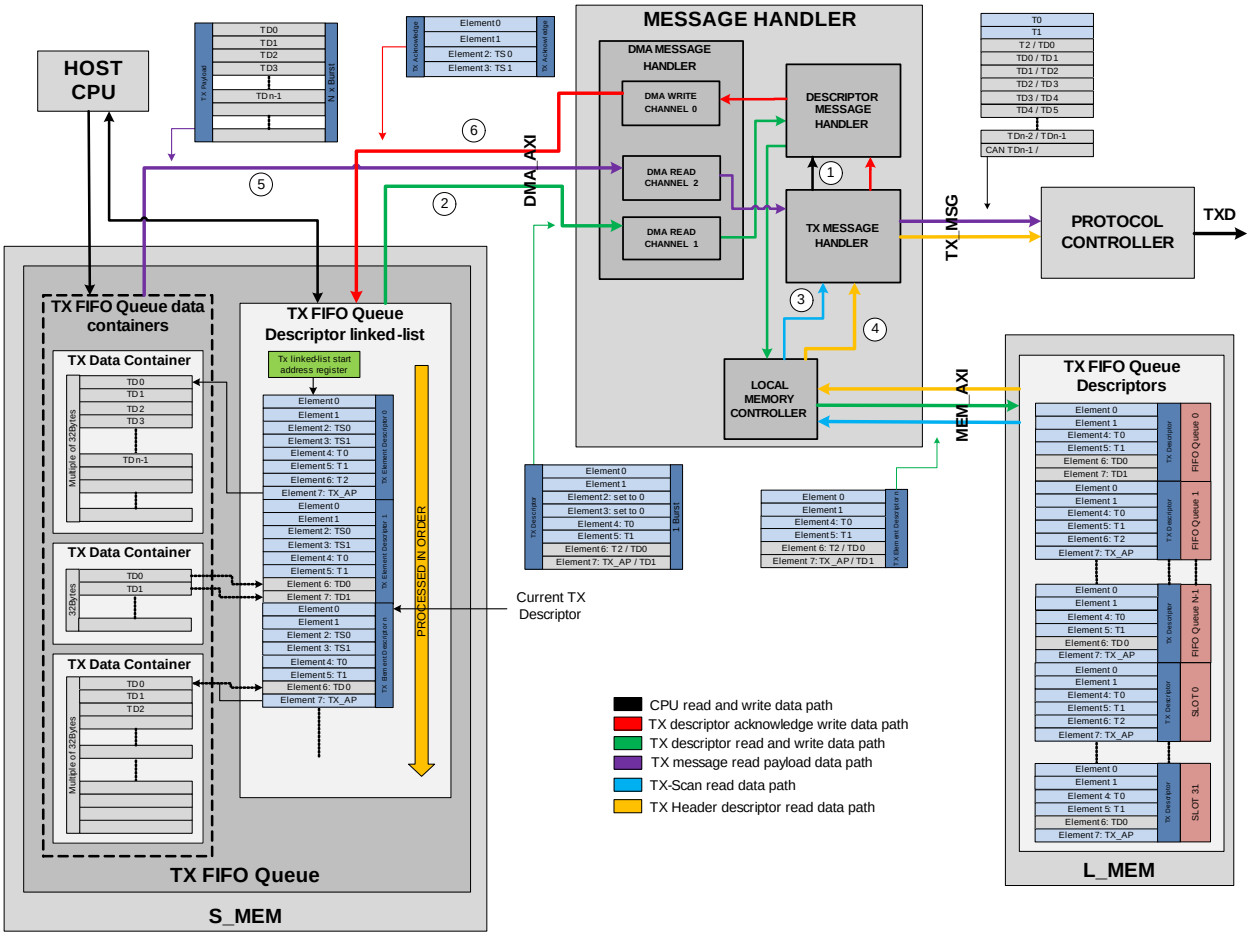

Figure: TX FIFO Queue data flow

1.4.5.18 TX Priority Queue Data Flow

The SW defines the TX descriptors, that have to be sent to the TX Priority Queue slot and declares the TX data buffers assigned to those TX descriptors. Once this is done, the SW triggers the TX MESSAGE HANDLER to have those messages sent as soon as possible. Those messages will compete against the one defined in the TX FIFO Queues. Only the ID is relevant for the selection of TX messages.

As soon as the TX Priority Queue slots are started, the TX MESSAGE HANDLER will process and fetch all the relevant TX descriptors and will store them in the L_MEM for arbitration.

When a TX message is sent, an acknowledge (status information and timestamp) is written back to the TX descriptor holding the header data. As soon as a TX descriptor is fully executed from a TX Priority Queue slot, it will be considered as inactive and won't be considered afterwards.

There is a way to keep track of the TX descriptors used for the TX Priority Queue, refer to the Trace and Debug chapter.

The following data flow is relevant for all TX Priority Queue slots.
Here below are the different steps when a TX message is selected and/or used:
Step 1: To trigger the TX message defined in the TX Priority Queue, the SW must write the start bit of the corresponding slot. Nothing prevents the SW to declare several TX messages at the same time and to launch them at once. The TX MESSAGE HANDLER sends requests to the DESCRIPTOR MESSAGE HANDLER for the TX descriptors to be fetched. If several TX descriptors need to be uploaded at once, they would be fetched in the order of their slot number, starting with 0

Step 2: The relevant TX descriptors of the TX FIFO Queue is fetched by the DESCRIPTOR MESSAGE HANDLER and is written to the L_MEM.

Step 3: As soon as the new TX descriptor is completely written to the L_MEM, an arbitration run is performed and only the TX descriptor uploaded for the slot will be considered. This arbitration will identify which TX descriptor has the highest priority, looping through the current TX descriptor for every TX FIFO Queue and through all slots of the TX Priority Queue that are declared as active. This selection is performed by doing a single read on all defined TX descriptor in the L_MEM. Once the two first candidates are identified, either the TX Priority Queue Slot number and/or the TX FIFO queue number, they are stored locally in the TX MESSAGE HANDLER.

Step 4: The TX MESSAGE HANDLER then tries to upload the TX descriptor with the highest priority locally. If a TX message is in progress, the TX MESSAGE HANDLER will wait for the end of the current transmission to read from the L_MEM the complete TX descriptor. If nothing prevents the upload of the next descriptor, it will be done immediately. As soon as the TX descriptor is stored locally, the first TX message data are sent to the PRT. The TX MESSAGE HANDLER will wait for the PRT to get the information if it has won the arbitration process. As long as no new TX descriptor changes the arbitration result, the selected TX descriptor remains in the TX MESSAGE HANDLER for further

arbitrations. As soon as the TX message wins the arbitration, all the data contained in the TX descriptor are sent to the PRT.

Step 5: The payload data assigned to the TX descriptor is fetched from the S_MEM.
Step 6: If the TX message is sent successfully on the CAN bus, an acknowledge request is sent to the DESCRIPTOR MESSAGE HANDLER with the status and information of the transfer. The DESCRIPTOR MESSAGE HANDLER writes the acknowledge of that descriptor back to the S_MEM. When the DESCRIPTOR MESSAGE HANDLER has finished writing the TX descriptor, an interrupt *TX_PQ_IRQ* for any of the TX Priority Queue slot may be triggered to the system. Once the acknowledge is written, the slot is considered as invalid and won't be used for the next arbitration run, up to the time, where the SW sets it back to active.

As all the TX Priority Queue slots are processed the same way, the data flow of one slot is depicted in the figure below with the reference number for each step.

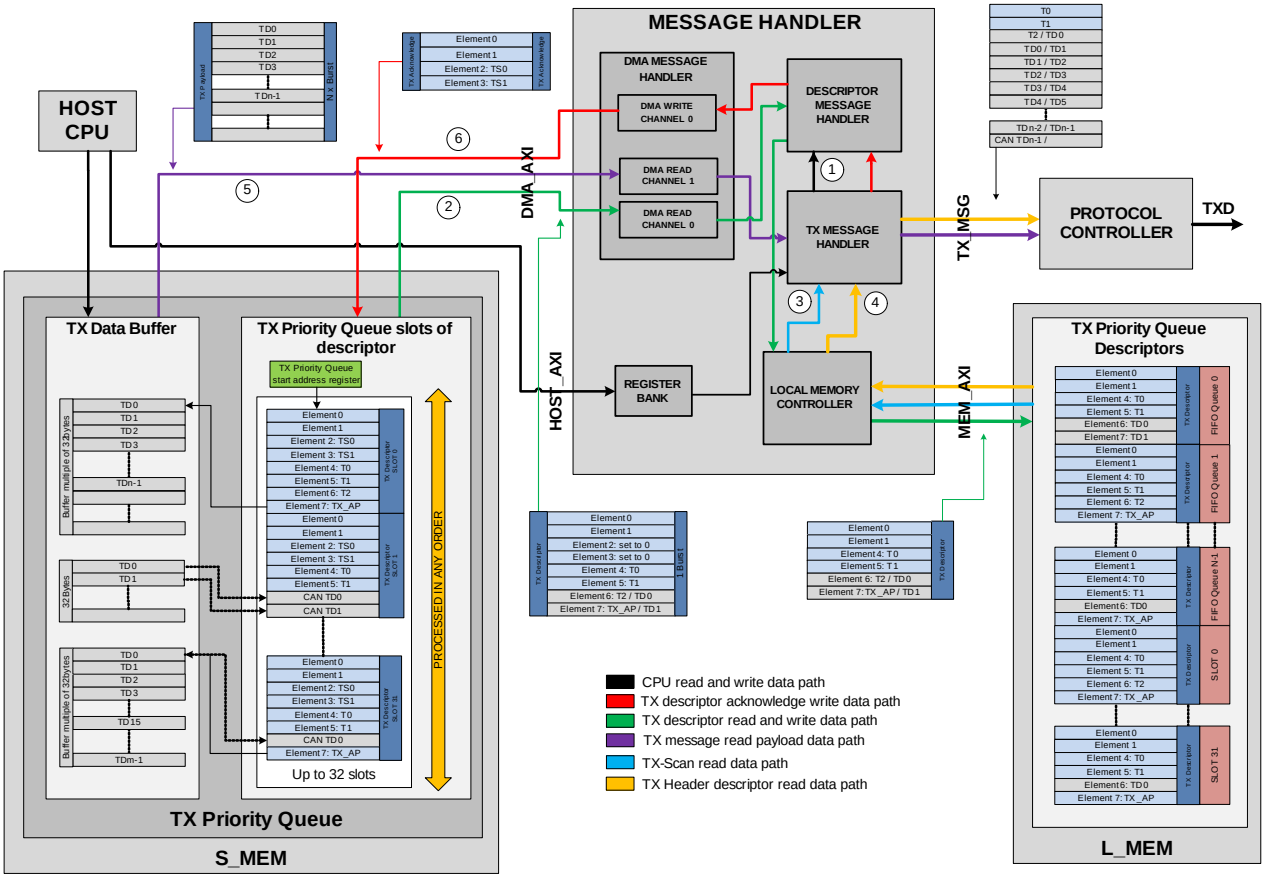

Figure: TX Priority Queue data flow

1.4.5.19 RX FIFO Queue Data Flow in Normal Mode

The SW needs to prepare the RX filter elements required to accept or reject RX messages. Once done, the SW writes those elements to the L_MEM. The SW cannot access this memory directly in write

mode through the HOST bus interface. As a consequence, the L_MEM must provide a way to protect the memory space allocated to the RX filtering elements from being read by any other masters.

The SW defines the RX descriptors for every RX FIFO Queue to be used and allocates the RX data buffers assigned to those RX descriptors.

As soon as the RX FIFO Queues are started, any RX messages will be filtered, meaning rejected or accepted, and are stored in the S_MEM when required.

Here below are the different steps when receiving an RX message:
Step 1: As soon as the RX message data R0, R1 and R2 are received, the RX filtering is started. All the incoming data are locally stored in the RX MESSAGE HANDLER, waiting for the result of RX filtering. The RX MESSAGE HANDLER identifies RX FIFO Queue to be used

Step 2: The RX MESSAGE HANDLER sends an RX descriptor request to the DESCRIPTOR MESSAGE HANDLER
Step 3: The relevant RX descriptor of the queue identified and fetched by the DESCRIPTOR MESSAGE HANDLER is given to the RX MESSAGE HANDLER

Step 4: The RX MESSAGE HANDLER uses the address pointer of the RX descriptor to write the message data to the S_MEM as soon as a complete burst is available. As long as the data buffer can accept message data, the process of writing can continue. In case that the last data can be written into the data buffer pointed by the current RX descriptor, go to Step 6, otherwise the next RX descriptor of the same queue is requested to the DESCRIPTOR MESSAGE HANDLER

Step 5: When the current RX descriptor is about to be completed, the new RX descriptor must be available for the next DMA data transfer, hereby go to Step 3

Step 6: The RX MESSAGE HANDLER gets the status of the last part of the RX message and the information of the latest data transfers. Those data are sent to the DESCRIPTOR MESSAGE HANDLER to be written back as an acknowledge to RX descriptor in the S_MEM holding the header. The timestamp and report status of the RX message are written at the same time. When the DESCRIPTOR MESSAGE HANDLER has finished writing the RX descriptor, an interrupt *RX_FQ_IRQ* for the RX FIFO Queue n may be triggered to the system

As all the RX FIFO Queues are processed the same way, the data flow of only one RX FIFO Queue is depicted in figure below with the reference number for each step.

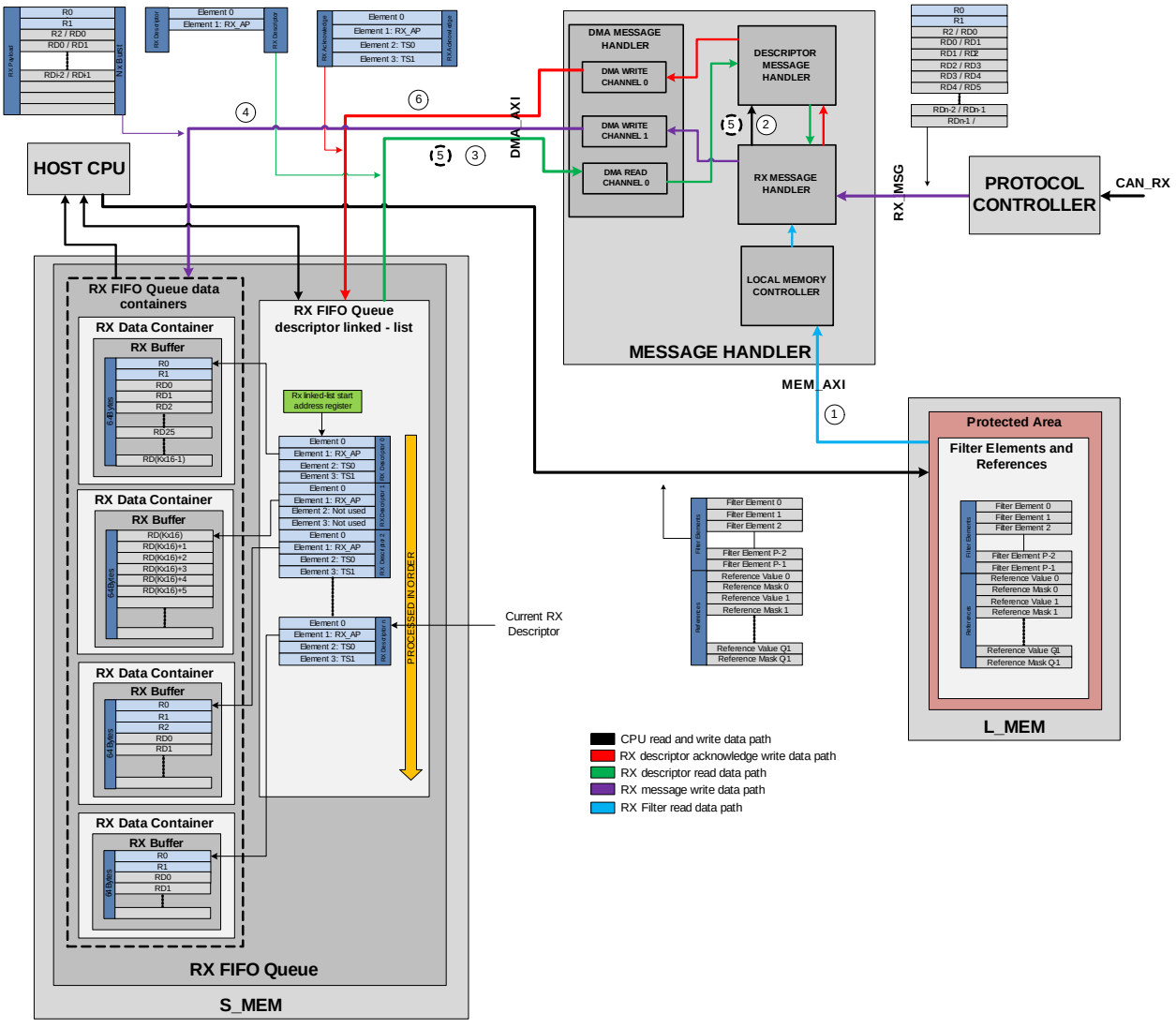

Figure: RX FIFO Queue data flow in Normal Mode

1.4.5.20 RX FIFO Queue Data Flow in Continuous Mode

The SW needs to prepare the RX filter elements required to accept or reject RX messages. Once done, the SW writes those elements to the L_MEM. The SW cannot access this memory directly in write mode through the HOST bus interface. As a consequence, the L_MEM must provide a way to protect the memory space allocated to the RX filtering elements from being read by any other masters.

The SW defines the RX descriptors for every RX FIFO Queue to be used and allocates the single data container for each of them.

As soon as the RX FIFO Queues are started, any RX messages will be filtered, meaning rejected or accepted, and stored into the S_MEM when required.

Here below are the different steps when receiving an RX message:

Step 1: As soon as the RX message data R0, R1 and R2 are received, the RX filtering is started. All the incoming data are stored locally in the RX MESSAGE HANDLER waiting for the result of RX filtering. The RX MESSAGE HANDLER identifies RX FIFO Queue to be used

Step 2: The RX MESSAGE HANDLER sends an RX descriptor request to the DESCRIPTOR MESSAGE HANDLER
Step 3: The relevant RX descriptor of the queue identified and fetched by the DESCRIPTOR MESSAGE HANDLER is given to the RX MESSAGE HANDLER

Step 4: The RX MESSAGE HANDLER holds the RX descriptor of that RX FIFO queue for further purpose. In case the current RX message cannot fit in the remaining space of the data container, the message is automatically written at the top (if possible). The message data are written to the S_MEM starting after the last RX message stored in the data container. As soon as a complete burst is available, it is written, and this process continues up to the last RX message data.

Step 5: The RX MESSAGE HANDLER gets the status of the last part of the RX message and the information of the latest data transfers. Those data are sent to the DESCRIPTOR MESSAGE HANDLER to be written back as an acknowledge to the RX descriptor fetched earlier from the S_MEM. The timestamp, the address of the RX message inside the data container and a report status of the RX message are written at the same time. When the DESCRIPTOR MESSAGE HANDLER has finished writing the RX descriptor, an interrupt *RX_FQ_IRQ* for the RX FIFO Queue n may be triggered to the system.

As all the RX FIFO Queues are processed the same way, the data flow of only one RX FIFO Queue is depicted in the figure below with the reference number for each step.

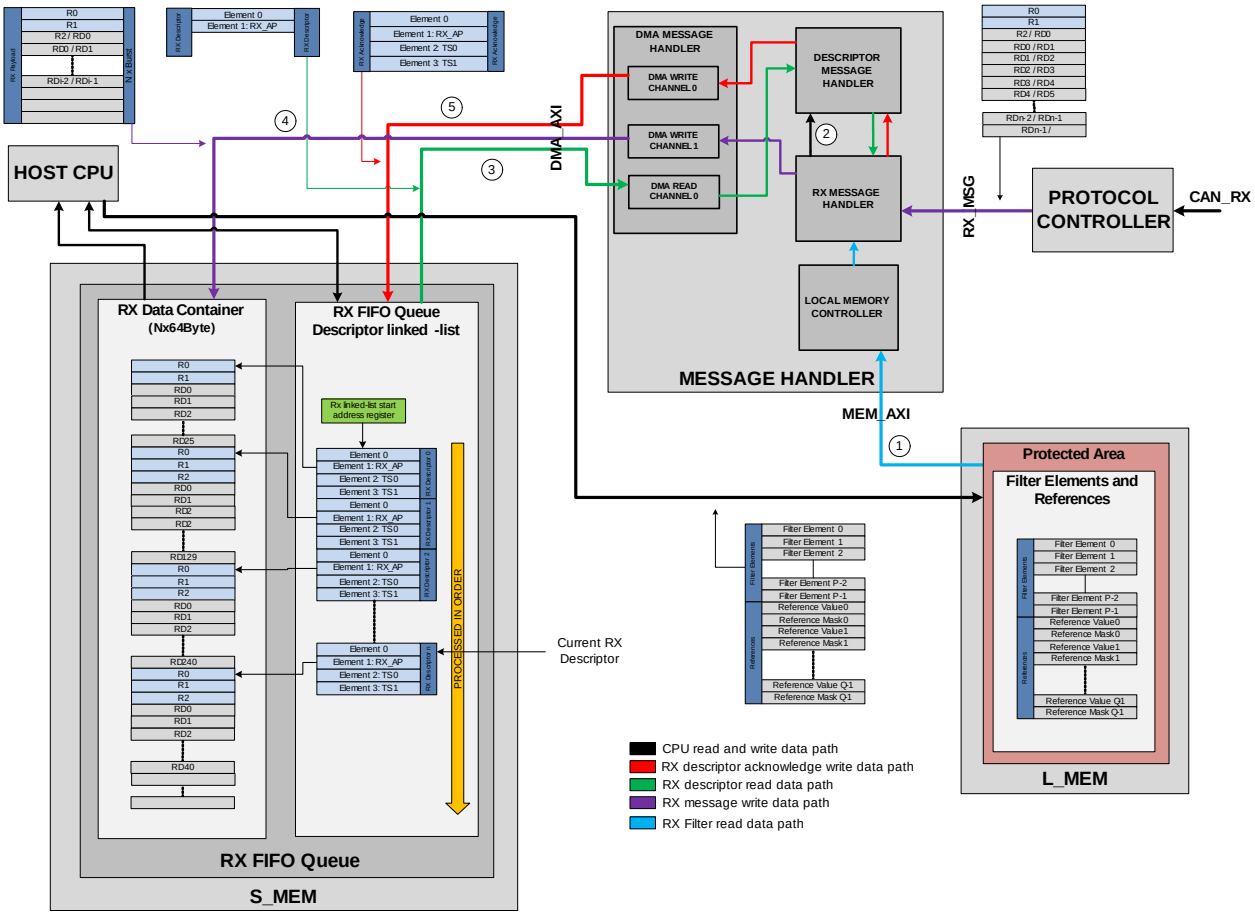

Figure: RX FIFO Queue data flow in Continuous Mode

1.4.5.21 TX-SCAN

To avoid any misunderstanding when talking about the selection of the next TX message to be sent to the PRT, the term TX-SCAN is used to define this process.

To arbitrate TX FIFO queues and cope with high latency in S_MEM, the TX Header Descriptor of every active TX FIFO Queues are stored into the L_MEM. The same applies for the TX Priority Queue slots when they are declared as active. It means up to 8 TX FIFO Queues Header Descriptor can be declared in L_MEM and up to 32 for the TX Priority Queue. Doing so, it becomes much easier to parse all the active TX Header Descriptors locally to identify which TX message has the highest priority. The TX-SCAN process would be very fast and the expected TX message order at CAN bus as close as possible to the one expected by the SW.

The TX-SCAN uses the list of TX descriptors available in L_MEM. When a new TX descriptor is added, a flag is set to indicate the availability of a new potential candidate. As long as the TX descriptor is not executed or discarded, it will remain as a valid candidate, see **TX_FQ_DESC_VALID** and **TX_PQ_DESC_VALID** registers.

Event to trigger a TX-SCAN run:

- A new TX message written in a TX Priority Queue slot
- A TX message sent successfully
- A TX message discarded after N re-transmissions
- A SW abort of a TX Priority Queue slot
- A SW abort of a TX FIFO Queue
- A TX message rejected by the TX Filter
- A TX message starting to be sent (in case of TX FIFO Queue, triggers the fetch of the next descriptor)

Any new TX message appended to a TX FIFO Queue does not trig a new TX-SCAN. The MH is only processing TX descriptor one after the other in every TX FIFO Queue, it does not know if a new message is added to one of them.

A re-transmission counter defines the number of re-transmissions allowed to the same TX message when this one is unsuccessful. For every trial of the same TX message, the re-transmission counter is incremented and compared to a maximum value defined in the **MH_CFG.MAX_RETRANS[2:0]**. If the counter exceeds the limit, the current TX message will no longer be considered and is skipped, the next TX message is taken instead. The re-transmission counter is set back to 0 when a new TX message is selected. There is the option to define an unlimited number of trials for TX messages.
The maximum number of re-transmissions is defined by the register **MH_CFG.MAX_RETRANS[2:0]** and covers the maximum value defined in CiA610-1.
Several options are defined:

- 0: No re-transmission
- 1 to 6: 1 to 6 re-transmissions
- 7: Unlimited re-transmissions (default value)

Here below is the definition of the 32bit priority value when considering Classical CAN, CAN FD, and CAN XL. The fields XLF, FDF, XTD, RTR, SRR and ID (defined in CAN protocol [1] and [2]) are used to determine the priority value of a given TX message. The priority value is computed for every TX message and then compared with each other to identify the highest priority message (the lowest value gives the highest priority message to transmit).
Only the T0 of the TX Header Descriptor is used for the selection of the highest priority message. As the relevant bits are defined in T0 element, only a single read access from the L_MEM is required. This would result to the following statement:
IMPORTANT: In Classical frame format, a data frame and a remote frame with the same identifier have the same priority in the TX-Scan.

CAN Protocol	Protocol Selection			Priority Value						
	XLF (T0[30])	FDF (T0[31])	XTD (T0[29])	31 down to 21	20	19	18	17	16 down to 1	0
Classic CAN	0	0	0	T0[28:18] (Base ID[10:0])	0 (RTR)	0 (XTD)	0 (FDF)	0	16'b0	0

Classic CAN (extended ID)	0	0	1	T0[28:18] (Base ID[10:0])	1 (SRR)	1 (XTD)	T0[17:0] (Identifier Extension[17:0])			0 (RTR)
CAN FD	0	1	0	T0[28:18] (Base ID[10:0])	0 (RRS)	0 (XTD)	1 (FDF)	0	16'b0	0
CAN FD (extended ID)	0	1	1	T0[28:18] (Base ID[10:0])	1 (SRR)	1 (XTD)	T0[17:0] (Identifier Extension[17:0])			0 (RRS)
CAN XL	1	X	X	T0[28:18] (Priority ID[10:0])	T0[17] (RRS)	0 (XTD)	1 (FDF)	1 (XLF)	16'b0	0

The selection of the TX message is done by looking at the queues in the following order, TX Priority Queue slots from 0 to 31, then the TX FIFO Queues are scanned from 0 to 7. The process of TX message selection will keep the two highest priority messages over the full scan.
IMPORTANT: When two or more TX messages have the same priority value, the first one will always be kept as the one to be sent first.
IMPORTANT: At initial time, when several TX FIFO Queues are started at the same time, the first TX messages may not be in the right order. Due to the scanning order (TX Priority Queue slots 0 to 31 and then TX FIFO Queue 0 to 7) and if the system memory latency is high, by the time the last TX descriptor is uploaded to the L_MEM, some TX messages may have been already scanned for the highest priority and sent to the PRT. This is normal behavior and will last only for the first TX messages.

As soon as a new Header Descriptor is available in the L_MEM, it will be arbitrated automatically if the TX-Scan process is not running. In case that a new TX Header Descriptor is stored in the L_MEM while the TX-Scan is running, the TX-Scan goes up to the end and will restarted to take this new descriptor into account.
Before starting the TX-Scan, the list of all potential candidates (valid) on the L_MEM is stored locally. This process will ensure a proper definition of the best candidates after a complete scan at the time it is done.
The duration of the TX-Scan mainly depends on the access time to the L_MEM and the number of TX FIFO Queues and TX Priority Queue Slots. Here is the list of parameters that will drive the overall time:

- The number of TX FIFO Queues being active at the same time
- The number of TX Priority Queue slots active at the same time
- The L_MEM read latency to fetch one single word

The processing time for one TX -Scan run can be defined as:
TX-Scan processing time (us) = Lr * (Nbfq + Nbpqs) * (1/CLK (MHz) where
Nbfq = Number of TX FIFO Queues active, Nbpqs = Number of TX Priority Queue slots active, Lr = read latency from L_MEM defined in number of *CLK* clock cycles

In any cases, when a new TX message is scheduled for transmission and it has the highest priority, the maximum delay to have this message selected by the TX-SCAN depends on, the maximum number of TX FIFO Queue and TX priority Queue running concurrently at that time. Considering a maximum of 8 TX FIFO Queues and 32 TX Priority Queue slots running at the same time this leads to (considering the previous formula):
Max TX SCAN duration (us) = Lr * 40 * (1/CLK (MHz).

As an example, CLK = 160MHz, Lr = 10 cycles leads to a Max delay TX message selection equal to 2.5 us. It is important to note that, in case of a TX message already being sent, the newer highest priority message will have to wait for the current one to finish. Thus, the overall delay to have this message on the CAN bus may change according to the CAN protocol, the payload data size and bit rate.

To ensure the continuity of a TX message, it is important to note that regarding TX FIFO Queues, the current and the next TX Header Descriptor for a given FIFO are loaded in the L_MEM. This assumption is valid only if the two TX descriptors are valid. Thus, if several TX messages in the same FIFO have the highest priority over the others, they will be sent back-to-back. For the TX priority Queue, things are different as one TX message is stored per slot. Only the TX message defined as active in a slot is considered at any time.

Two internal buffers are used to hold the TX descriptors in order to send TX messages in a row. One is holding the current TX descriptor to be sent right away to the PRT while the other stores the TX descriptor for the next message. It is important to note that the TX descriptors selected are the result of one TX-Scan. If any new events like, a message sent or a new message to be sent occurs, the two candidates may not be right ones. In this case the TX descriptors already buffered may need to be changed by some new ones. The change is performed step by step, to always have the highest priority message of one TX-Scan run available in one of the two local buffers. The previous TX descriptor with the highest priority is kept in one of the two local buffers while the new highest one is replacing the other. This procedure is repeated, if required, to change the second highest priority message.

The event of any new TX descriptor being loaded and available in the L_MEM triggers a TX-SCAN run. The TX descriptors describing TX messages can only considered by the TX-SCAN if they are available in L_MEM, see **TX_FQ_DESC_VALID** and **TX_PQ_DESC_VALID** registers.

To prepare the next TX descriptor and to react properly according to the results of the data being sent, there will be several actions to perform:

- The TX-SCAN computes the two next potential candidates, without considering the TX descriptor set as current in one of the two local buffers (the one with the highest priority). As soon as they are identified, the information related to the source of the two next highest priority messages is stored locally.

- The first candidate is compared to the one already in local buffer and has the lowest priority (the one with the highest priority is kept for the next transmission to come). If the first candidate computed is already in one of the two local buffers, nothing needs to be done. If

this is not the case, it is uploaded to provide the next highest priority TX message and will replace the one having the lowest priority in the local buffers. This is mandatory to ensure that there is always a valid TX descriptor with high priority to provide to the PRT, at any time. This is valid, even if the highest priority TX descriptor in local buffer may, at this time, not be the one with the highest priority. As soon as the first candidate is loaded in the local buffer, it may become the current one if it has the highest priority or the next one otherwise. It may happen that, while loading the first candidate, the current one is used as the next TX message. Nothing can prevent such scenario and either the one with the highest priority is sent first or at the second place.

- The second candidate is compared with the one previously defined as the current one. If the second candidate computed is already in one of the two local buffers, nothing needs to be done. If this is not the case, it is uploaded to provide the second highest TX message. In this particular case, the second candidate overwrites the other local buffer.

Here below is the flow chart of the TX-SCAN process.

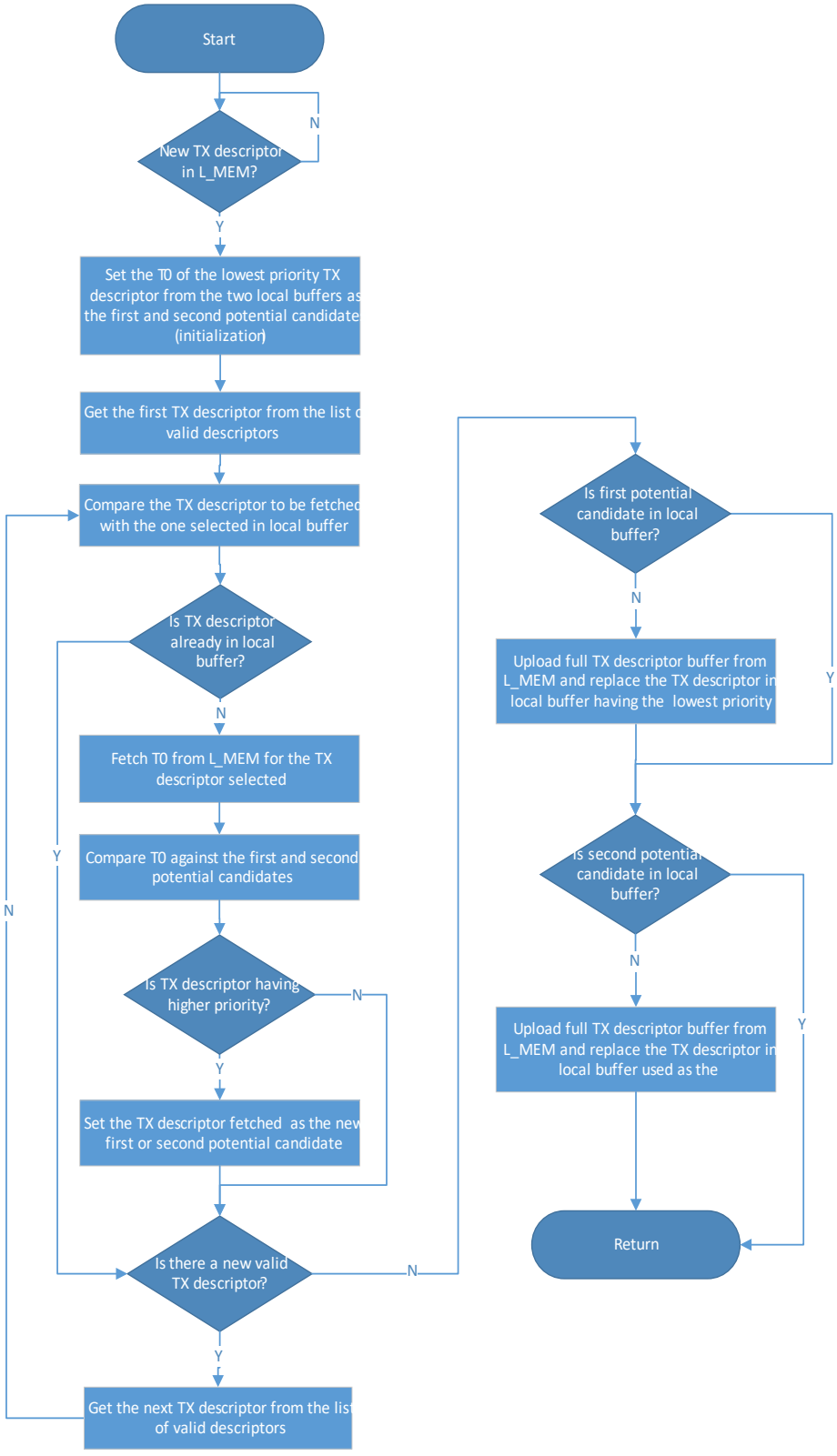

Figure: TX-Scan data flow

According to the status of the message sent, there will be two different actions:

- The TX message is sent successfully: the next candidate is considered as the current message. If no new TX descriptor is available in the L_MEM, the next candidate to be fetched is already known, otherwise a new TX-SCAN run is launched.

- The TX message is not sent successfully: The first next candidate is compared with the current one being not successful. If the current candidate has a higher priority, it will be considered as the one to select, otherwise the other candidate is used instead. On top of it, the re-transmission counter defined in the **MH_CFG** register will limit the number of possible trials for the same message. If the counter exceeds the limit, the current candidate will no longer be considered, even it has the highest priority. The TX message is skipped, and the next candidate is taken instead. If the counter does not reach the maximum value defined and a new message is taken instead, the counter is reset to 0.

Some TX-SCAN scenarios are described here below with the following assumptions:

- Three TX FIFO Queues and 3 TX Priority Queue slots are defined
- The TX messages are sent without any pause (no RX message received)
- For the sake of simplicity, only the ID in T0 is used as the main criteria to select the TX message to be sent
- A TX message is defined per TX descriptor (only Header Descriptor are defined)

The first scenario describes the TX-SCAN based only on TX FIFO Queues running and considering every message as sent successfully.

Descriptor number used by TX FIFO QUEUE	TX FIFO QUEUES		
	0	1	2
	ID	ID	ID
-	-	-	-
N	-	18	-
N+1	-	5	-
N+2	-	40	-
N+3	-	30	10
N+4	1	1	38
N+5	0	110	80
N+6	4	24	20
N+7	7	6	6
N+8	11	4	8
N+9	20	29	15
N+10	100	50	39
-	-	-	-

ID: TX Header Descriptor ID (one TX message per TX descriptor)

TX-SCAN run	TX Header Descriptor in L_MEM		CAN BUS	TX-SCAN results
	TX FIFO QUEUES	TX PRIORITY		

							QUEUE SLOTS						
	0		1		2		0	1	2				
	CID	NID	CID	NID	CID	NID	CID	CID	CID	IDIP	MSG Result	NIDMNS	NIDMS
0	-	1	18	-	10	-	-	-	-	-	ok	-	1
1	1	0	18		10		-	-	-	1	ok	1	0
2	0	4	18		10		-	-	-	0	ok	0	4
3	4	7	18		10		-	-	-	4	ok	4	7
4	7	11	18		10		-	-	-	7	ok	7	10
5	11		18		10	38	-	-	-	10	ok	10	11
6	11	20	18		38		-	-	-	11	ok	11	18
7	20		18	5	38		-	-	-	18	ok	18	5
8	20		5	40	38		-	-	-	5	ok	5	20
9	20	100	40		38		-	-	-	20	ok	20	38
10	100		40		38	80	-	-	-	38	ok	38	40
11	100		40	30	80		-	-	-	40	ok	40	30
12	100		30	1	80					30	ok	30	1
13	-	-	-	-	-	-	-	-	-	-		-	-

CID: Current TX Header Descriptor ID to consider for TX-SCAN
NID: Next TX Header Descriptor ID to consider for TX-SCAN
NIDMS: Next TX Header Descriptor ID if Message Successful
NIDMNS: Next TX Header Descriptor ID if Message Not Successful
IDIP: TX Header Descriptor ID In Progress
The second scenario describes the TX-SCAN based on TX FIFO Queues and TX Priority Queue slots running and considering every message as sent successfully.

Descriptor number used by TX FIFO QUEUE	TX FIFO QUEUES		
	0	1	2
	ID	ID	ID
-	-	-	-
N	-	18	-
N+1	-	5	-
N+2	-	40	-
N+3	-	30	10
N+4	1	1	38
N+5	0	110	80
N+6	4	24	20
N+7	7	6	6
N+8	11	4	8

N+9	20	29	15
N+10	100	50	39
-	-	-	-

ID: TX Header Descriptor ID (one TX message per TX descriptor)

TX-SCAN run	TX Header Descriptor in L_MEM									CAN BUS		TX-SCAN results	
	TX FIFO QUEUE						PRIORITY QUEUE SLOTS						
	0		1		2		0	1	2				
	CID	NID	CID	NID	CID	NID	CID	CID	CID	IDIP	MSG Result	NIDMNS	NIDMS
0	-	1	18	-	10	-	-	-	-	-	ok	-	1
1	1	0	18		10		-	-	-	1	ok	1	0
2	0	4	18		10		-	-	-	0	ok	0	4
3	4	7	18		10		-	1	-	4	ok	1	1
4	7		18		10		-	1	-	1	ok	1	7
5	7	11	18		10		-	-	12	7	ok	7	10
6	11		18		10	38	5	-	12	10	ok	5	5
7	11		18		38		5	-	12	5	ok	5	11
8	11	20	18		38		-	-	12	11	ok	11	12
9	20		18		38		-	-	12	12	ok	12	18
10	20		18	5	38		-	-	-	18	ok	18	5
11	20		5	40	38		-	-	-	5	ok	5	20
12	20	100	40		38					20	ok	20	38
13	-	-	-	-	-	-	-	-	-	-	-	-	-

CID: Current TX Header Descriptor ID to consider for TX-SCAN
NID: Next TX Header Descriptor ID to consider for TX-SCAN
NIDMS: Next TX Header Descriptor ID if Message Successful
NIDMNS: Next TX Header Descriptor ID if Message Not Successful
IDIP: TX Header Descriptor ID In Progress
The third scenario describes the TX-SCAN based on TX FIFO Queues and TX Priority Queue slots running and considering successful and not successful messages with re-transmission counter set to 1.

Descriptor number used by TX FIFO QUEUE	TX FIFO QUEUES		
	0	1	2
	ID	ID	ID
-	-	-	-

N	-	18	-
N+1	-	5	-
N+2	-	40	-
N+3	-	30	10
N+4	1	1	38
N+5	0	110	80
N+6	4	24	20
N+7	7	6	6
N+8	11	4	8
N+9	20	29	15
N+10	100	50	39
-	-	-	-

ID: TX Header Descriptor ID (one TX message per TX descriptor)

TX-SCAN run	TX Header Descriptor in L_MEM									CAN BUS		TX-SCAN results	
	TX FIFO QUEUE						PRIORITY QUEUE SLOTS						
	0		1		2		0	1	2				
	CID	NID	CID	NID	CID	NID	CID	CID	CID	IDIP	MSG Result	NIDMNS	NIDMS
0	-	1	18	-	10	-	-	-	-	-	ok	-	1
1	1	0	18		10		-	-	-	1	ok	1	0
2	0	4	18		10		-	-	-	0	ok	0	4
3	4	7	18		10		-	1	-	4	nok	1	1
4	4	7						1	-	1	ok	1	4
5	4	7	18		10		-	-	-	4	nok	4	7
6	4	7	18		10		-	-	12	4	ok	7	7
7	7	11	18		10		5	-	12	7	ok	5	5
8	11		18		10		5	-	12	5	ok	5	10
9	11		18		10	38	-	-	12	10	ok	10	11
10	11	20	18		38		-	-	12	11	nok	11	12
11	11	20	18		38		-	-	12	11	nok	12	12
12	20		18		38		-	-	12	12	ok	12	18
13	-	-	-	-	-	-	-	-	-	-	-	-	-

CID: Current TX Header Descriptor ID to consider for TX-SCAN
NID: Next TX Header Descriptor ID to consider for TX-SCAN
NIDMS: Next TX Header Descriptor ID if Message Successful
NIDMNS: Next TX Header Descriptor ID if Message Not Successful
IDIP: TX Header Descriptor ID In Progress
Some debug registers are used to monitor the activity of the TX-Scan:

- The **TX_SCAN_FC** register provides the 2 best candidates selected from the previous TX-Scan run as well as the 2 best candidates for the current run

- The TX_SCAN_BC register gives all the relevant information (The TX FIFO Queue number and TX descriptor offset in that Queue or the TX Priority Queue slot number) regarding the two best candidates uploaded in the local buffer and ready to be sent to the PRT
- The TX_FQ_DESC_VALID register identifies which TX descriptor is valid, uploaded in the L_MEM and belonging to the list of potential candidates for the TX-Scan. The information displayed in that register covers for a given TX FIFO Queue, the current and the next TX descriptors of a queue that may be loaded in the L_MEM and valid
- The TX_PQ_DESC_VALID register provides the information of the slots of the TX Priority Queue loaded in the L_MEM and valid (ready for the TX-Scan)

1.4.5.22 TX Filter

To ensure only declared TX messages can go through, the MH provides to the SW a way to define TX acceptance filters. Only the TX messages being filtered are considered for the arbitration process. There is the option to enable or disable this TX filtering process (see TX_FILTER_CTRL0.EN bit register) and so to leave all TX messages to go through or not. Several TX filter elements are defined and processed to determine if the TX message is accepted or rejected. The TX_FILTER_CTRL0 control register defines how the TX filter elements are used, either standalone or combined.

A TX filter element uses reference values to compare with the TX message header data. The selection of the bit field to do the comparison can be configured for every TX filter element. Up to 16 TX filter elements can be defined and apply to every TX message when fetched from the L_MEM. There is no way to define those filters only for some specific queues. They apply to all TX messages whatever the TX FIFO Queues and TX Priority Queue slots.

When a TX filter error occurs, the faulty TX message is acknowledged with the status report "message rejected by TX filter". Thus, the SW is able to identify which one has been rejected, while scanning the TX descriptors from the TX FIFO Queues or TX Priority Queue slots. In order to determine the one being faulty and to avoid waiting for the TX descriptor, the TX_FILTER_ERR_INFO register provides the relevant information. The TX_FILTER_ERR_INFO.FQ when set to 1 defines a faulty TX message from a TX FIFO Queue otherwise from the TX Priority Queue. The FIFO Queue number is defined with the TX_FILTER_ERR_INFO.FQNS_PQS[3:0] bit field and the slot number with the TX_FILTER_ERR_INFO.FQNS_PQS[4:0] bit field.

1.4.5.22.1 Global configuration

To protect the setting of those TX filter elements, registers assigned to the configuration are protected, they can only be accessed in write Privileged mode. Only the required application can modify the TX filter setting.

A global TX filter configuration register can be used to define, if the TX messages are accepted or rejected on match using the TX_FILTER_CTRL0.MODE bit register.

To notify the system that a TX message is rejected, a *TX_FILTER_IRQ* interrupt is generated. It is possible to enable and disable the TX filter interrupt using the **TX_FILTER_CTRL0.IRQ_EN** bit register. On top of it, the faulty TX descriptor is acknowledged immediately with the status rejected by TX filter.

The TX filter elements can be enabled or disabled independently from each other thanks to the **TX_FILTER_CTRL1.VALID[15:0]** bit registers.

In order to define the type of data to be compared with, either the VCID or SDT, the **TX_FILTER_CTRL1.FIELD[15:0]** is used. This register bit field is relevant for the CAN XL protocol only.

The definition of those TX filter elements is done through the setting of registers. Compared to the RX filter, the TX filter does not require to have access to the L_MEM, settings are done only in registers, see **TX_FILTER_CTRL0**, **TX_FILTER_CTRL1**, **TX_FILTER_REFVAL{n}** (n € {0, 1, 2, 3}) registers. It is assumed that the TX filter elements once defined are statics and won't change over time while the MH is running.

Refer to the TX filter registers for a more detailed description of the TX filters.
As the MH can support several CAN protocols, different options are possible on the TX filter, see the next sections for more details.

1.4.5.22.2 Classical CAN

All Classical CAN TX messages are either accepted or rejected, see **TX_FILTER_CTRL0.CC_CAN** bit register. There is no other option for such Classical CAN protocol. The TX filter elements are only used for the CAN XL protocol.

1.4.5.22.3 CAN FD

All CAN FD messages are either accepted or rejected, see **TX_FILTER_CTRL0.CAN_FD** bit register. There is no other option for the CAN FD protocol. The TX filter elements are only used for the CAN XL protocol.

1.4.5.22.4 CAN XL

Several options are possible to define the TX filter elements.
Two global modes are defined for the overall TX filter elements, either Allow or Reject on match, see **TX_FILTER_CTRL0.MODE** bit field register. When the Mode is configured to "Allow" (White List Approach), which is set by default, a TX message is only transmitted, if there is a match on one of the TX filter elements. When the Mode is configured to "Reject" (Black List Approach) a TX message is only transmitted if there is no match at all.

Every TX filter element is provided with a reference value to be compared with and which bit field in the message header to be used, either VCID or SDT.

There are three different options on how to define a TX filter element with the previous definition:
Option 1:
In normal mode, the reference value defined in **TX_FILTER_REFVAL{n}.REF_VAL3, TX_FILTER_REFVAL{n}.REF_VAL2, TX_FILTER_REFVAL{n}.REF_VAL1** and **TX_FILTER_REFVAL{n}.REF_VAL0** (n € {0, 1, 2, 3}) registers, is compared with either SDT or VCID. If e.g. **TX_FILTER_REFVAL{n}.REF_VAL0** is defined to be compared to VCID and **TX_FILTER_REFVAL{n}.REF_VAL1** to be compared to SDT than a CAN message will get a match if one of the two values matches.
Any of the 16 TX filter elements can be used to compare the VCID or the SDT value, refer to the control bit register **TX_FILTER_CTRL1.FIELD[n]** (n € {0, 1, 2, 3, ..., 15}) (when the bit n is set to 1, the SDT bit field is selected for the TX filter element otherwise VCID).
The Tx filter n is defined as valid or not valid using the **TX_FILTER_CTRL1.VALID[n]** (n € {0, 1, 2, 3, ..., 15}) bit register. If the TX filter 1 is not considered, just set the **TX_FILTER_CTRL1.VALID[1]** to 0. An example of the option 1 is given in the table below.
For such configuration, the **TX_FILTER_CTRL0.MASK[n]** (n € {0, 1, 2, ..., 7}) bit register must be set to 0 for the given TX filter element pair (n and n+1 assuming n is even)
To allow single compare value, meaning one reference value for one match, the **TX_FILTER_CTRL0.COMB[n]** must set to 0 for TX filter element n and n+1 (n being even). In this mode, there will be always TX filter element n and n+1 available.

Option 2:
Based on the normal mode and to increase the possible filtering options, two TX filter elements can be combined to allow VCID and SDT to be checked as only one filter. However, both values must be identical. As only a pair of TX filter elements reference values can be combined, **TX_FILTER_REFVAL{n}.REF_VAL0** and **TX_FILTER_REFVAL{n}.REF_VAL1** or **TX_FILTER_REFVAL{n}.REF_VAL2** and **TX_FILTER_REFVAL{n}.REF_VAL3** (n € {0, 1, 2, 3}) can be used.
The selection of the bit field value to be compared with is defined by the **TX_FILTER_CTRL1.FIELD[n]** (n € {0, 1, 2, 3, ..., 15}) bit register. The setting of this register is identical to the option 1.
Only two adjacent TX filter elements can be configured as combined, using the **TX_FILTER_CTRL0.COMB[n]** (n € {0, 1, 2, ..., 7}) bit register. When set to 1, TX filter n and n+1 (n being even) are combined. As an example, for the **REF_VAL0/REF_VAL1** in the **TX_FILTER_REFVAL0** register, the **TX_FILTER_CTRL0.COMB[0]** bit must be set to 1.
For such configuration, the **TX_FILTER_CTRL0.MASK[n]** (n € {0, 1, 2, ..., 7}) bit register must be set to 0 for the given TX filter element pair (n and n+1 assuming n is even).
The TX filter element n and n+1 which are combined (n being even) need to be set as valid (set to 1) using the **TX_FILTER_CTRL1.VALID[n]** and **TX_FILTER_CTRL1.VALID[n+1]** (n € {0, 1, 2, 3, ..., 15}) bit register. This means that combined TX filter elements require two bits to be set in the **TX_FILTER_CTRL1** register.

Option 3:

In order to compare a range of values, a reference value and a mask are required. To provide such option, two TX filter elements TX filter n and n+ 1 (n is even) can be paired in a way that one is the value to be compared with while the other is the mask. As only a pair of TX filter elements reference values can be combined, **TX_FILTER_REFVAL{n}.REF_VAL0** and **TX_FILTER_REFVAL{n}.REF_VAL1** or **TX_FILTER_REFVAL{n}.REF_VAL2** and **TX_FILTER_REFVAL{n}.REF_VAL3** (n € {0, 1, 2, 3}) can be used.
In order to set one of the two reference value as a mask, the appropriate bit in the **TX_FILTER_CTRL0.MASK[n]** (n € {0, 1, 2, ..., 7}) bit register must be set to 1. As an example, the **TX_FILTER_CTRL0.MASK[0]** set to 1 is referring to the pair **TX_FILTER_REFVAL0.REF_VAL0** and **TX_FILTER_REFVAL0.REF_VAL1**. In such configuration, the second reference value is the mask, hence the **REF_VAL1** when considering the **REF_VAL0**/**REF_VAL1** pair.
The bit field to be compared with is defined by the **TX_FILTER_CTRL0.FIELD** bit field register. For the above example the **TX_FILTER_CTRL0.FIELD[0]** must be set either to 1 for SDT or 0 for VCID. When **TX_FILTER_CTRL0.MASK[0]**=1 then **TX_FILTER_CTRL0.FIELD[1]** is ignored by the MH, **REF_VAL1** is interpreted as a mask only.
Only two adjacent TX filter elements can be configured as combined, using the **TX_FILTER_CTRL0.COMB[n]** (n € {0, 1, 2, ..., 7}) bit register. When set to 1, TX filter n and n+1 (n being even) are combined. As an example, for the **TX_FILTER_REFVAL0.REF_VAL0** and **TX_FILTER_REFVAL0.REF_VAL1** bit field, the **TX_FILTER_CTRL0.COMB[0]** bit must be set to 1.
It is essential to enable this pair of TX filter element by setting the appropriate bits in the **TX_FILTER_CTRL1** register. As an example, the **TX_FILTER_CTRL1.VALID[1:0]** set to 2'b11 will enable the **TX_FILTER_REFVAL0.REF_VAL0** (used as reference value) and **TX_FILTER_REFVAL0.REF_VAL1** (used as a mask).

The following example shows 4 reference values defined in the **TX_FILTER_REFVAL0** register, others behave the same.

Option 1 (single): A TX filter element uses only one reference value and one bit field (VCID or SDT)
Option 2 (Combined with matches): **REF_VAL0** and **REF_VAL1** are combined to provide a TX filter element that is able to compare VCID and SDT in the same filter. The same holds for **REF_VAL2** and **REF_VAL3**.

Option 3 (Combined with mask and value): same as Option 2 with the difference that, **REF_VAL0** is still the reference value to compare with (either VCID or SDT) but the **REF_VAL1** is the mask to apply.

In the table below, the 3 options are depicted.
Table: TX filter Element options

Reference value	Option 1 (single)	Option 2 (Combined with matches)	Option 3 (Combined with mask and value)
REF_VAL0	REF_VAL0 = (VCID or SDT)	REF_VAL0 = VCID or SDT AND REF_VAL1 = VCID or SDT	REF_VAL0 (value) = REF_VAL1 (mask) AND (VCID or SDT)
REF_VAL1	REF_VAL1 = (VCID or SDT)		
REF_VAL2	REF_VAL2 =	REF_VAL2 = VCID or SDT	REF_VAL2 (value) = REF_VAL3 (mask)

Reference value	Option 1 (single)	Option 2 (Combined with matches)	Option 3 (Combined with mask and value)
	(VCID or SDT)	AND **REF_VAL3** = VCID or SDT	AND (VCID or SDT)
REF_VAL3	**REF_VAL3** = (VCID or SDT)		

1.4.5.23 RX Filter

The RX filtering provides a way to reject or accept RX messages to the SW as well as to write those messages to a defined RX FIFO Queue.

Up to 255 RX filters can be defined. The RX filter is defined using a RX filter element (word of 32bit) associated with up to 2 pairs of reference(32bit)/mask(32bit) values. Those RX filter elements are continuous in the L_MEM and will be parsed one after the other. Regarding the reference/mask pairs they are defined after the RX filter element list in the L_MEM, up to 256 pairs can be declared.

To be flexible, a RX filter is made of up to 2 comparisons, each one using a reference value (32bit) and mask value (32bit) to apply on one of the incoming header message data words (R0, R1 or R3) from the PRT. A reference/mask pair can apply to any of the RX filter elements.

It is a SW task to define and write the RX filter elements and reference/mask pairs in the L_MEM. There is no direct access to it through the MH. The SW would need to access the L_MEM directly to program the relevant RX filter elements and reference/mask pairs.

The process of RX filtering is started as soon as the first 32bit word from the PRT is received, meaning R0. If the RX filter is fetching a filter element which requires R1, the process is on hold waiting for the 32bit word to be available. The same applies with R2 if only R0 and R1 are available.

The minimum time dedicated to the RX filtering is defined by the reception of an RX message when it is sent back-to-back. The timing window in this case is defined by the reception of two first 32bit word from the PRT (R0/R1) for the current RX message to the next two 32bit words (R0/R1) of the next message.

1.4.5.23.1 Global configuration

The register **RX_FILTER_CTRL** is used to set the general configuration for all RX Filters, a write access in Privileged mode is required. Once the MH is started (**MH_CTRL.START** = 1), the register is write-protected. Here below is the list of RX filter configuration setting:

- The number of RX filter elements is defined in the **RX_FILTER_CTRL.NB_FE[7:0]** bit field register

- non-matching CAN frames can be accepted or rejected by configuring the **RX_FILTER_CTRL.ANMF** bit register to 1. If non-matching frames are accepted, they are stored in the default RX FIFO Queue defined by the **RX_FILTER_CTRL.ANMF_FQ[2:0]** bit field register

- If the RX filtering is taking too much time, the data stored in the RX DMA FIFO may lead to an overflow. There is the option to allow the reception of such frames to the default RX FIFO Queue (defined by the **RX_FILTER_CTRL.ANMF_FQ[2:0]** bit field register). This feature is only enabled when setting the **RX_FILTER_CTRL.ANFF** bit field register to 1. The **RX_FILTER_CTRL.THRESHOLD[4:0]** defines the level in the RX DMA FIFO that has to be reached before sending the non-filtered frames are sent to the default RX FIFO Queue

- The default RX FIFO Queue number, to write RX message data when non-matching frames OR non-filtered frames are accepted, is defined in the **RX_FILTER_CTRL.ANMF_FQ[2:0]** bit field register. This default RX FIFO Queue value is considered if either the **RX_FILTER_CTRL.ANMF** bit or the **RX_FILTER_CTRL.ANFF** bit is set to 1. It is essential to enable and start this default RX FIFO Queue prior starting the PRT

1.4.5.23.2 Reference and Mask Pair

Two comparisons in the RX Filter Element can be defined. Each comparison requires a reference value (REFm) and a mask (MSKm) (m € {0, 1, 2, ..., 255}). The reference and mask value are 32bit data. The 32bit reference value is compared with the first (R0), the second (R1) or the third (R2) word (32bit) of the RX message header (coming from the PRT) after applying the 32bit mask value.

The formula to compute a match is defined as (Ri being R0 or R1 or R2; & = bitwise AND) :
MATCHm = 1 When (Ri & MSKm) = REFm Else 0
For one single comparison (REFm,MASKm) in a RX Filter Element:
MATCH = MATCHm
For a RX filter element with two comparisons (REFm,MSKm) and (REFn,MSKn) we have:
MATCH = MATCHm AND MATCHn

Here are the setting of the (REFm,MSKm) and/or (REFn,MSKn), to mask and compare a bit value:
To mask the bit Ri[j]:
- Set MSK(m/n)[j] = 0
- Set REFm(m/n)[j] = 0

To compare the bit Ri[j] to a defined bit value:
- Set MSK(m/n)[j] = 1
- Set REF(m/n)[j] = 1 or 0

Here are two examples of RX filters:

To accept only CAN FD and CAN CC frames (in other word to reject only CAN XL frames), only the XLF bit in the first header matters (R0[30]). This bit must always be set to 0 to indicate CAN FD and CAN CC frames. This leads to the following mask and reference value:
MASKm[30] = 1, other bits being set to 0
REFm[30] = 0, other bits being set to 0

To reject CAN FD and CAN CC frames (in other word to accept only CAN XL frames), only the XLF bit in the first header matters (R0[30]). This bit must always be set to 1 to indicate CAN XL frames. This leads to the following mask and reference value:
MASKm[30] = 1, other bits being set to 0
REFm[30] = 1, other bits being set to 0

The reference value and the mask are defined as a pair of two consecutive words of 32bit in L_MEM, starting with the REFm. Up to 256 pairs can be defined for the overall RX filter elements. All the pairs of reference value and mask are appended after the RX filter elements section in L_MEM, see MH Software Interface chapter. Any of the 256 reference/mask pairs can be selected for a given RX filter element.

The RX filter element will use an index to identify the position of the pairs to be used. The first pair is having the index 0, the second the index 1 and so on. The RX Filter Element is then referring to this index in the bit field CREFI0 and CREFI1 (see RX Filter Element Definition chapter), to identify the right pair to use.

1.4.5.23.3 RX Filter Element Definition

For every filter element it is possible to:

- Define which RX FIFO Queue to use, if an RX message is accepted or rejected on match
- Generate an interrupt when a filter matches, triggering the signal *RX_FILTER_IRQ* to the system
- Define, if the expected RX message is defined in the blacklist (BLK bit field)
- Define up to 2 comparisons with the option to:
    - Select the word index in the header message to look at (limited to R0, R1 and R2), see WI0 or WI1. It is important to note that R2 is a header data for the CAN XL while a payload data in case of Classical CAN and CAN FD. In case a remote frame is detected, any filter looking at R2 will be discarded.

    - Define the reference value and mask index to perform the comparison, see CREFI0 or CREFI1 bit field
    - Reject or accept on match, see AR0 and AR1
    - Perform two comparisons for a match. When WI0 and WI1 are both different from 0, both comparisons are performed to know if there is a match. In such configuration,

the AR0 and AR1 must be identical. In case they are not, only the comparison 0 is performed to check for a match.

RX filter element is described in table below:
Table: RX filter element description

Filter Element	Section	Bit Field	Name	Description
FEn	Control	FEn[31:28]	FIFO[3:0]	RX FIFO Queue number to store the receive CAN data
		FEn[27]	IRQ	Interrupt: When set to 1 an interrupt is triggered to the system when a match is detected
		FEn[26]	BLK	BlackList: When set to 1 the BLK bit defined in the RX message header is set to 1
		FEn[25:24]	Reserved	
	Comparison 1 (only considered with comparison 0)	FEn[23]	Reserved	
		FEn[22]	AR1	See AR0 bit field description. Must always be equal to AR0.
		FEn[21:20]	WI1[1:0]	See WI0 bit field description
		FEn[19:12]	CREFI1[7:0]	See CREFI0 bit field description
	Comparison 0	FEn[11]	Reserved	
		FEn[10]	AR0	Accept or reject on match: when set to 1 the RX message is rejected on match otherwise accepted on match
		FEn[9:8]	WI0[1:0]	Word Index: provide the index of the RX message header word to compare, 1 for R0, 2 for R1 and 3 for R2 (CAN XL) / RD0 (Classic CAN and CAN FD). 3 is not considered as a valid index for: - Classical CAN Remote frame - Classical CAN frame with no payload data - CAN FD frame without payload data The comparison is then cancelled (next filter element is taken instead). When set to 0, no comparison is performed
		FEn[7:0]	CREFI0[7:0]	Comparison Reference Index: provide the index of the reference pair (value and mask) to be used for comparison. Up to 256 reference pairs can be defined. Only the reference pair number need to be set in this bit field.

The RX filter element can be used in two different modes:

- One comparison defined: Only the Comparison 0 is considered (WI0 > 0 and WI1 = 0)
- Two comparisons defined: Comparison 0 and comparison 1 are used (WI0 > 0 and WI1 > 0)

It is essential to understand that if the Comparison 1 is defined, this RX Filter element will not be considered if the Comparison 0 is not defined. In such case, the filter element is skipped.

Each time an RX message is received, the RX filtering is triggered and will start the following sequence:

- Fetch the first filter element from L_MEM. The start address of this first filter element is defined in **RX_FILTER_MEM_ADD.BASE_ADDR[15:0]** register

- If the conditions listed below are met, the RX filter element will be discarded, and the next filter element be fetched (if there is one available). In all other cases, go to next step:

  - WI0 = 0
  - WI0 = 3 and the frame is either a Classical CAN/CAN FD without payload or a Classical CAN remote frame.
  - WI1 = 3 and the frame is either a Classical CAN/CAN FD without payload or a Classical CAN remote frame.
  - WI0 > 0 and WI1 > 0 and AR0/AR1 not equal
- For the comparison 0, using the index CREFI0, the two words for the reference value and mask are fetched from L_MEM.

- The comparison 0 is done between the word defined by the index WI0 and the reference value/mask fetched earlier. According to the WI1 bit, several actions are taken:

  - WI1 set to 0 (Comparison 0 only): if there is a match, the RX filter will look at the AR0 bit to identify what to do with the RX message. It would then be accepted or rejected, and the RX filter stops. If there is no match, the RX filter keeps going with the next filter element, if there is one available, otherwise it stops.

  - WI1 > 0 (Comparison 0 and Comparison 1 expected): if there is a match on the comparison 0, the RX filter will wait for the result of the comparison 1 to decide what to do with the RX message. If there is no match on comparison 0, the RX filter keeps going with the next filter element, if there is one available, otherwise it stops. For the comparison 1, using the CREFI1, the two words for the reference value and mask are fetched from L_MEM.

- The comparison 1 is done between the word defined by the index WI1 and the reference value/mask fetched earlier. If there is a match, the RX filter will look at the AR0 bit to identify what to do with the RX message. It would then be accepted or rejected, and the RX filter stops. If there is no match, the RX filter keeps going with the next filter element, if there is one available, otherwise it stops.

1.4.5.23.4 RX Header Descriptor Updates

When an RX message is accepted, the index of the RX filter element which has accepted the message on match is logged inside the RX Header Descriptor. In case the RX message is rejected, no information is provided to the SW.

The filtering process writes some filtering information to three header data bit fields (see RX Message header definition chapter):

- The BLK bit in the header data of the RX message can be set by an RX filter element to indicate to the SW unexpected messages. When the SW parses the RX message Header Descriptor, it can easily identify a blacklisted message. Having the AR0 and AR0/AR1 set to 1 (RX message rejected on match) with the BLK bit has no meaning.

- The FIDX[7:0] bit field is used to provide the information of the filter element index which has been triggered for that message

- The FAB bit field is set to 1 when the RX filtering process did not complete before the RX DMA FIFO level gets above its programmed threshold. This bit is set, when such issue occurs, only if the **RX_FILTER_CTRL.ANFF** bit register is set to 1 and the **RX_FILTER_CTRL.THRESHOLD[4:0]** bit field is greater than 0 (threshold mechanism active)

- The FM bit field when set to 1, notifies the SW that there was a match on the RX filtering. When the FAB and FM bit fields are set to 0, the RX filtering process ends with no match

1.4.5.23.5 MH Behavior According to RX Filter Setting

The RX filter elements are sequentially read from the L_MEM. This process continues up to the point, where the RX filter result is known, and the message can either be accepted or rejected.
In case the RX filter selects an RX FIFO Queue that is not enabled, the incoming frame is considered as a non-matching frame and is discarded, the *RX_ABORT_IRQ* is set to the system.
In a normal situation, when the RX filter result arrives in time, the RX message data is processed according to RX filter result:

- In case of a match, it is sent to the appropriate RX FIFO Queue
- In case of a reject, the RX message is discarded

Non-matching frames can be accepted or rejected according to the **RX_FILTER_CTRL.ANMF** bit register configuration. If **RX_FILTER_CTRL.ANMF** bit is configured to accept non-matching frames, the RX message is sent to a default RX FIFO Queue, see **RX_FILTER_CTRL.ANMF_FQ[2:0]** bit field register.

A threshold can be defined on the RX DMA FIFO to manage not filtered frames, at a defined fill level of the RX DMA FIFO. The **RX_FILTER_CTRL.ANFF** must be set to 1 to activate the function and the **RX_FILTER_CTRL.THRESHOLD[4:0]** bit field defining the threshold value must be greater than 0. Once

activated, those frames are sent to a default RX FIFO Queue, see **RX_FILTER_CTRL.ANMF_FQ[2:0]** bit field register.

Here below is the summary of the RX filter behavior when considering **RX_FILTER_CTRL.NB_FE[7:0]**, **RX_FILTER_CTRL.ANMF** and **RX_FILTER_CTRL.ANFF** bit fields:

ANMF	NBFE[7:0]	ANFF	RX Filter status
0	0	X	All RX frames are rejected
1	0	X	All RX messages are accepted and are going to the default RX FIFO Queue defined by **RX_FILTER_CTRL.ANMF_FQ[2:0]**
0	> 0	0	Frames with match are going to RX FIFO Queues Frames with no match are rejected No threshold monitoring is performed during RX filtering
0	> 0	1	Frames with match are going to RX FIFO Queues Frames with no match are rejected Frames reaching the RX DMA FIFO level set in the **RX_FILTER_CTRL.THRESHOLD[4:0]** register and not filtered, are going to the default RX FIFO Queue defined in **RX_FILTER_CTRL.ANMF_FQ[2:0]** register
1	> 0	0	Frames with match are going to RX FIFO Queues Frames with no match are going to the default RX FIFO Queue defined by **RX_FILTER_CTRL.ANMF_FQ[2:0]** No threshold monitoring is performed during RX filtering
1	> 0	1	Frames with match are going to RX FIFO Queues Frames with no match are going to the default RX FIFO Queue defined by **RX_FILTER_CTRL.ANMF_FQ[2:0]** Frames reaching the RX DMA FIFO level set in the **RX_FILTER_CTRL.THRESHOLD[4:0]** register and not filtered, are going to the default RX FIFO Queue defined in **RX_FILTER_CTRL.ANMF_FQ[2:0]** register

The MH will manage the RX message differently if a default RX FIFO Queue is defined or not, with or without a threshold defined. The configurations being able to use the two last options are described below:

1) Threshold mechanism is not active (**RX_FILTER_CTRL.ANFF** bit set to 0 and **RX_FILTER_CTRL.NBFE[7:0]** > 0).
Two scenarios can occur:

- The RX filtering result is not known before receiving the first word of the next RX message and the amount of the CAN frame data is lower than the RX DMA FIFO. The next RX message is discarded with the *RX_ABORT_IRQ* interrupt triggered to the system.

- In case the amount of data received, while waiting the RX filtering result, does exceed the maximum RX DMA FIFO size, the RX message is discarded with the *RX_ABORT*_IRQ interrupt. The *DP_DO_ERR* interrupt is triggered to notify a RX DMA FIFO overflow.

2) Threshold mechanism is active (**RX_FILTER_CTRL.ANNF** bit set to 1 and **RX_FILTER_CTRL.NBFE[7:0]** > 0 and **RX_FILTER_CTRL.THRESHOLD[4:0]** value is greater than 0). This backup solution to avoid losing the RX message is possible due to the monitoring of the RX DMA FIFO level and a threshold configured in the **RX_FILTER_CTRL.THRESHOLD[4:0]** bit field register (see next section for more details).
Two scenarios can occur:

- The RX filtering result takes a very long time, and the frame size is large. Once the threshold is reached, the RX descriptor from the default RX FIFO Queue is fetched from the S_MEM. Then, the RX buffer address pointer defined in the RX descriptor is used to write the first burst of data to the default RX FIFO Queue.
- The RX filtering result takes a very long time, and the frame size does not reach the threshold value. The RX filtering keeps going whatever the new incoming frames. In case a new RX message is received, it is discarded with the *RX_ABORT_IRQ* interrupt. The *RX_FILTER_ERR* is also triggered to the system to identify that the RX filtering has gone over the second RX frame. This is an indicator which could explain why the new RX frame is rejected.

Here below is a table to summarize the different scenarios:

Frame Length	Filter result when Threshold reached	Filter result when next RX message arrives	Action of MH	Comments
Short	Not possible	Available	Store frame in RX FIFO Queue defined by RX filter result	Normal behavior
Short	Not possible	Not available	Discard next RX message	The overall processing time is too long due to a high number of filters and/or a high latency on the L_MEM, see calculation in Excel-Sheet [6].
Long	Available and ANFF = 0 or 1	Not considered	Store frame in RX FIFO Queue defined by RX filter result	Normal behavior
Long	Not available and ANFF = 0	Not considered	Continues RX filtering to get filter result	Within this case, the message is written in time to the right RX FIFO Queue or discarded due to data overflow on the RX DMA FIFO
Long	Not available and ANFF = 1	Not considered	Store frame in default RX FIFO Queue	The threshold must be set to a value that there is enough time to fetch the RX descriptor and to write burst data to S_MEM, see calculation in

Frame Length	Filter result when Threshold reached	Filter result when next RX message arrives	Action of MH	Comments
				Excel-Sheet [6].

1.4.5.23.6 Threshold computation

The MH uses the **RX_FILTER_CTRL.THRESHOLD[4:0]** only when the **RX_FILTER_CTRL.ANFF** bit is set to 1 and the threshold value is greater than 0. The threshold value to be configured by the user depends on the S_MEM latency, the CAN protocols supported and the CAN XL data bit rate. The RX DMA FIFO has a size of 32 words (128 byte).

Case 1: Only Classical CAN and/or CAN FD messages are received: The RX DMA FIFO is capable of storing a complete Classical CAN or CAN FD message. The feature **RX_FILTER_CTRL.ANFF** should not be used.
Note: When **RX_FILTER_CTRL.ANFF** bit is set to 1 and **RX_FILTER_CTRL.THRESHOLD[4:0]** bit field is set to 19 or larger, then the threshold will never be reached. This implicitly disables the threshold.

Case 2: CAN XL messages are received: CAN XL messages can be much longer than the RX DMA FIFO size. When the fill level of the RX DMA FIFO reaches the threshold, the MH will fetch the RX descriptor from the default RX FIFO Queue and will write the first burst of data stored in the RX DMA FIFO. For the case, that no RX filter result is available at the point in time the threshold is reached, this mechanism prevents a data overflow that would occur on the RX DMA FIFO and allows the reception of the message. It is then, up to the SW to filter this frame.

The threshold divides the time budget provided by the RX DMA FIFO into two parts: (1) time to do the RX Filtering, (2) Time to fetch the RX Descriptor from S_MEM and to write the first burst of data to S_MEM. The user should configure the threshold as large as possible, to give the RX Filtering enough time, but as low as necessary to be able to receive the message, in case RX filtering is not finished yet. The Excel-Sheet [6] calculates the optimal threshold value, which depends on the S_MEM latency and the CAN XL data bit rate.

1.4.5.23.7 RX Filter Processing Time

The number of accesses to evaluate an RX filter element is defined as follow:
Classical CAN and CAN FD:
One single word access (definition of the filter element) and 2 words for the mask and value to compare with
CAN XL:

same as Classical CAN / CAN FD or one single word and 2 reads of 2 words for the 2 mask and value to compare with

The RX filter element access time depends on the read latency Lr (defined in number of *CLK* clock cycles):
Classical CAN and CAN FD:
RX Filter element processing time (us) = ((Lr+2) + (Lr+1+2)) * (1/*CLK* (MHz))
CAN XL:
RX Filter element processing time (us) = ((Lr+2) + (Lr+1+2)) * (1/*CLK* (MHz)) with one comparison
RX Filter element processing time (us) = ((Lr+2) + 2*(Lr+1+2)) * (1/*CLK* (MHz)) with two comparisons

The overall RX filter time is computed as follow:
RX Filter processing time (us) = (Nbfe1c * ((Lr+2) + (Lr+1+2)) + Nbfe2c * ((Lr+2) + 2*(Lr+1+2))) * (1/*CLK* (MHz)) where
Nbfe1c = Number of filter element with 1 comparison, Nbfe2c = Number of filter element with 2 comparisons and the read latency Lr (defined in number of *CLK* clock cycles)

1.4.5.24 Local Memory Controller

The XCAND_MH_MEM_CTRL block is in charge of reading and writing the L_MEM through its AXI4 master interface *MEM_AXI* (compliant to AMBA 4 ARM Ltd protocol, see [5]).

The L_MEM Controller manages all requests and data transfers for the different blocks running concurrently:

- The writes of TX descriptors from the XCAND_DESC block
- The read of RX filter elements from the XCAND_RX_PATH
- The read of TX descriptor from XCAND_TX_PATH when a message has to be sent
- The read of TX descriptor from XCAND_TX_PATH for TX SCAN (selection of the highest priority TX message)

1.4.5.24.1 Local Memory Side Band Signals

It is assumed that the L_MEM provides safety measure to protect data. The safety protection implemented in the L_MEM could either report error when reading a data (Single Error Detection) or be able to correct it in some cases (Single Error Correction and Double Error Detection for example). To address those two options, two input side band signals denominated *MEM_SFTY_CE* and *MEM_SFTY_UE*, are provided with the *MEM_AXI* interface.
Here below are the expected behavior of those signals and the expected response on the *MEM_AXI* interface:

- Error Detection Only: When a corrupted data is read from the L_MEM, a pulse of one *CLK* clock cycle must be generated on the *MEM_SFTY_UE* input signal. The *MEM_AXI* interface must report a

SLVERR response on the *MEM_AXI* bus. The *MEM_SFTY_UE* input signal can be fully asynchronous to the *MEM_AXI* interface.

- Error Detection and Correction: When a corrupted data is read from the L_MEM but is corrected, a pulse of one *CLK* clock cycle must be generated on the *MEM_SFTY_CE* input signal. The *MEM_AXI* interface must report an OKAY response on the *MEM_AXI* bus. The *MEM_SFTY_CE* input signal can be fully asynchronous to the *MEM_AXI* interface.

1.4.5.24.2 Address Bus

The XCAND_MH_MEM_CTRL block is able to address up to 64Kbyte memory space (*MEM_AXI_AWADDR[15:0]* and *MEM_AXI_ARADDR[15:0]*).

The address burst value is always 32bit aligned.

1.4.5.24.3 Burst Size

The maximum number of bytes to transfer in each data transfer is fixed and set to 4. Any read or write transfer is always using 32bit.

As a consequence, the write strobe signals are not managed by the XCAND_MH_MEM_CTRL as all 4 bytes are always written.

1.4.5.24.4 Burst Length

The L_MEM Controller for the AXI read and write transfers supports INCR burst length 1, 2 and 8 considering an AXI 32bit data bus width.

The burst length from/to the L_MEM is defined based on the data type of information to be used. Here below are the expected burst lengths from/to the different sub-blocks:

- XCAND_MH_DESC: This sub-block writes the TX descriptor allocated to TX FIFO Queues and TX Priority Queue slots. The burst length is fixed and set to 8x32bit. There is no read access from this sub-module.

- XCAND_MH_RX: This sub-block reads the RX filter elements and reference/mask values to perform the RX message filtering. The burst length is set to 1x32bit for the RX filter element and 2x32bit for the reference/mask value. There is no write access from this sub-module.

- XCAND_MH_TX: This sub-block reads two types of information, the TX descriptor to be sent as the next candidate to the TX_MSG interface and part of the TX descriptors assigned to TX FIFO Queues and TX Priority Queue slots. A fixed burst length of 1x32bit is used for the TX SCAN and 8x32bit is used for the TX descriptor.

1.4.5.24.5 Outstanding

As the L_MEM can be shared across several X_CAN instances and many accesses are required for RX filtering and TX SCAN, 2 outstanding read transactions can support. As only a few writes are required from the MH point of view, only 1 outstanding write transaction is supported.

1.4.5.24.6 Burst Type

The only burst type supported is the burst incrementing INCR.
The WRAP/FIXED burst type is not supported.

1.4.5.24.7 Multi-region

The L_MEM Controller does not support multiple region interfaces, see [5] for more details.

1.4.5.24.8 Memory Attributes

The memory attributes for the read or write accesses to the L_MEM is Normal, Non-modifiable (Non-cacheable in AXI3) and Non-bufferable. No read-allocate nor No Write-allocate are expected on this interface and would be set to 0. This means MEM_AXI_AWCACHE[3:0] and MEM_AXI_ARCACHE[3:0] are set to 0b0000.

As a reminder, Non-bufferable means (See [5] for more details):

- The write response must be obtained from the destination.
- Read data must be obtained from the destination.
- Transactions are Non-modifiable
- Read and write transactions from the same ID to addresses that overlap must remain ordered.

As a reminder, Non-modifiable means:

- A Non-modifiable transaction must not be split into multiple transactions or merged with other transactions.
- In a Non-modifiable transaction, the parameters AxADDR, AxSIZE, AxLEN, AxBURST and AxPROT must not be changed.

1.4.5.24.9 Access Permissions

It is considered that any access is always defined as a Data, Secure and the operating mode is Unprivileged, see [5] for more details. Those setting cannot be changed by SW.

Therefore, any access from the MH which needs to be non-secure, must be managed with an external and dedicated logic attached to the MEM_AXI interface.

As an example, the RX filter elements and reference/mask can be stored in a secure area in the L_MEM, as a consequence the MEM_AXI_ARPROT[1] and MEM_AXI_AWPROT[1] are set to 0. Doing so,

the MH is able to read secure and non-secure data in the L_MEM, with the assumption that non-secure area is always accessible by a secure access. This means MEM_AXI_A(W/R)PROT[2:0] is set to 0b000.

1.4.5.24.10 Transaction ID

The L_MEM Controller builds the ID of every burst access based on the source of request. It provides a way to track which sub-block is doing the access at any time on the system bus.

For the AXI read interface, the MEM_AXI_ARID[1:0] defines the channel number as follow:
2'b00 => XCAND_MH_TX reads TX descriptor from L_MEM
2'b01 => XCAND_MH_TX read part of TX descriptor from L_MEM for TX SCAN
2'b10 => XCAND_MH_RX reads RX filter elements and reference/mask values from L_MEM
2'b11 => Reserved
For the AXI write interface, the MEM_AXI_AWID[0] defines the channel number as follow:
1'b0 => XCAND_MH_DESC writes TX descriptor to L_MEM
1'b1 => Reserved

1.4.5.25 Trace and Debug

1.4.5.25.1 Interrupts

For integration verification, it is possible to trigger functional and safety interrupts by SW. Here is the procedure:

1) Unlock the **DEBUG_TEST_CTRL** register, see section Lock Mechanism Protection in Register Protection chapter
2) Write the **DEBUG_TEST_CTRL.TEST_IRQ_EN** bit to 1 in Privileged mode
3) Once the access to the **INT_TEST0** and **INT_TEST1** registers are allowed (always accessible once opened), write 1 to the relevant bit to set the appropriate interrupt line.

Re-lock the access to the **INT_TEST0** and **INT_TEST1** registers. Step 1) and 2) needs to be done with **DEBUG_TEST_CTRL.TEST_IRQ_EN** bit set to 0 instead.

1.4.5.25.2 Hardware Debug Port

The 16bit HDP bus provides some visibility to internal signals to debug the MH. By default, there is no activity on the HDP bus.

To enable the toggling of the HW signal on the HDP bus, set the **DEBUG_TEST_CTRL.HDP_EN** bit to 1.
1) Unlock the **DEBUG_TEST_CTRL** register, see section Lock Mechanism Protection in Register Protection chapter
2) Write the **DEBUG_TEST_CTRL.HDP_EN** bit to 1 and the selected set to be monitored on the HDP using the **DEBUG_TEST_CTRL.HDP_SEL[2:0].** This access must be done in Privileged mode.

To disable the Hardware Debug Port monitoring, do the previous set with **DEBUG_TEST_CTRL.HDP_EN** bit to 0.
Up to 8 sets can be defined using the **DEBUG_TEST_CTRL.HDP_SEL[2:0]** bit field. When the value is set to n the set n is selected on the HDP bus.

INTERRUPTS:
The interrupt line assigned to the RX or TX FIFO Queue can be monitored individually. Therefore, it is possible to track the activity of the FIFO Queues while they are running. To allow the visibility of all MH interrupts, on the same HDP set, the TX FIFO Queues interrupt lines are gathered to only one single HW internal signal called *TX_FQ_IRQ_ORED* (the interrupts are 'ored'). The same is done on the RX FIFO Queues and so the interrupts are 'ored' to provide the HW internal signal *RX_FQ_IRQ_ORED*.

IMPORTANT: There are two possible sources to trig an interrupt (valid for TX_FQ_IRQ[7:0], RX_FQ_IRQ[7:0] and TX_PQ_IRQ interrupt lines): one is related to functional and the other one is from the **INT_TEST0** and **INT_TEST1** registers (for integration test only). Only the functional interrupt source is displayed on the HDP set. Therefore, when an interrupt is triggered, by a write access to either **INT_TEST0** or **INT_TEST1** register, it will not be visible on the HDP. Nevertheless, the interrupt line is properly set at the MH interface.

INTERFACES:
To ensure the traceability of the traffic going from and to the MH, the following interfaces can be monitored through one of the HDP sets:

- DMA_AXI interface (control signals) used to manage RX/TX descriptors and RX/TX message data
- MEM_AXI interface (control signals) used to manage TX descriptors for TX-Scan and RX filtering
- TX_MSG interface (control signals) used to transmit TX message from MH to PRT
- RX_MSG interface (control signals) used to receive RX message from PRT to MH

Here below are the description of sets available on the MH HDP bus.

HDP[15:0]	Set 0 (Interrupts)	Set 1 (RX and TX path)
15	*TX_FQ_IRQ[7]*	*CLK*
14	*TX_FQ_IRQ[6]*	*TX_FQ_IRQ_ORED*
13	*TX_FQ_IRQ[5]*	*RX_FQ_IRQ_ORED*
12	*TX_FQ_IRQ[4]*	*TX_PQ_IRQ*
11	*TX_FQ_IRQ[3]*	*RX_FILTER_ERR*
10	*TX_FQ_IRQ[2]*	*RX_FILTER_IRQ*
9	*TX_FQ_IRQ[1]*	*TX_FILTER_IRQ*
8	*TX_FQ_IRQ[0]*	*STATS_IRQ*
7	*RX_FQ_IRQ[7]*	*TX_ABORT_IRQ*
6	*RX_FQ_IRQ[6]*	*RX_ABORT_IRQ*
5	*RX_FQ_IRQ[5]*	*DP_SEQ_ERR*

HDP[15:0]	Set 0 (Interrupts)	Set 1 (RX and TX path)
4	*RX_FQ_IRQ[4]*	*DP_DO_ERR*
3	*RX_FQ_IRQ[3]*	*STOP_IRQ*
2	*RX_FQ_IRQ[2]*	*RESP_ERR[1]*
1	*RX_FQ_IRQ[1]*	*RESP_ERR[0]*
0	*RX_FQ_IRQ[0]*	*ENABLE*

HDP[15:0]	Set 2 (TX Scan)	Set 3 (MH-PRT Interface)
15	*FH_OFFSET[9]*	*CLK*
14	*FH_OFFSET[8]*	*TX_MSG_WVALID*
13	*FH_OFFSET[7]*	*TX_MSG_WUSER[1]*
12	*FH_OFFSET[6]*	*TX_MSG_WUSER[0]*
11	*FH_OFFSET[5]*	*TX_MSG_WREADY*
10	*FH_OFFSET[4]*	*TX_MSG_BVALID*
9	*FH_OFFSET[3]*	*TX_MSG_BUSER_STATUS[2]*
8	*FH_OFFSET[2]*	*TX_MSG_BUSER_STATUS[1]*
7	*FH_OFFSET[1]*	*TX_MSG_BUSER_STATUS[0]*
6	*FH_OFFSET[0]*	*TX_MSG_BREADY*
5	*FH_FQN_PQN[4]*	*RX_MSG_WVALID*
4	*FH_FQN_PQN[3]*	*RX_MSG_WUSER[2]*
3	*FH_FQN_PQN[2]*	*RX_MSG_WUSER[1]*
2	*FH_FQN_PQN[1]*	*RX_MSG_WUSER[0]*
1	*FH_FQN_PQN[0]*	*RX_MSG_WREADY*
0	*FH_PQ*	*ENABLE*

HDP[15:0]	Set 4 (Write AXI DMA Interface)	Set 5 (Read AXI DMA Interface)
15	*CLK*	*CLK*
14	*DMA_AXI_BID[0]*	*DMA_AXI_RID[1]*
13	*DMA_AXI_BVALID*	*DMA_AXI_RID[0]*
12	*DMA_AXI_BREADY*	*NA*
11	*DMA_AXI_BRESP[1]*	*DMA_AXI_RRESP[1]*
10	*DMA_AXI_BRESP[0]*	*DMA_AXI_RRESP[0]*
9	*DMA_AXI_WREADY*	*DMA_AXI_RREADY*
8	*DMA_AXI_WVALID*	*DMA_AXI_RVALID*
7	*DMA_AXI_WLAST*	*DMA_AXI_RLAST*

HDP[15:0]	Set 4 (Write AXI DMA Interface)	Set 5 (Read AXI DMA Interface)
6	*NA*	*DMA_AXI_ARID[1]*
5	*DMA_AXI_AWID[0]*	*DMA_AXI_ARID[0]*
4	*DMA_AXI_AWVALID*	*DMA_AXI_ARVALID*
3	*DMA_AXI_AWREADY*	*DMA_AXI_ARREADY*
2	*DMA_AXI_AWLEN[2]*	*DMA_AXI_ARLEN[2]*
1	*DMA_AXI_AWLEN[1]*	*DMA_AXI_ARLEN[1]*
0	*DMA_AXI_AWLEN[0]*	*DMA_AXI_ARLEN[0]*

HDP[15:0]	Set 6 (Write AXI MEM Interface)	Set 7 (Read AXI MEM Interface)
15	*CLK*	*CLK*
14	*MEM_AXI_BID[0]*	*MEM_AXI_RID[1]*
13	*MEM_AXI_BVALID*	*MEM_AXI_RID[0]*
12	*MEM_AXI_BREADY*	*NA*
11	*MEM_AXI_BRESP[1]*	*MEM_AXI_RRESP[1]*
10	*MEM_AXI_BRESP[0]*	*MEM_AXI_RRESP[0]*
9	*MEM_AXI_WREADY*	*MEM_AXI_RREADY*
8	*MEM_AXI_WVALID*	*MEM_AXI_RVALID*
7	*MEM_AXI_WLAST*	*MEM_AXI_RLAST*
6	*NA*	*MEM_AXI_ARID[1]*
5	*MEM_AXI_AWID[0]*	*MEM_AXI_ARID[0]*
4	*MEM_AXI_AWVALID*	*MEM_AXI_ARVALID*
3	*MEM_AXI_AWREADY*	*MEM_AXI_ARREADY*
2	*MEM_AXI_AWLEN[2]*	*MEM_AXI_ARLEN[2]*
1	*MEM_AXI_AWLEN[1]*	*MEM_AXI_ARLEN[1]*
0	*MEM_AXI_AWLEN[0]*	*MEM_AXI_ARLEN[0]*

1.4.5.25.3 TX-Scan

In order to keep track of the TX-Scan process, some registers provide the relevant information to observe the selection of the next TX message, meaning, which TX FIFO Queue number and which message within this FIFO Queue is selected, or which TX Priority Queue slot number. The **TX_SCAN_FC** and **TX_SCAN_BC** registers are monitoring the TX-Scan activity, see Software Interface chapter for more details. The duration of a CAN frame is large enough to make it possible, to read those registers in time and get some valuable information.

The **TX_SCAN_FC** register provides the information of the source of the four first candidates selected by the TX-Scan., meaning the TX FIFO Queue number or the TX Priority Queue slot number.
The source of the two first candidates are defined by:

- The **TX_SCAN_FC.FQ_PQ{n}** (n € {0, 1}) bit register: when set to 0, the first or second candidate is a TX FIFO Queue. In fact, **TX_SCAN_FC.FQ_PQ0 = TX_SCAN_BC.FH_PQ** and **TX_SCAN_FC.FQ_PQ1 = TX_SCAN_BC.SH_PQ**, see TX_SCAN_BC register description below
- The **TX_SCAN_FC.FQN_PQSN{n}** (n € {0, 1}) bit register: define either a TX FIFO Queue number or a TX Priority Queue slot number according to the **TX_SCAN_FC.FQ_PQ{n}** (n € {0, 1}) bit register. In fact, **TX_SCAN_FC.FQN_PQSN0 = TX_SCAN_BC.FH_FQN_PQSN** and **TX_SCAN_FC.FQN_PQSN1 = TX_SCAN_BC.SH_FQN_PQSN**, see TX_SCAN_BC register description below
- The two sources of the last two candidates are monitoring the selection of a TX-Scan. It is essential to note that the value in those registers is not stable, compare to the source of the two first candidate, and will change during a TX-Scan run. When the **TX_SCAN_FC.FQ_PQ{n}** (n € {0, 1, ..., 3}) is set to 0, the candidate n is a TX FIFO Queue, looking at the **TX_SCAN_FC.FQN_PQSN{n}** (n € {0, 1, ..., 3} bit field, provides the number. If the **TX_SCAN_FC.FQ_PQ{n}** (n € {0, 1, ..., 3}) is set to 1, the candidate n is a TX Priority Queue and the slot number is defined by the **TX_SCAN_FC.FQN_PQSN{n}** (n € {0, 1, ..., 3} bit field.

The value of the **TX_SCAN_FC** register is updated when a new TX Scan result is available, see TX-SCAN chapter.

The **TX_SCAN_BC** register gives the full reference of the first and second highest priority messages, defined and uploaded at the end of a TX-Scan run (see Buffer A and B in TX Message Handler chapter). The first highest candidate is the one selected and sent to the CAN bus. The second highest priority candidate is the TX message to be sent, once the transmission of the first highest candidate is completed. The register values provide full visibility of the two message candidates stored locally in Buffer A and B, see TX Message Handler chapter for more details. As such, those registers are stable over time and do change only after at the end of a TX-Scan.
The first best candidate is defined by:

- The **TX_SCAN_BC.FH_PQ** bit register: when set to 0, the candidate is a TX FIFO Queue
- The **TX_SCAN_BC.FH_FQN_PQSN** bit register: define either a TX FIFO Queue number or a TX Priority Queue slot number according to the **TX_SCAN_BC.FH_PQ** bit register
- The **TX_SCAN_BC.FH_OFFSET** bit register: define the offset (in 32byte) of the TX descriptor selected, starting from the initial start address of the TX FIFO Queue (defined in the **TX_SCAN_BC.FH_FQN_PQS** bit register) with TX descriptor address = TX FIFO Queue start address + offset * 32byte. When the candidate is a TX Priority Queue slot, the **TX_SCAN_BC.FH_OFFSET** register has no meaning and is set to 0.

The second best candidate is defined by:

- The **TX_SCAN_BC.SH_PQ** bit register: when set to 0, the candidate is a TX FIFO Queue
- The **TX_SCAN_BC.SH_FQN_PQSN** bit register: define either the TX FIFO Queue number or the TX Priority Queue slot number according to the **TX_SCAN_BC.FH_PQ** bit register
- The **TX_SCAN_BC.SH_OFFSET** bit register: define the offset (in 32byte) of the TX descriptor selected, starting from the initial start address of the TX FIFO Queue (defined in the

TX_SCAN_BC.SH_FQN_PQSN bit register) with TX descriptor address = TX FIFO Queue start address + offset * 32byte. It is important to note that, when the TX FIFO queue selected for the first best candidate is identical to the one for the second, the offset would be also identical. In such scenario, the second best candidate is always the next TX descriptor of that TX FIFO Queue. When the candidate is a TX Priority Queue slot, the **TX_SCAN_BC.SH_OFFSET** register has no meaning and is set to 0

1.4.5.25.4 TX Descriptor Tracking in a TX FIFO Queue

The current and next TX descriptors for a given TX FIFO Queue n are stored in the L_MEM and can be identified in the **TX_FQ_DESC_VALID.DESC_NC_VALID[n]** and **TX_FQ_DESC_VALID.DESC_CN_VALID[n]** bit registers.
Here is the status for those bit registers when progressing with the TX FIFO Queue n:
Initial start:
1) The current TX descriptor (first one in case this is an initial start) fetched from S_MEM and written to L_MEM is leading to **TX_FQ_DESC_VALID.DESC_CN_VALID[n]** = 1 and **TX_FQ_DESC_VALID.DESC_NC_VALID[n]** = 0.
2) If the current TX descriptor is about to be sent go to 3), otherwise stay in 2) and no updates are done on bit registers.
3) The next descriptor is fetched from S_MEM and written in L_MEM.

- If the next TX descriptor is not valid then the TX FIFO Queue n is put on hold. The **TX_FQ_DESC_VALID.DESC_CN_VALID[n]** set to 1, goes to 0 once the TX message is sent (**TX_FQ_DESC_VALID.DESC_NC_VALID[n]** =0).
- If the descriptor is valid, **TX_FQ_DESC_VALID.DESC_CN_VALID[n]** = 1 and **TX_FQ_DESC_VALID.DESC_NC_VALID[n]** =1, go to 4)

4) When the current TX message is fully sent, **TX_FQ_DESC_VALID.DESC_CN_VALID[n]** = 0 and **TX_FQ_DESC_VALID.DESC_NC_VALID[n]** =1.
5) If the current TX descriptor is about to be sent go to 6), otherwise stay in 5) and no updates are done on bit registers.
6) The next descriptor is fetched from S_MEM and written in L_MEM.

- If the next TX descriptor is not valid then the TX FIFO Queue n is put on hold. The **TX_FQ_DESC_VALID.DESC_NC_VALID[n]** set to 1, goes to 0 once the TX message is sent (**TX_FQ_DESC_VALID.DESC_CN_VALID[n]** =0).
- If the descriptor is valid, **TX_FQ_DESC_VALID.DESC_NC_VALID[n]** = 1 and **TX_FQ_DESC_VALID.DESC_CN_VALID[n]** =1, go to 7)

7) When the current TX message is fully sent, **TX_FQ_DESC_VALID.DESC_NC_VALID[n]** = 0 and **TX_FQ_DESC_VALID.DESC_CN_VALID[n]** =1, go to 2)

1.4.5.25.5 TX Descriptor Tracking in TX Priority Queue

As soon as a TX Priority Queue slot n is started, the corresponding TX descriptor is fetched from the S_MEM and written to the L_MEM. When the TX descriptor assigned to the slot n is fully written in the L_MEM, the **TX_PQ_DESC_VALID.DESC_VALID[n]** is set to 1.
If the TX descriptor fetched is not valid or has any safety issue, the **TX_PQ_DESC_VALID.DESC_VALID[n]** is not set. In case the TX message of the slot n is discarded, the **TX_PQ_DESC_VALID.DESC_VALID[n]** is set back to 0.

1.4.5.25.6 Safety Measures

If a safety measure is active and an issue is detected, the MH may stop. Therefore, it would be difficult for the SW to analyze and identify the possible root causes. To allow such debugging, every embedded safety measure can be disabled individually, see **MH_SFTY_CTRL** register.

1.4.5.26 RX and TX Statistics

1.4.5.26.1 RX Statistic Counters

Two 12bit counters are provided to keep track of how many messages have been received successfully/unsuccessfully, see **RX_STATISTICS** register. When a counter has reached the maximum value, it will wrap to zero with the next increment. The counters can be cleared (set to 0) by writing 0 to the dedicated register bit field. To identify when counters are wrapping, the *STATS_IRQ* interrupt line is triggered to the system. To identify the counter which has wrapped, a read to the **STATS_INT_STS** register is required. Writing a 1 to the corresponding bit will clear the bit.

Here is the list of root cause to increment the **RX_STATISTICS.SUCC[11:0]** counter:

- When a RX message is stored in S_MEM and its RX Header descriptor is acknowledged

Here is the list of root cause to increment the **RX_STATISTICS.UNSUCC[11:0]** counter.
Safety or Errors:

- When a RX data parity error is detected
- When an RX address pointer parity error is detected
- When an RX descriptor error (request, CRC or invalid) is detected and used for the current RX message

Functional:

- When an ABORT code word is received from the PRT
- When a DO code word is received from the PRT
- When the RX message cannot be written to the RX FIFO Queue (queue not enabled and/or started)
- When a data overflow occurs on the RX DMA FIFO
- When a new RX message is received while one is already in progress
- When the PRT ENABLE signal is going from High to Low when receiving an RX frame

NOTE: an ABORT codeword from PRT, for an arbitration lost, does not increment the counter of unsuccessful transmissions.

1.4.5.26.2 TX Statistic Counters

Two 12bit counters are provided to keep track of how many messages have been transmitted successfully/unsuccessfully, see **TX_STATISTICS** register. When a counter has reached the maximum value, it will wrap to zero with the next increment. The counters can be cleared (set to 0) by writing 0 to the dedicated register bit field. To identify when counters are wrapping, the *STATS_IRQ* interrupt line is triggered to the system. To identify the counter which has wrapped, a read to the **STATS_INT_STS** register is required. Writing a 1 to the corresponding bit will clear the bit.
NOTE: an ABORT codeword from PRT, for an arbitration lost, does not increment the counter.

Here is the list of root cause to increment the **TX_STATISTICS.SUCC[11:0]** counter:

- When a TX message is fully sent to the PRT, and its TX Header descriptor is acknowledged

Here is the list of root cause to increment the **TX_STATISTICS.UNSUCC[11:0]** counter.
Safety or Errors:

- Not applicable

Functional:

- When a HFI code word is received from the PRT
- When the maximum number of transmissions allowed for a given TX message is reached
- When the TX filtering has rejected a TX message

1.4.5.27 Register Access

The MH registers are accessible in read/write mode through its AXI4-Lite slave interface *HOST_AXI* (compliant to AMBA 4 ARM Ltd protocol, see [5]).

Any access to registers, either read or write, must use a 32bit aligned address, otherwise a SLVERR is provided as a response.

When an access is performed to a non-mapped register in the address range, a SLVERR is provided as a response.

When a read access to write-only registers or a write access to read-only registers is performed, a SLVERR is provided as a response.

When an access is performed to a write-only Privileged register in the address range, a SLVERR is provided as a response.

The phrase ‘SLVERR is provided as a response’ means that the HOST_AXI responds with RRESP = ‘SLVERR’ respective BRESP = ‘SLVERR’.
The error is only reported on the AXI4-Lite protocol, no interrupt is triggered for such issue.

The register interface does not provide any access to the L_MEM. It would in charge of the integrator to provide a direct access to the L_MEM to write the relevant data for the MH.

1.4.5.28 Register Protection

1.4.5.28.1 Lock Mechanism Protection

To secure the access to some of the critical registers, an unlock key sequence is required prior to any write-modified access. This procedure must be done before every write to a locked register. As soon as the write is completed, the register is automatically set back to lock mode.

When an access is performed to a locked register, a SLVERR is provided as a response. The error is only reported on the AXI4-Lite protocol, no interrupt is triggered for such issue.

Two locks are provided for two different purposes:

- A lock that protects the register in charge of stopping RX and TX FIFO Queues as well as TX Priority Queue slots

- A lock that protects the MH to be set in debug mode

Functional Lock

This sequence is based on three steps as defined below:

- Write 0x1234 to the **LOCK.ULK[15:0]** bit field register
- Write 0x4321 to the **LOCK.ULK[15:0]** bit field register
- Write to the unlocked register the expected value

Once the write is performed to the register, it will automatically be locked again.
The following list of registers are using this protection:

- **TX_FQ_CTRL1**
- **TX_PQ_CTRL1**
- **RX_FQ_CTRL1**

Test Mode Lock

An unlock key sequence is required to access the registers assigned to debug and test purpose in write mode. This procedure must be done before any write action is executed to the locked registers.

This sequence is based on three steps as defined below:

- Write 0x6789 to the **LOCK.TMK[15:0]** bit field register
- Write 0x9876 to the **LOCK.TMK[15:0]** bit field register
- Write to the unlocked register the expected value

Once the write is performed to the register, it will automatically be locked again.
The only register using this specific key sequence is the **DEBUG_TEST_CTRL** register as it does control the debug mode.

1.4.5.28.2 Conditional Access Protection

Some registers can be written if a bit, that is defined in another register, allows the access. As long as this conditional bit has the right value, any single or consecutive writes can be performed. Configuration registers are protected by this mechanism to avoid any changes while the logic is running.

When an access is performed to a write protected register, a SLVERR is provided as a response. The error is only reported on the AXI4-Lite protocol, no interrupt is triggered for such issue.

The registers with conditional accesses are defined in table below.
Table: Conditional Access Register List

Register Name	Condition to write access	Description/Constraints
MH_CFG	MH_STS.BUSY=0	The register can be written if the MH is not running
MH_SFTY_CFG		
MH_SFTY_CTRL		
RX_FILTER_MEM_ADD		
TX_DESC_MEM_ADD		
AXI_ADD_EXT		
AXI_PARAMS		
TX_FILTER_CTRL0		
TX_FILTER_CTRL1		
TX_FILTER_REFVAL0		
TX_FILTER_REFVAL1		
TX_FILTER_REFVAL2		
TX_FILTER_REFVAL3		
RX_FILTER_CTRL		
TX_FQ_START_ADD{n}	TX_FQ_STS0.BUSY[n] = 0	The register can be written if the TX FIFO Queue n is not running (n € {0, 1, 2, ..., 7})
TX_FQ_SIZE0{n}		
TX_PQ_START_ADD	TX_PQ_STS0 = 0x00000000	The register can be written if no TX Priority Queue slots are running
RX_FQ_START_ADD{n}	RX_FQ_STS0.BUSY[n] = 0x00	The register can be written if the RX FIFO Queue n is not running (n € {0, 1, 2, ..., 7})
RX_FQ_SIZE{n}		
RX_FQ_DC_START_ADD{n}		
INT_TEST0	DEBUG_TEST_CTRL.TEST_IRQ_EN = 1	The interrupt lines can be trigger by SW if the interrupt test mode is enabled
INT_TEST1		

1.4.5.28.3 Register Access Mode

To protect the Debug/Integration Test functions and RX/TX filtering setting, the following registers can only be written using Privileged/Non-secure/Data access (HOST_AXI_AWPROT[0] = 1):

- TX_FILTER_CTRL0
- TX_FILTER_CTRL1
- TX_FILTER_REFVAL0
- TX_FILTER_REFVAL1
- TX_FILTER_REFVAL2
- TX_FILTER_REFVAL3
- RX_FILTER_CTRL
- DEBUG_TEST_CTRL

Other registers than the ones listed above can use Normal/Non-secure/Data access.

1.4.5.28.4 Register CRC Computation

To protect the MH configuration, control and configuration registers are protected using CRC. A reference CRC, computed by the SW, is set to a register, and compare with an internal CRC value computed by the MH. It is important to note that some of the registers won't be accessible once the MH is started, **MH_STS.BUSY** set to 1, refer to Conditional Access Protection and Lock Mechanism Protection sections for more details.
Once the overall MH setting is done and only when there are no more changes on the control and configuration register, do the following:

- The SW must provide the expected CRC for list of protected registers. The CRC reference value must be computed, only for the registers defined as CRC protected, starting from the lowest address offset. The order of the register to be considered for the CRC computation is defined below (all register values will be checked). The CRC is computed using the 32bit value of the register defined in the list.
- Once the 32bit CRC value is computed by SW, it must be written to the **CRC_REG** register. The write access to this register does not trig a CRC check
- To check the CRC for the registers, set the **CRC_CTRL.START** bit to 1. The MH goes through the list of CRC protected registers and compute the global CRC. After a few cycles, the CRC reference value in the **CRC_REG** register is compared with the one already computed. If a CRC error is detected, the *REG_CRC_ERR* interrupt signal is triggered. As the check is only done and control by SW, there is no enable defined

Nothing is preventing the SW to launch at a regular time interval a CRC check by setting the **CRC_CTRL.START** bit to 1.
It is recommended to perform a register CRC check, for any new configuration, to ensure a proper setting before starting the MH.

Here below is the list of registers protected by CRC, in the order they need to be considered (refer to sections in Register Protection chapter for register accessibility):

- VERSION
- MH_CFG
- MH_SFTY_CFG
- MH_SFTY_CTRL
- RX_FILTER_MEM_ADD
- TX_DESC_MEM_ADD
- AXI_ADD_EXT
- AXI_PARAMS
- LOOP n from 0 to 7
- TX_FQ_START_ADD{n}
- TX_FQ_SIZE{n}
- END LOOP
- TX_PQ_START_ADD
- LOOP n from 0 to 7
- RX_FQ_START_ADD{n}
- RX_FQ_SIZE{n}
- RX_FQ_DC_START_ADD{n}
- END LOOP
- TX_FILTER_CTRL0 (Privileged)
- TX_FILTER_CTRL1 (Privileged)
- LOOP n from 0 to 3
- TX_FILTER_REFVAL{n} (Privileged)
- END LOOP
- RX_FILTER_CTRL (Privileged)
- DEBUG_TEST_CTRL (Privileged)
- INT_TEST0
- INT_TEST1

Here below is the normal polynomial representation and implementation of the CRC-32 used to protect the registers:
CRC-32 = $(x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1)$ ⁺

Here below is the pseudo code to compute the CRC for the MH register bank:
The reg_table[] is the array of 32bit registers defined previously (in the order they are listed):

```
static logic[31:0] rem32      = 32'hffffffff;
static logic[31:0] rem32_old  = 32'hffffffff;
static logic[31:0] poly       = 32'h4c11db7;
static logic[31:0] crc32;

// initialize CRC shift register
// This algorithm is indirect
```

```
rem32 = 32'hffffffff;
foreach (reg_table[i]) begin
   for (int j = 31; j >= 0; j--) begin

     // to decide whether reduction with polynomial will be required based on MSB before shift
     rem32_old = rem32;

     // shift out MSB of CRC
     rem32 = rem32 << 1;
     rem32[0] = reg_table[i].get()[j];

     // perform reduction if required
     if (rem32_old[31]) rem32 = rem32 ^ poly;
   end
end

// processing 32 0s more
repeat(32) begin
   // to decide whether reduction with polynomial will be required based on MSB before shift
   rem32_old = rem32;

   // shift out MSB of CRC
   rem32 = rem32 << 1;
   rem32[0] = 0;

   // perform reduction if required
   if (rem32_old[31]) rem32 = rem32 ^ poly;
end
crc32 = rem32;
```

1.4.5.29 Error and Exception Handling

Here is the list of potential issues the MH may have to handle and how it will react:

Error	source	Interrupt	MH behavior
		MH	
RX acknowledge path overflow	Acknowledge data not sent in time before new one needs to be stored	*DP_DO_ERR*	The current RX message is discarded and an RX_ABORT_IRQ is triggered to the system. The interrupt *DP_DO_ERR* is triggered to the system and the **ERR_INT_STS.DP_RX_ACK_DO_ERR** bit status register is set to 1. The MH finishes its

Error	source	Interrupt	MH behavior
			current transactions and stops with **MH_CTRL.BUSY** = 0. The SW needs to restart it and the MH will keep going with its current task.
TX acknowledge path overflow	Acknowledge data not sent in time before the new one being stored	*DP_DO_ERR*	As soon as an acknowledge data locally stored and ready to be send cannot be done (due to some overflow) no new messages will be sent to the PRT. The *DP_DO_ERR* interrupt is triggered to the system and the **ERR_INT_STS.DP_TX_ACK_DO_ERR** bit status register is set to 1. The MH finishes its current transactions and stops with **MH_CTRL.BUSY** = 0. The SW needs to restart it and the MH will keep going with its current task.
RX DMA FIFO overflow	The FIFO overflow on the RX path	*DP_DO_ERR*	The current RX message is discarded and an *RX_ABORT_IRQ* is sent to the system. The already RX descriptors used are allocated for the next message. No status is sent back to the Header Descriptor. The MH keeps receiving RX message despite this temporary issue. The **ERR_INT_STS.DP_RX_FIFO_DO_ERR** bit status register is set to 1.
RX DMA FIFO above threshold while RX filtering in progress	The RX filter has not completed in time to avoid a potential overflow	*NONE*	The current RX message is sent to the default RX FIFO Queue as backup solution if enable, see the **RX_FILTER_CTRL** register. The threshold is defined to provide enough time for the MH to write the message in S_MEM. The RX Header descriptor of that RX message will have its status report bit field set to “message received but not filtered”. The MH keeps receiving RX message.
RX descriptor CRC error when fetched from S_MEM	A CRC error is detected on RX descriptor	*DESC_ERR*	As the RX descriptor has a CRC error, the related RX FIFO Queue is stopped and the interrupt *DESC_ERR* is triggered to the system, see **RX_FQ_STS1, RX_FQ_STS0** registers. The **SFTY_INT_STS.RX_DESC_CRC_ERR** bit status register is set to 1. Other RX FIFO Queues would still be running.
Wrong RX	The expected	*DESC_ERR*	As the RX descriptor is not the one

Error	source	Interrupt	MH behavior
descriptor fetched from S_MEM	descriptor is not the one coming back from S_MEM. Several issue on the address could lead to such result		expected, the related RX FIFO Queue is stopped and the interrupt *DESC_ERR* is triggered to the system, see **RX_FQ_STS1, RX_FQ_STS0** registers. The **SFTY_INT_STS.RX_DESC_REQ_ERR** bit status register is set to 1. Other RX FIFO Queues would still be running.
TX descriptor CRC error when fetched from S_MEM	A CRC error is detected on TX descriptor when fetched from S_MEM	*DESC_ERR*	As the TX descriptor has a CRC error, the related TX FIFO Queue is stopped and the interrupt *DESC_ERR* is triggered to the system. The **SFTY_INT_STS.TX_DESC_CRC_ERR** bit status register is set to 1. Other TX FIFO Queues would still be running. TX Priority Queue slots are managed differently. If an issue occurs on the TX descriptor the slot will have its busy and sent flags set to 0, see **TX_PQ_STS0** and **TX_PQ_STS1** registers.
Wrong TX descriptor fetched from S_MEM	The expected descriptor is not the one coming back from S_MEM. Several issue on the address could lead to such result	*DESC_ERR*	As the TX descriptor is not the one expected, the related TX FIFO Queue is stopped and the interrupt *DESC_ERR* is triggered to the system. The **SFTY_INT_STS.TX_DESC_REQ_ERR** bit status register is set to 1. Other TX FIFO Queues would still be running. TX Priority Queue slots are managed differently. If an issue occurs on the TX descriptor the slot will have its busy and sent flags set to 0, see **TX_PQ_STS0** and **TX_PQ_STS1** registers.
Wrong TX descriptor fetched from L_MEM	The expected descriptor is not the one coming back from L_MEM. Several issue on the address could lead to such result	*DESC_ERR*	The TX descriptor selected to be the next message candidate is corrupted. Either the related TX FIFO Queue is stopped (see **TX_FQ_STS0** and **SFTY_INT_STS** registers) or the TX Priority Queue slot is set disable (busy flag set to 0), see **TX_PQ_STS0** and **SFTY_INT_STS** registers). The interrupt *DESC_ERR* is triggered to the system. The **SFTY_INT_STS.TX_DESC_REQ_ERR** bit status register is set to 1. Other TX FIFO Queues would still be running as well as TX Priority Queue slots.
Parity error	One of the address	*AP_PARITY_ERR*	As the source of parity issue, could lead to

Error	source	Interrupt	MH behavior
detected on TX address pointers	pointers managing the TX FIFO Queues or the TX Priority Queue is corrupted		wrong memory accesses, the MH stops. The MH finishes all pending data transfers and then stops with **MH_STS.BUSY** = 0. The SW is notified through the *AP_PARITY_ERR* interrupt and the **SFTY_INT_STS.AP_TX_PARITY_ERR** status bit register is set to 1.
Parity error detected on RX address pointers	One of the address pointers managing the RX FIFO Queues is corrupted	*AP_PARITY_ERR*	As the source of parity issue, could lead to wrong memory accesses, the MH stops. The MH finishes all pending data transfers and then stops with **MH_STS.BUSY** = 0. The SW is notified through the *AP_PARITY_ERR* interrupt and the **SFTY_INT_STS.AP_RX_PARITY_ERR** status bit register is set to 1.
Register CRC error	One of the configuration registers protected by CRC is corrupted	*REG_CRC_ERR*	There is no way to define which register is corrupted and to evaluate which part of the logic would be impacted. The MH is stopped. When receiving an RX message, the current message is discarded, and all RX FIFO Queues are stopped. When transmitting a message, it is aborted. All TX FIFO Queues are stopped, and all TX Priority Queue slot are disabled. The interrupt *REG_CRC_ERR* is sent to system
TX data path sequence error	Any error sequence detected on the TX_MSG interface	*DP_SEQ_ERR*	If any code word reported by the PRT does not match the expected sequence the PRT and MH are no more synchronized. The MH finishes all pending data transfers and then stops with **MH_STS.BUSY** = 0. the *DP_SEQ_ERR* interrupt is triggered with the **ERR_INT_STS.DP_TX_SEQ_ERR** bit status register set to 1.
RX data path sequence error	Any error sequence detected on the RX_MSG interface	*DP_SEQ_ERR*	If any code word reported by the PRT does not match the expected sequence the PRT and MH are no more synchronized. The MH finishes all pending data transfers and then stops with **MH_STS.BUSY** = 0. the *DP_SEQ_ERR* interrupt is triggered with the **ERR_INT_STS.DP_RX_SEQ_ERR** bit status register set to 1.
RX frame	Due to a high	*RX_ABORT_IRQ*	As the current RX message has not complete

Error	source	Interrupt	MH behavior
reception in progress when receiving a new RX message	number of filter elements and a high latency on the local memory, the RX filtering process may cover almost the shortest CAN frame, leading to an overlap on the current and new RX messages		prior receiving the next frame, the new frame is discarded to provide the remaining time to complete the process on the current one. Therefore, any new RX message is aborted with an RX_ABORT_IRQ interrupt.
RX filter not done in time before a new RX frame	The RX filter does not complete in time its process to identify the RX FIFO Queue	*RX_FILTER_ERR*	When the RX filter is taking too much time and a new RX message is coming, the *RX_FILTER_ERR* interrupt is triggered. The new RX message is discarded, see **RX_FILTER_CTRL** register. The MH keeps running on its current frame. Such interrupt is a good indicator for SW to identify large RX filtering time on some frames.
RX FIFO Queue not enabled for reception	The RX FIFO Queue selected to receive the RX message is not running, either not set or wrongly set	*RX_ABORT_IRQ*	The selected RX FIFO Queue defined after the RX filtering process is disable. The MH discards the RX message with the *RX_ABORT_IRQ* interrupt. Every RX message going to this disabled RX FIFO Queue will trigger this interrupt. The SW must ensure RX FIFO Queues are enable at first time.
	The RX Filter cannot send message to the RX FIFO Queue as it is disable	*RX_FILTER_ERR*	In case the RX Filter identifies an RX FIFO Queue to receive an RX message, but this queue is disable, the RX_FILTER_ERR interrupt is triggered, and the current message is discarded. The SW must ensure RX FIFO Queues are enable at first time, otherwise several interrupts will occur in a row.
TX message rejected by TX filter	The Header Descriptor is filtered to ensure only well-defined TX message can go through	*TX_FILTER_IRQ*	When a TX message is rejected, it will be skipped by the MH. When the Header descriptor is in a TX FIFO Queue, the next message in the FIFO is used instead. An acknowledge is sent to the TX descriptor with the status rejected. Regarding TX Priority Queue, the corresponding slot is

Error	source	Interrupt	MH behavior
			disabled. The MH keeps running all other TX FIFO Queues or slots defined as valid.
DMA channel interface mixed up	Wrong data sent or received to DMA channel interface detected by the DMA	*DMA_CH_ERR*	As such issue would mean data are mixed-up between channels, there is no way to recover. The MH finishes all pending data transfers and then stops with **MH_STS.BUSY** = 0. The system is notified through the DMA_CH_ERR interrupt line. There is no status flag assigned to such interrupt as the DMA channel being faulty cannot be identified.
Parity error on RX message data	A bit flip is detected on data from the RX_MSG to the AXI system bus interface	*DP_PARITY_ERR*	If such issue occurs while receiving data, the RX message would be discarded. No acknowledge data is sent. An interrupt *DP_PARITY_ERR* is triggered. The **SFTY_INT_STS.DP_RX_PARITY_ERR** bit status register is set to 1. As the RX message would be aborted, the *RX_ABORT_IRQ* interrupt would also be set. The MH keeps going with new messages.
Parity error on TX message data	A bit flip is detected on payload data from the AXI system bus interface to the TX_MSG	*DP_PARITY_ERR*	If such issue occurs while transmitting data, the TX message would be aborted. An interrupt *DP_PARITY_ERR* would be triggered. The **SFTY_INT_STS.DP_TX_PARITY_ERR** bit status register is set to 1. The *TX_ABORT_IRQ* interrupt is set.
Parity error on RX message acknowledge data	A bit flip is detected on data from the RX_MSG to the AXI system bus interface	*DP_PARITY_ERR*	If such issue occurs on the acknowledge data, the RX message would be discarded. No acknowledge data is sent. An interrupt *DP_PARITY_ERR* is triggered. The **SFTY_INT_STS.ACK_RX_PARITY_ERR** bit status register is set to 1. The MH keeps going with new messages.
Parity error on TX message acknowledge data	A bit flip is detected on payload data from the AXI system bus interface to the TX_MSG	*DP_PARITY_ERR*	If such issue occurs while acknowledging the TX message. An interrupt *DP_PARITY_ERR* is triggered. The **SFTY_INT_STS.ACK_TX_PARITY_ERR** bit status register is set to 1. The SW can identify such issue reading the report status of that TX descriptor.
RX message	MH_CTRL.START	*NONE*	The MH does not accept RX message data

Error	source	Interrupt	MH behavior
received while MH not started	bit wrongly set to 1		from the PRT. As the PRT cannot sent data to the MH, a data overflow on the PRT will occur leading to an interrupt.
		PRT	
RX data path overflow	DO code word received from PRT. . Several issues could lead to such issue, peak latency preventing write accesses in time or RX path being stopped or ...	*NA*	The RX message is discarded. The already used RX descriptor are reused for the next message. Then, no status is sent back to the Header Descriptor in S_MEM.
RX message on CAN bus not successful	ABORT code word received from PRT. Invalid CAN message detected on CAN bus	*NA*	This is normal behavior. The RX message is discarded. The already used RX descriptors are allocated for the next message. No acknowledge data is sent back to the S_MEM
TX data path underrun	DU code word received from PRT. TX message data not provided in time	*NA*	The current TX message, selected and started on the MH side, is aborted but the PRT keeps going with its current frame and will generated a wrong CRC to invalidate the frame at the receiver side. All data transfers from S_MEM is aborted. The issue may be the result of a peak latency. The TX message is still valid and will be part of the next TX-Scan. The MH can restart to transmit the same message according to the restart counter setting. The PRT is triggered an interrupt to the system when such code word is transmitted to the MH. It is essential to understand that the MH will still be active and fully functional. There is no message loss when such issue occurs
TX message on CAN bus not successful	RESTART code word received from PRT	*NA*	The current TX message, selected and started on the MH side, is aborted. All data transfers from S_MEM is aborted. The current TX message is still valid and will be part of the next TX-Scan. The MH can restart to transmit the same message according to the restart counter setting or use another one with highest priority .

Error	source	Interrupt	MH behavior
TX message header invalid	HFI code word received from PRT	*NA*	The current TX message is discarded, and a data acknowledge is sent back to the Header Descriptor in S_MEM. The report status of the TX descriptor is updated with the issue. If the TX message was in a TX FIFO Queue, the MH keeps running and skips this TX message to fetch the next one. In case of a TX Priority Queue slot, the slot is set as done but not sent (see report status in TX descriptor)
Unexpected Start Of Sequence (USOS) at the TX_MSG interface	When PRT detects USOS, it stops CAN protocol operation and sets *ENABLE*=0	*STOP_IRQ*	In case such code word is received, the MH and the PRT are no more synchronized. The MH finishes its current transfers and stops. The *STOP_IRQ* interrupt is set to notify MH is no more active. In such scenario the only action would be to reset the MH and PRT to recover.
PRT entered CAN protocol's Bus-Off state	PRT stops CAN protocol operation and sets *ENABLE*=0	*STOP_IRQ*	MH finishes all pending data transfers and then stops (put on hold). All FSM in the MH go to idle. The *STOP_IRQ* interrupt is set to notify MH is no more active. A write to the **MH_CTRL.START** bit register allows the SW to restart everything at the point it was stopped, if required.
PRT stopped by SW	PRT stops CAN protocol operation and sets *ENABLE*=0	*STOP_IRQ*	
PRT TX_MSG interface not responding	PRT is having a deadlock and cannot answer to MH request or receive data	*DP_TO_ERR*	When the timeout assigned to the TX_MSG interface fires, the MH finishes all pending data transfers and then stops with **MH_STS.BUSY** = 0. The DP_TO_ERR interrupt is triggered to the system, with the **ERR_INT_STS.DP_TX_TO_ERR** bit status set to 1
PRT RX_MSG interface not responding	PRT is having a deadlock and cannot send data to MH	*DP_TO_ERR*	When the timeout assigned to the RX_MSG interface fires, the MH finishes all pending data transfers and then stops with **MH_STS.BUSY** = 0. The DP_TO_ERR interrupt is triggered to the system with the **ERR_INT_STS.DP_RX_TO_ERR** bit status set to 1
LOCAL MEMORY (L_MEM)			
Local memory safety error while	The L_MEM is providing a safety	*MEM_SFTY_ERR*	As the corrupted data word is corrected, the RX filtering can be done on the current RX

Error	source	Interrupt	MH behavior
reading RX filter element. Corrupted data has been corrected	error on *MEM_SFTY_CE* input signal while reading a data		message. The MH keeps running and will be able to receive new messages. The interrupt *MEM_SFTY_ERR* is triggered to the system with the **SFTY_INT_STS.MEM_SFTY_CE** bit status register set to 1. It is essential for such issue that there is no error response on the memory interface while reading the corrected data.
Local memory safety error while reading TX descriptor. Corrupted data has been corrected	The L_MEM is providing a safety error on *MEM_SFTY_CE* input signal while reading a data	*MEM_SFTY_ERR*	As the TX descriptor selected to be the next message candidate is corrupted but corrected, the related TX FIFO Queue or the TX Priority Queue slot will run as normal. The TX-Scan is reading a corrected TX descriptor and will complete. The interrupt *MEM_SFTY_ERR* is triggered to the system, with the **SFTY_INT_STS.MEM_SFTY_CE** bit status register set to 1. It is essential for such issue that there is no error response on the memory interface while reading the corrected data.
Local memory safety error while reading RX filter element. Corrupted data is not corrected	The L_MEM is providing a safety error on *MEM_SFTY_UE* input signal while reading a data with SLVERR response	*MEM_SFTY_ERR*	As no more filtering can be done on the current RX message, it is discarded. As it is not possible to keep going with a corrupted RX filter element, the MH stops. The interrupt *MEM_SFTY_ERR* is triggered to the system with the **SFTY_INT_STS.MEM_SFTY_UE** bit status register set to 1. The MH finishes all pending data transfers and then stops with **MH_STS.BUSY** = 0. It is essential for such issue, to have the memory interface reporting a SLVERR when reading the corrupted data.
Local memory safety error while reading TX descriptor. Corrupted data is not corrected	The L_MEM is providing a safety error on *MEM_SFTY_UE* input signal while reading a data with a SLVERR response	*MEM_SFTY_ERR*	As it is not possible to keep going with a corrupted TX descriptor, the MH stops. The interrupt *MEM_SFTY_ERR* is triggered to the system if such issue occurs with the **SFTY_INT_STS.MEM_SFTY_UE** bit status register set to 1. The MH finishes all pending data transfers and then stops with **MH_STS.BUSY** = 0. It is essential for such issue, to have the memory interface

Error	source	Interrupt	MH behavior
			reporting a SLVERR when reading the corrupted data.
Error response received on local memory write access. A safety issue is not considered here	The L_MEM is providing a DECERR/SLVERR error response on BRESP[1:0] for a write access	*RESP_ERR[0]*	The TX descriptor written to the L_MEM is not valid. In such cases as the L_MEM cannot be trusted anymore, the MH stops. The MH finishes all pending data transfers and then stops with the **MH_STS.BUSY** = 0. The *RESP_ERR[0]* interrupt is triggered to the system. To identify the issue the BRESP[1:0] and the ID of the transaction are logged in the **AXI_ERR_INFO.MEM_ID[1:0]** and **AXI_ERR_INFO.MEM_RESP[1:0]** bit status register.
Error response received on local memory read access. A safety issue is not considered here	The L_MEM is providing a DECERR/SLVERR error response on RRESP[1:0] for a read access	*RESP_ERR[1]*	The TX descriptor or RX Filter element read from the L_MEM is not valid. In such cases as the L_MEM cannot be trusted anymore, the MH stops. The MH finishes all pending data transfers and stops with the **MH_STS.BUSY** = 0. The *RESP_ERR[0]* interrupt is triggered to the system. To identify the issue the RRESP[1:0] and the ID of the transaction are logged in the **AXI_ERR_INFO.MEM_ID[1:0]** and **AXI_ERR_INFO.MEM_RESP[1:0]** bit status register.
A read from the L_MEM cannot complete	The MH does not complete a read within a defined time frame	*MEM_TO_ERR*	When the timeout assigned to the L_MEM AXI read channel fires, the MH finishes all pending data transfers and then stops with **MH_STS.BUSY** = 0. The MEM_TO_ERR interrupt is triggered to the system and the **SFTY_INT_STS.MEM_AXI_RD_TO_ERR** bit status is set to 1
A write to the L_MEM cannot complete	The MH does not complete a write within a defined time frame	*MEM_TO_ERR*	When the timeout assigned to the L_MEM AXI write channel fires, the MH finishes all pending data transfers and then stops with **MH_STS.BUSY** = 0. The MEM_TO_ERR interrupt is triggered to the system and the **SFTY_INT_STS.MEM_AXI_WR_TO_ERR** bit status is set to 1
		SYSTEM	
Address decoding error on DMA	Error response from AXI system	*RESP_ERR[0]*	When the error is detected on the RX message data or acknowledge data being

Error	source	Interrupt	MH behavior
write channels	bus interface, DECERR received on BRESP[1:0] for write access		written, the interrupt *RESP_ERR[0]* interrupt is sent to the system. As the S_MEM is not reliable, the MH stops. The MH finishes all pending data transfers and stops with the **MH_STS.BUSY** = 0. To identify the issue the BRESP[1:0] and the ID of the transaction are logged in the **AXI_ERR_INFO.DMA_ID[1:0]** and **AXI_ERR_INFO.DMA_RESP[1:0]** bit status register.
Address decoding error on DMA read channels	Error response from AXI system bus interface, DECERR received on RRESP[1:0] for read access	*RESP_ERR[1]*	When the error is detected on the TX message data, RX or TX descriptors, the interrupt *RESP_ERR[1]* is sent to the system. As the S_MEM is not reliable, the MH stops. The MH finishes all pending data transfers and stops with the **MH_STS.BUSY** = 0. To identify the issue the RRESP[1:0] and the ID of the transaction are logged in the **AXI_ERR_INFO.DMA_ID[1:0]** and **AXI_ERR_INFO.DMA_RESP[1:0]** bit status register.
System memory CRC error or Access to wrong slave on DMA write channel	Error response from AXI system bus interface, SLVERR received on BRESP[1:0] for write access	*RESP_ERR[0]*	There is no way to identify the exact error source, either a CRC error or a wrong slave access. See “Address decoding error on DMA write channels” description in current table
System memory CRC error or Access to wrong slave on DMA read channel	Error response from AXI system bus interface, SLVERR received on RRESP[1:0] for read access	*RESP_ERR[1]*	There is no way to identify the exact source, either a CRC error or a wrong slave access. See “Address decoding error on DMA read channels” description in current table
A read from the S_MEM cannot complete	The MH does not complete a read within a defined time frame	*DMA_TO_ERR*	When the timeout assigned to the S_MEM AXI read channel fires, the MH finishes all pending data transfers and then stops with **MH_STS.BUSY** = 0. The DMA_TO_ERR interrupt is triggered to the system, with the **SFTY_INT_STS.DMA_AXI_RD_TO_ERR** bit status set to 1
A write to the S_MEM cannot complete	The MH does not complete a write within a defined	*DMA_TO_ERR*	When the timeout assigned to the S_MEM AXI write channel fires, the MH finishes all pending data transfers and then stops with

Error	source	Interrupt	MH behavior
	time frame		**MH_STS.BUSY** = 0. The DMA_TO_ERR interrupt is triggered to the system with the **SFTY_INT_STS.DMA_AXI_WR_TO_ERR** bit status set to 1

1.4.5.30 Interrupts

Interrupt	Description
TX_FQ_IRQ[7:0]	When considering TX FIFO Queues, there is the option, thanks to the IRQ bit field in TX Descriptor, to trigger an interrupt to the system, when a TX message is sent successfully or skipped. This interrupt can only be declared in a TX Header Descriptor (HD=1). When a TX descriptor has HD=0 and IRQ=1, no interrupt is generated. A dedicated interrupt signal *TX_FQ_IRQ[n]* is provided per TX FIFO queue n (0 <= n <=7). When a TX Header Descriptor is mentioning an interrupt (IRQ bit set to 1) and the message is successfully sent or skipped, the DESC_MESSAGE_HANDLER identifies the TX FIFO queue source number of that descriptor and triggers the relative line of the interrupt bus signal. The interrupt will be effective only when the acknowledge data of that descriptor is fully written in S_MEM. It is then possible to define for a TX FIFO Queue n, with a fix number of messages, the interrupt *TX_FQ_IRQ[n]* only to the last Header Descriptor. Doing so, this approach will limit the number of interrupts to the system. The main purpose of the TX FIFO Queue is to append on the fly new messages. A race condition may occur between the SW and the Message Handler regarding the definition of valid TX message in that queue. In case a TX FIFO Queue n does not provide a valid TX descriptor, the MH notifies the SW with the *TX_FQ_IRQ[n]* that the TX FIFO Queue n is on hold, despite being active. The **TX_FQ_INT_STS** register provides the relevant information to detect the root cause. As a summary three different source of events can trig those interrupts: • This interrupt is triggered when the IRQ bit field in TX Header Descriptor is set to 1 and the TX message is sent successfully. The **TX_FQ_INT_STS.SENT[n]** bit register is set to 1 for the TX FIFO Queue n and the bit field STS[3:0] in the TX descriptor is set to 0’b0001 • This interrupt trigs when the TX message is skipped. The **TX_FQ_INT_STS.SENT[n]** bit register is set to 1 for the TX FIFO Queue n and the bit field STS[3:0] in the TX descriptor is set to 0’b0010 or 0’b0011 • The TX FIFO Queue n execution is stopped due to the fetch of an invalid TX descriptor in this queue (no more TX message defined, and no END bit

Interrupt	Description
	set to 1 for the last TX message). The **TX_FQ_INT_STS.UNVALID[n]** bit register is set to 1 for the TX FIFO Queue n
RX_FQ_IRQ[7:0]	When considering RX FIFO Queues there is the option, thanks to the IRQ bit field in RX descriptor, to trigger an interrupt to the system when an RX message is received successfully. The interrupt bus signal *RX_FQ_IRQ[n]* provides an interrupt line for the RX FIFO queue n (0 <= n <=7). When the DESC_MESSAGE_HANDLER fetches an RX descriptor for a given RX message and identifies an IRQ bit set to 1 in one of them, it stores this information. Once the RX message is received successfully and a IRQ bit set has been detected in one RX descriptor, an interrupt is triggered. This interrupt is triggered only when the acknowledge data (written in the Header Descriptor) is fully written in the S_MEM. As a summary there are two options to define this interrupt bit in RX descriptors: • In case the SW requires an interrupt per RX message, the IRQ bit in all RX descriptors must be set to 1. This setting is valid for Normal and Continuous mode with the same effect. • The SW can set the IRQ bit in a regular interval along a RX FIFO Queue, avoiding interrupts at every RX message. Only the RX message covering the RX descriptor having this IRQ bit set will trigger an interrupt. In Continuous mode, it is then possible to set an interrupt every two, three or N messages. In Normal mode, the interrupt could be defined every two, three or N RX descriptors According to the RX message size, several RX descriptors will be used and so could trig the interrupt. It is important to note that RX messages are received with various bit rate, thus the interrupt time interval will not be identical. A race condition may occur between the SW and the Message Handler regarding the definition of valid RX descriptor in a queue. In case a RX FIFO Queue n does not provide a valid RX descriptor in time, the interrupt notifies the SW with the *RX_FQ_IRQ[n]* interrupt that the RX FIFO Queue n is on hold despite being active. The **RX_FQ_INT_STS** register provides the related information to identify the root cause. As a summary two different source of events can trig those interrupts: • This interrupt is triggered when the IRQ bit field in a RX Descriptor is set to 1 and the RX message is received successfully. The **RX_FQ_INT_STS.RECEIVED[n]** bit register is set to 1 for the RX FIFO Queue n and the bit field STS[3:0] in the RX descriptor is set to 0’b0001 • The RX FIFO Queue n execution is stopped due to the fetch of an invalid RX descriptor in this queue. The **RX_FQ_INT_STS.UNVALID[n]** bit register is set to 1 for the TX FIFO Queue n

Interrupt	Description
TX_PQ_IRQ	A single *TX_PQ_IRQ* interrupt is assigned to all TX Priority Queue slots. Any TX Priority Queue slot can trigger an interrupt (IRQ = 1) when the relative TX message is successfully sent or skipped. Any TX Header Descriptor having the IRQ bit set and used by the TX Priority Queue trigs this interrupt line. When a new TX message is defined in a TX Priority Queue slot, the TX descriptor used to define this message must be valid. When the message is sent, the slot is set to inactive and nothing else can occur. Whereas the TX FIFO Queue, which is processing up to the point a TX descriptor is invalid, the TX Priority Queue slot must not fetch any invalid descriptor. To protect the execution of TX message and to have a common TX Queue management, the TX priority Queue can also report invalid descriptor. The SW need to look at the interrupt status register **TX_PQ_INT_STS0** and **TX_PQ_INT_STS1** to identify which slot has generated the interrupt and for which reason. As a summary three different source of events can trig this interrupt: • This interrupt is triggered when the IRQ bit field in TX Header Descriptor is set to 1 and the TX message is sent successfully. The **TX_PQ_INT_STS0.SENT[n]** bit register is set to 1 for the TX Priority Queue slot n and the bit field STS[3:0] in the TX descriptor is set to 0'b0001 • This interrupt is triggered when the TX message is skipped. The **TX_PQ_INT_STS0.SENT[n]** bit register is set to 1 for the TX Priority Queue slot n and the bit field STS[3:0] in the TX descriptor is set to 0'b0010 or 0'b0011 • The TX Priority Queue slot n execution is stopped due to the fetch of an invalid TX descriptor in this queue (TX descriptor is not valid). The **TX_PQ_INT_STS1.UNVALID[n]** bit register is set to 1 for the TX Priority Queue slot n
STATS_IRQ	Four Statistic counters are used to monitor successful and unsuccessful RX and TX messages. As soon as one of those counters overflows the *STATS_IRQ* is triggers to the system, refer to the RX and TX Statistics chapter for more details. When looking at the **STATS_INT_STS** register, the SW can identify which counter has reached its maximum value: • When the number of unsuccessful RX message received has reached the maximum counter value, the **STATS_INT_STS.RX_UNSUCC** is set to 1 • When the number of successful RX message received has reached the maximum counter value, the **STATS_INT_STS.RX_SUCC** is set to 1 • When the number of unsuccessful TX message received has reached the maximum counter value, the **STATS_INT_STS.TX_UNSUCC** is set to 1 • When the number of successful TX message received has reached the maximum counter value, the **STATS_INT_STS.TX_SUCC** is set to 1

Interrupt	Description
STOP_IRQ	When the PRT is stop (*ENABLE* signal goes from high to low), the MH finishes its current tasks. It puts all active RX/TX FIFO Queues on hold and discard all active TX Priority Queue slots. Once done, the MH notifies such state by triggering the STOP_IRQ interrupt. The interrupt *STOP_IRQ* is raised under the following conditions only: • **TX_FQ_STS0** = 0x0000 and **RX_FQ_STS0** = 0x0000 and **TX_PQ_STS0** = 0x00000000 • **TX_FQ_STS0** = 0xXYXY and **RX_FQ_STS0** = 0xWVWV and **TX_PQ_STS0** = 0x00000000, where XY defined the active and on hold TX FIFO Queues and WV the active and on hold RX FIFO Queues
RX_FILTER_IRQ	In order to track RX filtering results, an interrupt can be defined when a match is detected on any defined RX filter element. The *RX_FILTER_IRQ* can only be triggered if the IRQ bit in the RX filter element is set to 1 AND there is a match. When a match is detected, the FM bit (set in RX message header) is set to 1 and the filter element index is defined in the FIDX[7:0] bit field (set in the RX message header). Note: The BLK bit field in the RX Filter element is a side band information and is not considered for the interrupt generation.
TX_FILTER_IRQ	The interrupt is triggered when the TX filter is enabled, and a TX message is rejected. Despite being rejected, the TX descriptor used to define the TX message is acknowledged. To identify the TX descriptor allocated to the TX message rejected, the STS[3:0] bit field in the TX descriptor is set to 0'b0100. The **TX_FILTER_ERR_INFO** register provides the relevant information to identify which TX FIFO Queue or TX Priority Queue slot is impacted.
TX_ABORT_IRQ	This interrupt line is only triggered when the MH needs to abort a TX message being sent to the PRT. This interrupt does not have any status flags, as it will always be linked to functional or safety errors. Thus, another interrupt will provide the require information related to the issue. Several source of events can lead to this interrupt: • TX address pointer parity error (refer to AP_PARITY_ERR interrupt) • Timeout on S_MEM, L_MEM or PRT interface (refer to MEM_TO_ERR, DMA_TO_ERR or DP_TO_ERR interrupt) • DMA channel routing error (refer to DMA_CH_ERR interrupt) • A TX_MSG sequence error (refer to DP_SEQ_ERR interrupt) • A DMA AXI or MEM AXI error response (refer to RESP_ERR interrupt) • An uncorrectable error detected on the L_MEM (refer to MEM_SFTY_ERR interrupt) • A TX data parity error (refer to DP_PARITY_ERR interrupt) Aborting a TX FIFO Queue or a TX Priority Queue slot does not set this interrupt as no TX message abort is expected to occur (the MH will complete the current TX message before aborting the TX FIFO Queue).

Interrupt	Description
RX_ABORT_IRQ	This interrupt line is triggered when the MH needs to abort a RX message received from the PRT. This interrupt does not have any status flags, as it will always be linked to functional or safety errors. Thus, another interrupt will provide the require information related to the issue. Several source of events can lead to this interrupt: • An RX message is about to be sent to a disabled RX FIFO Queue. • An RX message is in progress and the MH receives a new RX message at the same time. • RX address pointer parity error (refer to AP_PARITY_ERR interrupt) • Timeout on S_MEM, L_MEM or PRT interface (refer to MEM_TO_ERR, DMA_TO_ERR or DP_TO_ERR interrupt) • DMA channel routing error (refer to DMA_CH_ERR interrupt) • A RX_MSG sequence error (refer to DP_SEQ_ERR interrupt) • A DMA AXI or MEM AXI error response (refer to RESP_ERR interrupt) • An uncorrectable error detected on the L_MEM (refer to MEM_SFTY_ERR interrupt) • A RX data parity error (refer to DP_PARITY_ERR interrupt) • A RX descriptor error (refer to DESC_ERR interrupt) • An overflow on RX DMA FIFO or on the RX descriptor acknowledge path (refer to the DP_DO_ERR interrupt) Aborting a RX FIFO Queue will never set this interrupt, as the MH will complete its current reception before this action.
RX_FILTER_ERR	This interrupt line is triggered when the RX filter has not finished in time, to define the RX FIFO Queue number, before the reception of a new RX message. It provides information to the SW about large RX filtering time. Refer to the RX Filter chapter for detailed description. There is no status flag related to this interrupt, as the second source of event, defined below, is a programming issue and should never occur. Two different sources of events can trig this interrupt: • RX filtering not finished before a new RX frame • RX FIFO Queue to receive RX frame not running
MEM_SFTY_ERR	Safety error detected at the L_MEM interface. In fact, this interrupt is triggered when either the *MEM_SFTY_CE* or *MEM_SFTY_UE* input signal is active. To identify the root cause of such interrupt, refer to the **SFTY_INT_STS** register Two different sources of events can trig this interrupt: • The *MEM_SFTY_UE* input signal, when set, to indicate an uncorrectable error from the L_MEM when reading (this signal must be generated by the L_MEM memory controller). The **SFTY_INT_STS. MEM_SFTY_UE** bit register is set to 1 in this case • The *MEM_SFTY_CE* input signal, when set, to indicate a correctable error from the L_MEM when reading (this signal must be generated by the

Interrupt	Description
	L_MEM memory controller). The **SFTY_INT_STS. MEM_SFTY_CE** bit register is set to 1 in this case
REG_CRC_ERR	CRC error detected on the register bank. This interrupt is triggered after a few cycles if the CRC written in the **CRC_REG.VAL[31:0]**, prior writing 1 to the **CRC_CTRL.START** bit, is not matching the one computed in hardware. Such interrupt event does not trig any actions in the MH. Therefore, it is a SW task to do the appropriate actions to stop the MH.
DESC_ERR	CRC error detected on RX/TX descriptor or unexpected RX/TX descriptor received. Status flags allow SW to identify the root cause of such interrupt, see **SFTY_INT_STS** register. Several source issues could lead to this interrupt: • When the **SFTY_INT_STS.RX_DESC_CRC_ERR** is set to 1, a RX descriptor is received and is having a CRC error • When the **SFTY_INT_STS.RX_DESC_REQ_ERR** is set to 1, a RX descriptor is received and is not compliant to the one requested (wrong RX FIFO Queue, wrong instance number, wrong position in the queue, …) • When the **SFTY_INT_STS.TX_DESC_CRC_ERR** is set to 1, a TX descriptor is received and is having a CRC error • When the **SFTY_INT_STS.TX_DESC_REQ_ERR** is set to 1, a TX descriptor is received and is not compliant to the one requested (wrong TX FIFO Queue, wrong instance number, wrong position in the queue, wrong TX Priority Queue slot…) The **DESC_ERR_INFO0** and **DESC_ERR_INFO1** registers provide a detailed description of the faulty RX/TX descriptor. Only the first RX/TX descriptor error will lead to an update of those registers, in case several ones occur. To capture the next descriptor error information, the SW must clear the interrupt source.
AP_PARITY_ERR	Address pointers used to manage TX FIFO Queues, RX FIFO Queues and TX Priority Queue are protected using parity bit (1bit per byte). Any issue detected trigs this interrupt. The parity bits are checked only when the address pointer is used for S_MEM accesses. Status flags allow SW to identify the root cause, see **SFTY_INT_STS** register. Several source issues could lead to this interrupt: • When the **SFTY_INT_STS.AP_RX_PARITY_ERR** is set to 1, an address pointer used to manage the RX path is having a parity error • When the **SFTY_INT_STS.AP_TX_PARITY_ERR** is set to 1, an address pointer used to manage the TX path is having a parity error
DP_PARITY_ERR	Parity error detected on RX message data received from PRT to AXI system bus or TX payload data transmitted from AXI system bus to PRT. Any issue detected trigs this interrupt. Status flags allow SW to identify the root cause, see **SFTY_INT_STS**

Interrupt	Description
	register. Several source issues could lead to this interrupt: • When the **SFTY_INT_STS.DP_RX_PARITY_ERR** is set to 1, a RX message data is having a parity error • When the **SFTY_INT_STS.DP_TX_PARITY_ERR** is set to 1, a TX message data is having a parity error
DP_SEQ_ERR	The RX_MSG or TX_MSG interface used to synchronize the MH and PRT data exchange is not functional. A wrong PRT or MH behavior could lead to this issue. A problem on the logic managing the clock domain crossing on RX_MSG or TX_MSG interface may be one of the source issues. Status flags are available to identify the faulty interface, see **ERR_INT_STS** register. Several source issues could lead to this interrupt: • When the **ERR_INT_STS.DP_RX_SEQ_ERR** is set to 1, an issue is detected on the RX_MSG interface • When the **ERR_INT_STS.DP_TX_SEQ_ERR** is set to 1, an issue is detected on the TX_MSG interface
DP_DO_ERR	An overflow is detected on the RX data path or while acknowledging a RX/TX descriptor. Some status flags are provided to identify the interrupt source, see **ERR_INT_STS** register Several source issue could trig this interrupt: • When the **ERR_INT_STS.DP_RX_FIFO_DO_ERR** is set to 1, an RX DMA FIFO overflow is detected. Several reasons could explain such issue: a very high system latency (over the expected limit considered for the MH), a system memory no more accessible and a wrong MH behavior. • When the **ERR_INT_STS.DP_RX_ACK_DO_ERR** is set to 1, an ACK DMA FIFO overflow is detected. Such issue occurs when the acknowledgment of an RX descriptor is not possible due to some pending ones. A system memory not accessible or a wrong MH behavior (DMA controller not functional, deadlock on RX/TX acknowledge path) could explain such issue. • When the **ERR_INT_STS.DP_TX_ACK_DO_ERR** is set to 1, an ACK DMA FIFO overflow is detected. Such issue occurs when the acknowledgment of a TX descriptor is not possible due to some pending ones. A system memory not accessible or a wrong MH behavior (DMA controller not functional, deadlock on RX/TX acknowledge path) could explain such issue .
DP_TO_ERR	When the PRT is not responding after a certain amount of time, either on RX or on TX path, the *DP_TO_ERR* interrupt is triggered. The counter on RX_MSG or TX_MSG interface starts with the Start Of Frame and stop when receiving the timestamp. The timeout value is programmable by SW, refer to the Programming Guidelines chapter for more details. Some status flags provide the interrupt source, see **SFTY_INT_STS** register. Several source issue could trig this interrupt:

Interrupt	Description
	• When the **SFTY_INT_STS.DP_PRT_RX_TO_ERR** is set to 1, the timeout value defined on the RX_MSG interface is over. The PRT or MH may be locked, preventing data reception • When the **SFTY_INT_STS.DP_PRT_TX_TO_ERR** is set to 1, the timeout value defined on the TX_MSG interface is over. The PRT or MH may be locked, preventing data transmission .
DMA_TO_ERR	When the S_MEM is not responding after a defined time interval, the *DMA_TO_ERR* is triggered. The timeout value is programmable by SW, refer to the Programming Guidelines for more details. Some status flags provide the interrupt source, see SFTY_INT_STS register. Several source issue could trig this interrupt: • When the **SFTY_INT_STS.DMA_AXI_RD_TO_ERR** is set to 1, the timeout value defined on the DMA AXI read channel interface is over. A system memory no more accessible or a DMA controller in deadlock could explain such issue. • When the **SFTY_INT_STS.DMA_AXI_WR_TO_ERR** is set to 1, the timeout value defined on the DMA AXI write channel interface is over. A system memory no more accessible or a DMA controller in deadlock could explain such issue.
MEM_TO_ERR	When the L_MEM is not responding after a defined time interval the *MEM_TO_ERR* is triggered. The timeout value is programmable by SW, refer to the Programming Guidelines for more details. Some status flags are provided to identify the interrupt source, see SFTY_INT_STS register. Several source issue could trig this interrupt: • When the **SFTY_INT_STS.MEM_AXI_RD_TO_ERR** is set to 1, the timeout value defined on the MEM AXI read channel interface is over. A local memory no more accessible or a memory controller in deadlock could explain such issue. • When the **SFTY_INT_STS.MEM_AXI_WR_TO_ERR** is set to 1, the timeout value defined on the MEM AXI write channel interface is over. A local memory no more accessible or a memory controller in deadlock could explain such issue.
DMA_CH_ERR	Data received or sent are not routed to or from the right DMA channels. Such issue will lead to data corruption and wrong MH behavior. There are no status flags to identify the source channel being faulty.
RESP_ERR[1:0]	Any error response from the DMA AXI and MEM AXI interfaces can lead to a RESP_ERR[1:0] interrupts. Some status flags provide the interrupt source, see SFTY_INT_STS register. Several source issue could trig those interrupts: • When the RESP_ERR[0] interrupt is set, a write access error is detected on either the DMA_AXI or MEM_AXI write channel.

Interrupt	Description
	• When the RESP_ERR[1] interrupt is set, a read access error is detected on either the DMA_AXI or MEM_AXI read channel. The **AXI_ERR_INFO** register provides a detailed description of the faulty AXI interface, refer to the **AXI_ERR_INFO.MEM_RESP[1:0]** or **AXI_ERR_INFO.DMA_RESP[1:0]** bit field to determine which one (must be different from 0'b00). The traffic getting the error response is defined when looking at the **AXI_ERR_INFO.MEM_ID[1:0]** (if **AXI_ERR_INFO.MEM_RESP[1:0]** is different from 0'b00) or **AXI_ERR_INFO.DMA_ID[1:0]** bit field (if **AXI_ERR_INFO.DMA_RESP[1:0]** is different from 0'b00). In case several response errors occur on the same interface, only the AXI ID of the last one is captured.

1.4.5.31 Clock and Reset

There is only one clock *CLK* to drive the whole core logic.
A clock *CLK_AXI* is used at the host interface, this clock is synchronous to *CLK* clock.
The only reset available is the *RESET_N* signal, it is asynchronously asserted (set to low) and synchronously de-asserted.

To lower power consumption, the MH can have its core clock *CLK* disabled. Registers can still be programmed through the host interface (*CLK_AXI* clock still active)

The *CLK* and *CLK_AXI* are used only with the rising edge, this means they can be defined with some spread to lower EMI.

As the MH uses only the rising edge of the CLK_AXI and CLK, the duration of the clock's high pulse may vary between 10% and 90% of the clock period during operation.

The PRT signalizes via *ENABLE* whether it is active and requires message handling or not.

- When this signal is going low, the MH will stop its current activities. This means the RX/TX FIFO queues and TX Priority Queue are put on hold as well as all the relevant traffic from and to the S_MEM. Once it is done, the bit status **MH_STS.BUSY** is set to 0 and the *CLK* clock signal of the MH can be stopped.
- When the ENABLE is already low and the MH_STS.BUSY bit status is set to 0, nothing prevents the SW to switch off the *CLK* clock signal.

1.4.6 Application Information

This section describes some general information related to MH performances, flags to be look at and cluster guidelines.

1.4.6.1 Queue Status Flags

The TX FIFO Queue status is defined according to the bit status in the **TX_FQ_STS0** register, see table below:

TX_FQ_STS0.BUSY[n]	TX_FQ_STS0.STOP[n]	Status for TX FIFO QUEUE n (n € {0, 1, ..., 31)
0	0	Inactive: The TX FIFO Queue can be programmed and started if enabled
0	1	na
1	0	Active and running: The TX FIFO Queue is enabled and has been started. TX messages are sent whenever possible to the PRT
1	1	Active and on hold: When considering no functional or safety errors, this status is reached when an invalid TX descriptor is fetched from S_MEM.

The TX Priority Queue slot n status is defined according to the bit status in the **TX_PQ_STS0** register, see table below:

TX_PQ_STS0.BUSY[n]	Status for TX PRIORITY QUEUE slot n (n € {0, 1, ..., 31)
0	Inactive: The TX Priority Queue slot n can be programmed and started if enabled
1	Active and running: The TX Priority Queue slot n is enabled and has been started. TX message in the slot n can be transmitted whenever possible

Compared to the TX FIFO Queues, there is no STOP bits. Any errors related to a TX Priority Queue slot execution sets the slot as inactive.

The RX FIFO Queue status is defined according to the bit status in the **RX_FQ_STS0** register, see table below:

RX_FQ_STS0.BUSY[n]	RX_FQ_STS0.STOP[n]	Status for RX FIFO QUEUE n (n € {0, 1, ..., 31)
0	0	Inactive: The RX FIFO Queue can be programmed and started if enabled
0	1	na
1	0	Active and running: The RX FIFO Queue is enabled and has been started. RX messages can be received from the PRT
1	1	Active and on hold: When considering no functional or safety errors, this

RX_FQ_STS0.BUSY[n]	RX_FQ_STS0.STOP[n]	Status for RX FIFO QUEUE n (n € {0, 1, ..., 31)
		status is reached when an invalid RX descriptor is fetched from S_MEM.

1.4.6.2 Cluster

The same L_MEM can be shared across several Message Handler, but several points need to be highlighted. A trade-off needs to be found to ensure every MH will get enough time to complete their RX filter process as well as their TX-Scan for a given L_MEM bandwidth.

- The worst scenario on RX path is defined when all MH in a cluster is receiving an RX message at the same time. Therefore, it is essential to ensure the available bandwidth on the L_MEM is able to support the RX filter process from all concurrent MH. Several measures can be taken to lower the bandwidth for a given value: limit the number of RX filter elements and the number of comparison (1 or 2) per filter element.
- The worst scenario on TX path is defined by all TX FIFO queues active for every MH as well as new TX messages being added to all TX Priority Queue slots. As one message is added or sent at a time for every MH, the impact of the TX-Scan may be limited but may play an important role by generating more arbitration occurrences
- The read latency to access the L_MEM is a common factor for all MH and should be as low as possible. This access time is driven the overall performances when in cluster mode

1.4.6.3 Performances

Several processing times have a direct impact on the overall MH performances, see sections below.

1.4.6.3.1 Core Clock Frequency

The minimum MH core clock frequency is driven by several parameters:

- The maximum number of RX filter elements to support
- The Classical CAN, CAN FD, and CAN XL bit rates (Arbitration and Data Phase)
- The L_MEM read latency
- The maximum number of TX FIFO Queues
- The maximum number of TX Priority Queue slots

To estimate the minimum core clock frequency to set, please refer to excel file [6]. One must keep in mind that the computed value is a minimum. Other clock frequency constraints may require a higher clock speed on the MH.

1.4.6.3.2 TX-Scan

The TX-Scan does compute the highest priority message over the TX FIFO queues and the TX Priority Queue slots. The processing time is mainly link to the number of TX FIFO Queues active at the same time as well as the number of TX Priority Queue slots being set active. The higher the number of slots and TX FIFO queues active, the higher the bandwidth from the L_MEM. The sooner the result is known the better the expected transmission order is. For more detail on TX-Scan refer to the TX-Scan chapter.

1.4.6.3.3 RX Filter

As the RX filter elements are defined and read from the L_MEM, any RX message received will generate many accesses. The number of RX filter elements and the number of comparisons per element drive the bandwidth from the L_MEM and so the processing time. The higher the number of filter elements the higher it takes to define if an RX message is accepted or rejected. Despite some measures are in place to avoid discarding the current RX message, the SW would need to sort the non-dispatched messages later on. For more detail on RX Filter refer to the RX Filter chapter.

The process of filtering is started as soon as the first RX message header data is received. When an RX filter element expects an RX data word that is not already stored, the process stops and waits for the RX data word. As the RX filter element are fetched linearly from the L_MEM, it is required to have them organized in a specific way to optimize the filtering time. The Classical CAN with a low bit rate does provide more margin to complete the RX filtering in time. The critical path is defined when receiving CAN FD frame with no payload data.
As a general rule, it is recommended to defined RX filter elements in this order:

- First: CAN FD, assuming that only one comparison with the first message header word is required
- Second: CAN XL, assuming either one or two comparisons could be defined
- Third: Classical CAN , assuming that only one comparison with the first message header word is require

Such RX filter elements organization will optimize the overall processing time.

1.4.6.3.4 RX/TX Descriptors Memory Organization

RX/TX descriptors are fetched from the S_MEM. The DMA controller is reading and writing data to the S_MEM using burst length of various sizes. As soon as the address to read or write data is aligned on burst length of 8, all the following burst transfer are using maximum burst length of 8. If the address to fetch the RX/TX descriptor does cross a burst of 8 boundary, two read accesses are required.

TX path:
As the TX header descriptor is store locally in the MH there is no constraint regarding the access time from the S_MEM. Nevertheless, several recommendations will help to increase access performances and to limit power consumption:

- Align TX FIFO Queue start address on maximum burst length (8 word of 32bit)
- Align TX Priority Queue start address on maximum burst length (8 word of 32bit)

- Define TX FIFO/Priority Queues (linked list of TX descriptors) in SRAM to leave more time for payload data fetching. This is best practice to declare TX descriptor in SRAM whenever possible

RX path:
RX descriptors fetches are driving the RX messages write to S_MEM. On top of it, if the RX filtering is taking too much time, an RX DMA FIFO overflow may occur. Several recommendations will help to increase access performances and to limit power consumption:

- Align RX FIFO Queue start address on maximum burst length (8 word of 32bit)
- Define RX FIFO Queues (linked list of RX descriptors) in SRAM to leave more time for RX filtering. This is best practice to declare RX descriptor in SRAM whenever possible

1.4.6.3.5 Data Payload Buffer Memory Organization

Any accesses done from or to the S_MEM by the DMA will be fully optimized is the address is aligned on the maximum burst length (8x32bit).
TX path:

- Align data container start address on maximum burst length (8 word of 32bit)
- Use data container size multiple of maximum burst length (8 word of 32bit)

RX path:

- Align data container start address on maximum burst length (8 word of 32bit), whatever the mode (Normal or Continuous)
- Use data container size multiple of maximum burst length (8 word of 32bit)

1.4.6.3.6 High System Memory Latency

The maximum system memory latency is driven by the Classical CAN, CAN FD, and CAN XL bit rates (Arbitration and Data Phase) and can be computed using the excel file [6].

If the latency time to get the first payload data burst is greater than the computed value, an underrun will occur when starting to transmit a TX message to the PRT. As many DMA requests may occur to the system bus at the same time, some critical scenarios could lead to delay the fetch of the first payload data, providing underrun. If one of the TX descriptor DMA requests is pre-empting the access to the first payload data for the current TX message, the delay would be larger than the one expected. As an example, starting all TX FIFO Queues and TX Priority Queue slots at the same time may increase the probability to have an underrun.

As a matter of fact, very high system latency may lead to underrun due to the high constraints on burst accesses. The data underrun is a warning and won't affect the MH behavior and the order of the TX messages. No TX message with underrun is dropped and it will still be considered in the next TX-Scan run.
Nevertheless, here below are a list of recommendations to avoid and limit issues in a system with high latency:

TX path:
Every TX descriptor and its payload data are fetched from the S_MEM. The payload data is only read from the S_MEM when the TX message (defined by its TX descriptor) started to be transmitted on the CAN bus (meaning the message has won the CAN bus arbitration). Thus, in case of very high latency system, a data underrun may occur on the PRT. As a matter of fact, the critical path is defined by the first bunch of payload data to be fetched. Several actions can be done to cope with high system latency:

- Align data container start address on maximum burst length (8 word of 32bit)
- Align TX FIFO Queue start address on maximum burst length (8 word of 32bit)
- Align TX Priority Queue start address on maximum burst length (8 word of 32bit)
- Use data container size multiple of maximum burst length (8 word of 32bit)
- The usage of write outstanding transaction may not provide a significant improvement as the write accesses are somehow sequential. Nevertheless, it is recommended to set it to the maximum value, see **AXI_PARAMS.AW_MAX_PEND[1:0]** bit register
- Make use of read outstanding transaction (this is mandatory to avoid cases where DMA channel are competing against each other to read the S_MEM). The maximum value is recommended, see **AXI_PARAMS.AR_MAX_PEND[1:0]** bit register
- Define TX FIFO/Priority Queues (linked list of TX descriptors) in SRAM to leave more time for payload data fetching. This is best practice to declare descriptor in SRAM whenever possible

RX path:
Every RX descriptor is fetched from the S_MEM. As for the TX path, the critical path is defined by the first RX descriptor to be fetched, once the RX FIFO Queue number is defined by the RX filter. As soon as the RX FIFO Queue is known, there is still some time required to read the corresponding RX descriptor and to write the data payload to the S_MEM. To avoid any RX DMA FIFO overflow and to limit the constraints at system level, the faster the RX descriptor is read from the S_MEM the faster the payload data can be written to the S_MEM. Nevertheless, several actions can be done to cope with high system latency:

- Align data container start address on maximum burst length (8 word of 32bit)
- Align RX FIFO Queue start address on maximum burst length (8 word of 32bit)
- Use data container size multiple of maximum burst length (8 word of 32bit)
- The usage of write outstanding transaction may not provide a significant improvement as the write accesses are somehow sequential and the RX DMA FIFO sized to support high latency. Nevertheless, it is recommended to set it to the maximum value, see **AXI_PARAMS.AW_MAX_PEND[1:0]** bit register
- Make use of read outstanding transaction (this is mandatory to avoid cases where DMA channel are competing against each other to read the S_MEM). The maximum value is recommended, see **AXI_PARAMS.AR_MAX_PEND[1:0]** bit register
- Define RX FIFO Queues (linked list of RX descriptors) in SRAM to shorter the reaction time when receiving an RX message. This is best practice to declare descriptor in SRAM whenever possible

1.4.7 Programming Guidelines

Here below are the general procedures to program the MH.

1.4.7.1 Initial MH Start Procedure

Before starting any process, the SW driver must ensure the MH core clock is active (**MH_STS.CLOCK_ACTIVE** = 1). It is assumed that the *CLK_AXI* clock to access to the MH register bank is up and running.
Here is the procedure:
1) Configure the MH global registers:
- The **MH_CFG.INST_NUM[2:0]** bit field to indicate the X_CAN instance number
- The **MH_SFTY_CFG** and **MH_SFTY_CTRL** registers according to the safety measures to apply
- The **AXI_ADD_EXT** register if the *DMA_AXI* address bus interface is greater than 32bit
  The **AXI_PARAMS** register to define read and write outstanding
- The **RX_STATISTICS** and **TX_STATISTICS** registers must be set to 0 to ensure status of new transmissions and receptions (successful or unsuccessful) are properly incremented in those counters

2) Configure the RX Filter, see RX Filter Setting section
3) Configure the TX Filter, see TX Filter Setting section
4) Configure the RX FIFO Queues up to the start step, see RX FIFO Queue Initial Start section
5) Configure the TX FIFO Queues up to the start step, see TX FIFO Queue Initial Start section
6) Prepare TX Priority Queue, see TX Priority Queue Initialization section and define TX Priority Queue slots (if any) up to the start step, see Starting TX Priority Queue Slot section
7) Unmask error and safety interrupts as well as functional interrupts which are relevant in the interrupt controller
8) Compute the CRC of the registers protected by CRC and write the value to the **CRC_REG** register. Then, write to the **CRC_CTRL.START** bit register to do the CRC checking. Any CRC issue triggers an *REG_CRC_ERR* interrupt to the system
9) Write 0b1 to the **MH_CTRL.START** bit register to start the MH. As long as the PRT is not started (**MH_STS.ENABLE** = 0) and there is no active RX/TX FIFO queue or TX Priority Queue slot (**MH_STS.BUSY** = 0), the **MH_CTRL.START** bit can still be set back to 0
10) Start the RX FIFO Queues writing to the **RX_FQ_CTRL0** register. Once a RX FIFO Queue starts, the registers related to that queue and defined in 4) are write protected (excepted the **RX_FQ_CTRL2** register). As soon as one RX FIFO Queue is active, the **MH_CTRL.START** bit register cannot be modified. This means, it is no more possible to stop the MH without stopping the PRT
11) Start the PRT and wait for the **MH_STS.ENABLE** to be set to 1, meaning the PRT is up and running
12) Start the relevant TX FIFO Queues writing to **TX_FQ_CTRL0** register. Once a TX FIFO Queue starts, the registers related to that queue and defined in 5) are write protected (excepted the **TX_FQ_CTRL2** register)
13) Start the relevant TX Priority Queue slots writing to the **TX_PQ_CTRL0** register

1.4.7.2 Stopping MH Procedure

As the PRT is the physical link to the CAN bus, once everything is started (PRT and MH), the MH can stop only if the PRT is switched off (*ENABLE* signal going from High to Low leading to **MH_STS.ENABLE** bit status set to 0).
For more details on the different conditions to have the CAN protocol operation stopped (namely normal stop, immediate stop or Bus-Off), refer to the PRT chapter. As there is no way for the MH to identify the exact root cause of the PRT stop, the MH performs some actions and will leave the SW to finalize the procedure defined in section Full Stop:

Actions done by the MH when the PRT is switched off:

- When receiving a RX message and if the PRT is having an immediate stop, the RX message is discarded. An *RX_ABORT_IRQ* interrupt is generated
- When receiving a RX message and if the PRT is having a normal stop, the RX message reception is completed
- When transmitting a TX message and if the PRT is having an immediate stop, the TX message is aborted. A *TX_ABORT*_IRQ interrupt is generated
- When transmitting a TX message and if the PRT is having a normal stop, the TX message transmission is completed
- All active RX FIFO Queue n (n € {0, 1, ..., 7}) are put on hold, this means no RX message can be received. The **RX_FQ_STS0.STOP[n]** and **RX_FQ_STS0.BUSY[n]** bit registers for those queues are set to 1. The other inactive RX FIFO Queues status flags (busy and stop) remain set to 0.
- All active TX FIFO Queue n (n € {0, 1, ..., 7}) are put on hold, this means no TX message can be transmitted. The **TX_FQ_STS0.STOP[n]** and **TX_FQ_STS0.BUSY[n]** status registers for those TX FIFO Queues are set to 1. The other inactive TX FIFO Queues status flags (busy and stop) remain set to 0.
- All TX Priority Queue slots are set inactive, this means the **TX_PQ_STS0.BUSY[31:0]** status register is set to 0x00000000

1.4.7.2.1.1 Full Stop

The RX/TX FIFO Queues linked list are set back to their initial value defined by their respective registers. To complete the procedure, the SW must abort all the RX/TX FIFO Queues being still actives, see Aborting RX FIFO Queue and Aborting TX FIFO Queue sections. Once done, the **MH_STS.BUSY** bit register is set to 0. With the **MH_STS.BUSY** = 0 and the **MH_STS.ENABLE** = 0, the MH can be stopped writing 0 to the **MH_CTRL.START** bit register. At this point, the MH can be entirely reprogrammed.

Summary for a full stop of the MH:
1) Stop PRT
2) Abort all TX Priority Queue slots
3) Abort all RX FIFO Queues
4) Abort all TX FIFO Queues

5) Write **MH_CTRL.START**=0 (unlocks the MH global configuration registers)

1.4.7.3 RX FIFO Queue Initial Start

For the RX FIFO Queues, some common configuration registers need to be set prior any start. It is essential to note that those registers are write-protected when the MH is started (**MH_CFG.START** = 1):

- The **MH_CFG.INST_NUM[2:0]** bit field to indicate the X_CAN instance number
- The **MH_CFG.RX_CONT_DC** bit register to select either the Normal or Continuous mode for all RX FIFO Queues
- The **MH_SFTY_CFG** and **MH_SFTY_CTRL** registers according to the safety measures to apply
- The **AXI_ADD_EXT** register if the *DMA_AXI* address bus interface is greater than 32bit
- The **AXI_PARAMS** register to define read and write outstanding
- The **RX_FILTER_MEM_ADD** register to define address of the RX Filter elements and RX Filter reference/mask address in the L_MEM
- The **RX_FILTER_CRTL** register to define how the RX filter must behave when enabled

Before starting a RX FIFO Queue n in Normal mode several configuration registers need to be defined:

- The **RX_FQ_START_ADD{n}** register defines the address of the First RX Descriptor of the linked list
- The **RX_FQ_SIZE{n}.MAX_DESC** bit field register provides the size of the linked list in number of RX descriptors. The memory area is then computed with the formula:
  RX_FQ_SIZE{n}.MAX_DESC * 16byte
- The **RX_FQ_SIZE{n}.DC_SIZE** bit field register provides the size of the data container attached to every RX descriptor
- The **RX_FQ_CTRL2.ENABLE[n]** bit register to enable the RX FIFO Queue n before a start.

Before starting a RX FIFO Queue n in Continuous mode several configuration registers need to be defined:

- The **RX_FQ_START_ADD{n}** register defines the address of the First RX Descriptor of the linked list
- The **RX_FQ_SIZE{n}.MAX_DESC** bit field register provides the size of the linked list in number of RX descriptors. The memory area is then computed with the formula:
  RX_FQ_SIZE{n}.MAX_DESC * 16byte
- The **RX_FQ_SIZE{n}.DC_SIZE** bit field register provides the size of the single data container attached to all RX descriptors
- The **RX_FQ_DC_START_ADD{n}** register provides the base address of the single data container defined for the RX FIFO Queue n and attached to all RX descriptors of that queue
- The **RX_FQ_RD_ADD_PT{n}** register provides the read address pointer used by the SW to read an RX message in the data container
- The **RX_FQ_CTRL2.ENABLE[n]** bit register to enable the RX FIFO Queue n before a start

In order to define and start a RX FIFO Queue n, do the following:

- A RX FIFO Queue can only start if the **MH_CTRL.START** bit register is set to 1 (refer to the Initial MH Start Procedure section)
- The SW must check the **RX_FQ_STS0.BUSY[n]** and **RX_FQ_CTRL2.ENABLE[n]** (n € {0, 1, ..., 7}) bit registers are set to 0 (RX FIFO Queue n not already active and enabled). With such bit configuration the **RX_FQ_STS0.STOP[n]** bit register must be equal to 0
- Configure the start address of the RX descriptor linked list, for the RX FIFO Queue n, using the **RX_FQ_START_ADD{n}** (n € {0, 1, ..., 7}) register
- Configure the maximum number of defined RX descriptors in the linked list, for the RX FIFO Queue n, using the **RX_FQ_SIZE{n}.MAX_DESC[9:0]** (n € {0, 1, ..., 7}) register. The memory size allocated is expected to be **RX_FQ_SIZE{n}.MAX_DESC[9:0]** * 16byte (RX descriptor size). To avoid changing the rolling counter bit field in the RX descriptor (see RC[4:0] in RX descriptor chapter), from time to time and after a wrapping, set the **RX_FQ_SIZE{n}.MAX_DESC[9:0]** bit field as a multiple of 32
- Defined in S_MEM the RX descriptors linked list for the RX FIFO Queue n. Those RX descriptors are continuous in S_MEM and must be prepared before starting the RX FIFO Queue. Only valid RX descriptors (VALID bit set to 0 in descriptor) can ensure the reception of RX messages
- In case of Normal mode, a dedicated RX data container must be defined in S_MEM per RX descriptor. For a given RX FIFO Queue, data containers are of the same size and defined in **RX_FQ_SIZE{n}.DC_SIZE[6:0]** bit field register. The memory size expected per RX data container is equal to **RX_FQ_SIZE{n}.DC_SIZE[6:0]** * 32byte
- In case of Continuous mode, only a single RX data container must be declared for a RX FIFO Queue n. The **RX_FQ_DC_START_ADD{n}** (n € {0, 1, ..., 7}) register defines the start address of that container and the **RX_FQ_SIZE{n}.DC_SIZE[11:0]** (n € {0, 1, ..., 7}) its size. The memory size to be allocated to that data container is equal to **RX_FQ_SIZE{n}.DC_SIZE[11:0]** * 32byte.
- In case of Continuous mode, the **RX_FQ_RD_ADD_PT{n}** (n € {0, 1, ..., 7}) register must be initialized to {**RX_FQ_DC_START_ADD{n}.VAL[31:1]** & 0b11} otherwise left to its default value
- Unmask the interrupt *RX_FQ_IRQ[n]* (n € {0, 1, ..., 7}) on the interrupt controller
- Enable the RX FIFO Queue n writing 1 to the **RX_FQ_CTRL2.ENABLE[n]** (n € {0, 1, ..., 7}) bit register
- Start the RX FIFO Queue n writing 1 to the **RX_FQ_CTRL0.START[n]** bit register prior starting the PRT
- Wait for the **RX_FQ_STS0.BUSY[n]** (n € {0, 1, ..., 7}) bit status register to be set to 1. The RX FIFO Queue n is considered as active, meaning the **RX_FQ_STS0.BUSY[n]** bit register is set to 1 and running when the **RX_FQ_STS0.STOP[n]** bit register is set to 0

As soon as the **RX_FQ_STS0.BUSY[n]** = 1, the **RX_FQ_START_ADD{n}**, **RX_FQ_SIZE{n}**, **RX_FQ_RD_ADD_PT{n}** and **RX_FQ_DC_START_ADD{n}** (n € {0, 1, ..., 7}) configuration registers are write-protected for the RX FIFO Queue n.

1.4.7.4 Restarting a RX FIFO Queue

When the MH receives a RX message and there is no valid RX descriptor (VALID bit not set to 0) to write data to the S_MEM, the RX FIFO Queue n is put on hold (**RX_FQ_STS0.BUSY[n]** = 1 and **RX_FQ_STS0.STOP[n]** = 1). When such event occurs, the RX message is discarded and the RX_FQ_IRQ[n] interrupt is triggered to the system. Whatever the mode, Normal or Continuous, a restart is required to set the RX FIFO Queue n back to running. Only an active and running RX FIFO Queue can receive RX messages.

In order to restart a RX FIFO Queue n, do the following:

- A RX FIFO Queue can only restart if the **MH_CTRL.START** bit register is set to 1
- The SW must ensure **RX_FQ_STS0.BUSY[n], RX_FQ_STS0.STOP[n]** and **RX_FQ_CTRL2.ENABLE[n]** (n € {0, 1, ..., 7}) bit status registers are all set to 1. In this configuration, the **RX_FQ_START_ADD{n}**, **RX_FQ_SIZE{n}**, **RX_FQ_RD_ADD_PT{n}** and **RX_FQ_DC_START_ADD{n}** (n € {0, 1, ..., 7}) configuration registers are write-protected
- Start the RX FIFO Queue n writing 1 to the **RX_FQ_CTRL0.START[n]** bit register.
- Wait for the **RX_FQ_STS0.STOP[n]** bit register to be set to 0, to ensure the RX FIFO Queue n is considered as active and running

1.4.7.5 Aborting a RX FIFO Queue

Aborting a RX FIFO Queue n does make sense if it is active (**RX_FQ_STS0.BUSY[n]** = 1) otherwise nothing is done. This action can be taken at any time and will terminate with various delays depending on the MH states, see the bullet list below. Aborting a RX FIFO Queue does not affect the other ones currently running.
This kind of hard stop on a RX FIFO Queue would be mainly used for:

- Restarting properly a RX FIFO Queue when an error or issue has been detected while running
- To stop completely the MH, see Stopping MH Procedure chapter

In order to abort a RX FIFO Queue n running (**RX_FQ_STS0.BUSY[n]** = 1 and **RX_FQ_STS0.STOP[n]** = 0) or on hold (**RX_FQ_STS0.BUSY[n]** = 1 and **RX_FQ_STS0.STOP[n]** = 1), do the following:

- Write 1 to the **RX_FQ_CTRL1.ABORT[n]** (n € {0, 1, ..., 7}) bit register (the **RX_FQ_CTRL2.ENABLE[n]** bit register must be still set to 1)
- Wait for the **RX_FQ_STS0.BUSY[n]** and **RX_FQ_STS0.STOP[n]** (n € {0, 1, ..., 7}) bit status register to be set to 0. All status bit registers related to the RX FIFO Queue n are cleared, **RX_FQ_STS1.ERROR[n]** and **RX_FQ_STS1.UNVALID[n]** are set to 0. Once done the RX FIFO Queue n is considered as no more active
- Write 0 to the **RX_FQ_CTRL1.ABORT[n]** (n € {0, 1, ..., 7}) bit register
- Set the **RX_FQ_CTRL2.ENABLE[n]** (n € {0, 1, ..., 7}) bit register back to 0 to protect the RX FIFO Queue n from being restarted

As soon as the RX FIFO Queue is inactive, it is then possible to configure and change the setting of the RX FIFO Queue n, no write protection is active.

When aborting a RX FIFO Queue n, five cases need to be considered:

- The RX FIFO Queue n is active and running (**RX_FQ_STS0.BUSY[n]** = 1 and **RX_FQ_STS0.STOP[n]** = 0) and a new RX message is coming. The RX FIFO Queue number to receive the RX message data is not known when the abort is executed. If the RX message after filtering **is rejected**, the MH will wait for the reception of the timestamp before setting the **RX_FQ_STS0.BUSY[n]** bit register to 0. In case the RX filtering is too long, and has not finished before receiving the timestamp, the **RX_FQ_STS0.BUSY[n]** bit register is set to 0, only at the end of the RX filtering process.
- The RX FIFO Queue n is active and running (**RX_FQ_STS0.BUSY[n]** = 1 and **RX_FQ_STS0.STOP[n]** = 0) and a new RX message is coming. The RX FIFO Queue number to receive the RX message data is not known when the abort is executed. If the RX message after filtering **is accepted**, the MH will wait for the last data to be written in S_MEM and the writing of the acknowledge, before setting the **RX_FQ_STS0.BUSY[n]** bit register is set to 0. Even if the RX message accepted does not target the RX FIFO Queue to abort, the same principle applies.
- The RX FIFO Queue n is active and running (**RX_FQ_STS0.BUSY[n]** = 1 and **RX_FQ_STS0.STOP[n]** = 0) and no RX message data is received from the PRT (transmission in progress for instance). As there is no RX message received, the RX FIFO Queue n is aborted immediately and the **RX_FQ_STS0.BUSY[n]** bit register is set to 0.
- The RX FIFO Queue n is active and not running (**RX_FQ_STS0.BUSY[n]** = 1 and **RX_FQ_STS0.STOP[n]** = 1) when the abort is executed. The RX FIFO Queue n is aborted immediately and the **RX_FQ_STS0.BUSY[n]** and **RX_FQ_STS0.STOP[n]** bit registers are set to 0.
- The RX FIFO Queue n is inactive (**RX_FQ_STS0.BUSY[n]** = 0 and **RX_FQ_STS0.STOP[n]** = 0) when the abort is executed. Nothing is done.

As the MH will complete its current tasks before stopping the RX FIFO Queue, the *RX_ABORT_IRQ* interrupt will never be set.

1.4.7.6 TX FIFO Queue Initial Start

For the TX FIFO Queues, some common configuration registers need to be set prior any start. It is essential to note that those registers are write-protected when the MH is started (**MH_CFG.START** = 1):

- The **TX_DESC_MEM_ADD.FQ_BASE_ADDR[15:0]** bit field register defines the base address to store the TX descriptors for the TX FIFO Queues in the L_MEM
- The **MH_CFG.MAX_RETRANS[7:0]** bit field to define the number of possible re-transmissions for the same message
- The **MH_CFG.INST_NUM[2:0]** bit field to indicate the X_CAN instance number
- The **MH_SFTY_CFG** and **MH_SFTY_CTRL** registers according to the safety measures to apply
- The **AXI_ADD_EXT** register if the *DMA_AXI* address bus interface is greater than 32bit
- The **AXI_PARAMS** register to define read and write outstanding
- The **TX_FILTER_CTRL0/TX_FILTER_CTRL1** control registers and **TX_FILTER_REFVAL{n}** (n € {0, 1, 2, 3}) configuration registers, if the TX filter is enabled. All the registers assigned to this

feature must be configured before starting any TX FIFO Queues, see TX Filter chapter for more details

Before starting a TX FIFO Queue n several configuration registers need to be defined:

- The **TX_FQ_START_ADD{n}** register defines the address of the First TX Descriptor of the linked list
- The **TX_FQ_SIZE{n}.MAX_DESC** register provides the size of the linked list in number of TX descriptors
- The **TX_FQ_CTRL2.ENABLE[n]** bit register to enable the TX FIFO Queue n before a start

In order to define a TX FIFO Queue n, do the following:

- A TX FIFO Queue can only start if the **MH_CTRL.START** bit register is set to 1 (refer to the Initial MH Start Procedure section)
- The SW must check the **TX_FQ_STS0.BUSY[n]** and **TX_FQ_CTRL2.ENABLE[n]** (n € {0, 1, ..., 7}) bit registers are set to 0 (TX FIFO Queue n not already active and enabled). With such bit configuration the **TX_FQ_STS0.STOP[n]** bit register must be equal to 0
- Configure the start address of the TX descriptor linked list for the TX FIFO Queue n using the **TX_FQ_START_ADD{n}** (n € {0, 1, ..., 7}) register
- Define the maximum number of TX descriptors in the linked list, for the TX FIFO Queue n, using the **TX_FQ_SIZE{n}.MAX_DESC[9:0]** (n € {0, 1, ..., 7}) register. The memory size allocated is expected to be **TX_FQ_SIZE{n}.MAX_DESC[9:0]** * 32byte (TX descriptor size)
- In case no TX message is expected to be sent right away: The First TX Descriptor must be declared with a VALID bit set to 0. Doing so, the TX FIFO Queue will be put on hold right away after being started, waiting for a valid TX descriptor to send a TX message.
- In case some TX messages are expected to be sent right away: Define the relevant number of TX descriptors in the linked list with their respective data container. Only valid TX descriptors (VALID bit set to 1 in descriptor) can trig the transmission of TX messages. It is mandatory to declare an invalid TX descriptor (VALID bit set to 0) after the latest one being valid. This is required to put the RX FIFO Queue on hold when no more messages need to be sent.
- Unmask the interrupt *TX_FQ_IRQ[n]* (n € {0, 1, ..., 7}) on the interrupt controller
- Enable the TX FIFO Queue n writing 1 the **TX_FQ_CTRL2.ENABLE[n]** (n € {0, 1, ..., 7}) bit register
- Start the TX FIFO Queue n writing 1 to the **TX_FQ_CTRL0.START[n]** bit register. The PRT must be started prior this action
- Wait for the **TX_FQ_STS0.BUSY[n]** (n € {0, 1, ..., 7}) bit status register to be set to 1. Once done, the TX FIFO Queue n is considered as active
- The TX FIFO Queue n will be running (**TX_FQ_STS0.STOP[n]** = 0) up to the point an invalid TX descriptor is fetched from the S_MEM and then goes on hold (**TX_FQ_STS0.STOP[n]** = 1). The *TX_FQ_IRQ[n]* interrupt is triggered to the system to notify such state. In case the TX FIFO Queue is started with no TX messages, this interrupt is expected to happen in a very short time (roughly the time to fetch the TX descriptor from the S_MEM)

As soon as the **TX_FQ_STS0.BUSY[n] = 1,** the **TX_FQ_START_ADD{n}** and **TX_FQ_SIZE{n}** configuration registers are write-protected for the TX FIFO Queue n.

1.4.7.7 Restarting a TX FIFO Queue

When the MH has transmitted a TX message and there is no valid TX descriptor (VALID bit set to 0), the TX FIFO Queue n is put on hold (**TX_FQ_STS0.BUSY[n]** = 1 and **TX_FQ_STS0.STOP[n]** = 1). The *TX_FQ_IRQ[n]* interrupt is triggered to notify the system of such state. This is normal behavior, and such scenario can occur if the SW does not provide new TX messages in time before the MH gets to the last valid descriptor

In order to restart a TX FIFO Queue n, do the following:

- A TX FIFO Queue can only start if the **MH_CTRL.START** bit register is set to 1 and **MH_STS.ENABLE** = 1
- The SW must ensure **TX_FQ_STS0.BUSY[n], TX_FQ_STS0.STOP[n]** and **TX_FQ_CTRL2.ENABLE[n]** (n € {0, 1, ..., 7}) bit status registers are all set to 1. In this configuration, the **TX_FQ_START_ADD{n}**, **TX_FQ_SIZE{n}** and **TX_FQ_RD_ADD_PT{n}** (n € {0, 1, ..., 7}) configuration registers are write- protected
- If a new TX message needs to be sent, declare the TX descriptor in the linked list at the address defined in the **TX_FQ_ADD_PT{n}** register
- Start the TX FIFO Queue n writing 1 to the **TX_FQ_CTRL0.START[n]** bit register. Once started the **TX_FQ_STS0.STOP[n]** bit register goes to 0

Once done, the TX FIFO Queue n is considered as active, meaning the **TX_FQ_STS0.BUSY[n]** bit register is set to 1 and running if the **TX_FQ_STS0.STOP[n]** bit register is set to 0.

1.4.7.8 Aborting a TX FIFO Queue

Aborting a TX FIFO Queue n does make sense if it is active (**TX_FQ_STS0.BUSY[n]** = 1) otherwise nothing is done. This action can be taken at any time and will terminate with various delays depending on the MH states, see the bullet list below. Aborting a TX FIFO Queue does not affect the other ones currently running.
This kind of hard stop on a TX FIFO Queue would be mainly used for:

- Restarting properly a TX FIFO Queue when an error or issue has been detected while running
- To stop completely the MH, see Stopping MH Procedure chapter

In order to stop a TX FIFO Queue n running, do the following:

- Write 1 to the **TX_FQ_CTRL1.ABORT[n]** (n € {0, 1, ..., 7}) bit register
- Wait for the **TX_FQ_STS0.BUSY[n]** and **TX_FQ_STS0.STOP[n]** (n € {0, 1, ..., 7}) bit status register to be set to 0
- Write 0 to the **TX_FQ_CTRL1.ABORT[n]** (n € {0, 1, ..., 7}) bit register
- Set the **TX_FQ_CTRL2.ENABLE[n]** (n € {0, 1, ..., 7}) bit register back to 0 to protect the TX FIFO Queue n from being restarted

Once done the TX FIFO Queue n is considered as no more active, meaning the **TX_FQ_STS0.BUSY[n]** and **TX_FQ_STS0.STOP[n]** bit registers are both set to 0. It is then possible to configure and change the setting of the TX FIFO Queue n, no write protection is active. All status bit registers related to the TX FIFO Queue n are cleared, **TX_FQ_STS1.ERROR[n]** and **TX_FQ_STS1.UNVALID[n]** are set to 0.

When aborting a TX FIFO Queue n, five scenarios can occur:

- The TX FIFO Queue n is active and running (**TX_FQ_STS0.BUSY[n]** = 1 and **TX_FQ_STS0.STOP[n]** = 0) and one of its TX messages is selected as the highest priority message. As the message is not yet sent, nothing is preventing the MH to abort the TX FIFO Queue n. The **TX_FQ_STS0.BUSY[n]** bit register is set immediately to 0. The TX FIFO Queue n becomes inactive after a few cycles.
- The TX FIFO Queue n is active and running (**TX_FQ_STS0.BUSY[n]** = 1 and **TX_FQ_STS0.STOP[n]** = 0) and one of its TX messages is being transmitted to the PRT. The MH finishes the current TX message before aborting the TX FIFO Queue n. The TX FIFO Queue n becomes inactive when the current TX message of that TX FIFO Queue n is acknowledged. Some delays may occur before having the **TX_FQ_STS0.BUSY[n]** set to 0.
- The TX FIFO Queue n is active and running (**TX_FQ_STS0.BUSY[n]** = 1 and **TX_FQ_STS0.STOP[n]** = 0) and another TX FIFO Queue is sending a TX message to the PRT. As nothing is preventing the MH to abort the TX FIFO Queue n, the **TX_FQ_STS0.BUSY[n]** bit register is set immediately to 0. The TX FIFO Queue n becomes inactive after a few cycles.
- The TX FIFO Queue n is active and not running (**TX_FQ_STS0.BUSY[n]** = 1 and **TX_FQ_STS0.STOP[n]** = 1) when the abort is executed. As nothing is preventing the MH to abort the TX FIFO Queue n, the **TX_FQ_STS0.BUSY[n]** and **TX_FQ_STS0.STOP[n]** bit registers are set immediately to 0. The TX FIFO Queue n becomes inactive after a few cycles.
- The TX FIFO Queue n is inactive (**TX_FQ_STS0.BUSY[n]** = 0 and **TX_FQ_STS0.STOP[n]** = 0) when the abort is executed. Nothing is done.

As the MH will complete its current tasks before stopping the TX FIFO Queue, the *TX_ABORT_IRQ* interrupt will never be set.

1.4.7.9 TX Priority Queue Initialization

A TX Priority Queue is defined by up to 32 slots where a TX Header descriptor is defined per slot.
For the TX FIFO Priority Queue, some common configuration registers need to be set for all slots prior a start. It is essential to note that those registers are write-protected when the MH is started (**MH_CFG.START** = 1) :

- The **MH_CFG.MAX_RETRANS[7:0]** bit field to define the number of possible re-transmissions for the same message
- The **MH_CFG.INST_NUM[2:0]** bit field to indicate the X_CAN instance number
- The **MH_SFTY_CFG** and **MH_SFTY_CTRL** registers according to the safety measures to apply
- The **AXI_ADD_EXT** register if the *DMA_AXI* address bus interface is greater than 32bit
- The **AXI_PARAMS** register to define read and write outstanding

- The **TX_DESC_MEM_ADD.PQ_BASE_ADDR[15:0]** bit field register defines the base address to store the TX descriptors for the TX Priority Queue in the L_MEM
- The **TX_FILTER_CTRL0**/**TX_FILTER_CTRL1** control registers and **TX_FILTER_REFVAL{n}** (n € {0, 1, 2, 3}) configuration registers, if the TX filter is enabled. All the registers assigned to this feature must be configured before starting any TX FIFO Queues, see TX Filter chapter for more details

In order to define a TX Priority Queue, do the following:

- Check that **TX_PQ_STS0.BUSY[n]** is equal to 0 for n € {0, 1, ..., 31}, no TX Priority Queue slots must be active
- Define the TX Priority Queue start address in the **TX_PQ_START_ADD** register
- Define the size of the TX Priority Queue based on the number of expected slots to be active at the same time. Considering n slots, the expected memory size to be allocated for the TX Priority Queue is equal to n * 32byte (TX descriptor size)

As soon as one of the slots is started, the **TX_PQ_START_ADD** register is write-protected. Refer to the Starting a TX Priority Queue Slot section for more details.

1.4.7.10 Starting a TX Priority Queue Slot

Ensure the TX Priority Queue is initialized, refer to the TX Priority Queue Initialization section.
In order to start a TX Priority Queue slot n, do the following:

- A TX Priority Queue slot n can only start if the **MH_CTRL.START** bit register is set to 1 and **MH_STS.ENABLE** = 1 (refer to the Initial MH Start Procedure section)
- Define in S_MEM the TX descriptor for the TX Priority Queue slot n, at the address **TX_PQ_START_ADD[31:0]** + n * 32byte
- Unmask the interrupt *TX_PQ_IRQ* on the interrupt controller
- Enable the TX Priority Queue slot n in writing 1 to the **TX_PQ_CTRL2.ENABLE[n]** (n € {0, 1, ..., 31}) bit register
- Start the TX Priority Queue slot n in writing 1 to the **TX_PQ_CTRL0.START[n]** (n € {0, 1, ..., 31}) bit register. The PRT must be started prior this action.
- Wait for the **TX_PQ_STS0.BUSY[n]** (n € {0, 1, ..., 31}) bit status register to be set to 1. As soon as one TX Priority Queue slot is busy, the **TX_PQ_START_ADD** register is write protected.

Once the previous steps are completed, the TX Priority Queue slot n is considered as active, meaning the **TX_PQ_STS0.BUSY[n]** (n € {0, 1, ..., 31}) is set to 1.
As soon as the TX message defined in TX Priority Queue slot n is sent, the **TX_PQ_STS0.BUSY[n]** is set to 0 automatically. It is recommended to set the **TX_PQ_CTRL2.ENABLE[n]** bit register back to 0, once the transmission is completed, to avoid any start of non-initialized slots.
It is essential to note that when the MH stops, all the TX Priority Queue slots are set automatically inactive, meaning **TX_PQ_STS0.BUSY[31:0]** bit register are all set to 0.

1.4.7.11 Aborting a TX Priority Queue slot

In order to stop a TX Priority Queue slot n running, do the following:

- Write 1 to the **TX_PQ_CTRL1.ABORT[n]** (n € {0, 1, ..., 31}) bit register
- Wait for the **TX_PQ_STS0.BUSY[n]** (n € {0, 1, ..., 31}) bit status register to be set to 0
- Write 0 to the **TX_PQ_CTRL1.ABORT[n]** (n € {0, 1, ..., 31}) bit register
- Set the **TX_PQ_CTRL2.ENABLE[n]** (n € {0, 1, ..., 31}) back to 0 to protect the TX Priority Queue slot n from being restarted

Once done the TX Priority Queue slot n is set inactive, meaning the **TX_PQ_STS0.BUSY[n]** bit register is set to 0. It is possible to configure and change the global setting of the TX Priority Queue, only if there are no TX Priority Queue slot actives, meaning **TX_PQ_STS0.BUSY[31:0]** equal 0. All status bit registers related to the RX FIFO Queue n are cleared, **TX_PQ_STS1.SENT[n]** (n € {0, 1, ..., 31}) is set to 0.

When aborting a TX Priority Queue slot n, four scenarios can occur:

- The TX Priority Queue slot n is active (**TX_PQ_STS0.BUSY[n]** = 1) and its TX message is selected as the highest priority message. As the message is not yet sent, nothing is preventing the MH to abort the TX Priority Queue slot n. The **TX_PQ_STS0.BUSY[n]** bit register is set immediately to 0. The TX Priority Queue slot n becomes inactive after a few cycles.
- The TX Priority Queue slot n is active (**TX_PQ_STS0.BUSY[n]** = 1) and its TX message is being transmitted to the PRT. The MH finishes the TX message in progress. The TX Priority Queue slot n becomes inactive when the TX descriptor of the slot n is acknowledged. Some delays may occur before having the **TX_PQ_STS0.BUSY[n]** set to 0
- The TX Priority Queue slot n is active (**TX_PQ_STS0.BUSY[n]** = 1) and another TX Priority Queue slot is sending a TX message to the PRT. As nothing is preventing the MH to abort the TX Priority Queue slot n, the **TX_PQ_STS0.BUSY[n]** bit register is set immediately to 0
- The TX Priority Queue slot n is inactive (**TX_PQ_STS0.BUSY[n]** = 0) when the abort is executed. Nothing is done.

As the MH will complete its current tasks before stopping the TX Priority Queue slot, the *TX_ABORT_IRQ* interrupt will never be set.

1.4.7.12 RX Filter Setting

Prior doing any RX Filter configuration, the MH must be stopped (**MH_CTRL.START** = 0).
The RX Filter configuration does require the SW to:
1) Write the RX Filter Elements in the L_MEM, refer to the RX Filter chapter for RX Filter Element definition. How they are organized in the L_MEM is described in the Local Memory Map section (RX Filter Elements) in the Software Interface chapter.
2) Set the base address of the RX Filter elements in the **RX_FILTER_MEM_ADD.BASE_ADDR[15:0]** register. This base address value is specified in a 64KByte memory space area (L_MEM address bus width is 16bit only).
3) Set the **RX_FILTER_CTRL** register in write Privileged mode, see RX Filter chapter for more details.

Once the MH is started (**MH_CTRL.START** = 1), the **RX_FILTER_MEM_ADD** and **RX_FILTER_CTRL** are write-protected.

1.4.7.13 TX Filter Setting

Prior doing any TX Filter configuration, the MH must be stopped (**MH_CTRL.START** = 0).
The TX Filter configuration does require the SW to:
1) Set the **TX_FILTER_CTRL0** register (in write Privileged mode) to define the TX Filter global setting, see TX Filter chapter for more details.
2) Write the **TX_FILTER_CTRL1** register (in write Privileged mode) to enable one of the 16 TX Filters and to select the right bit field in the TX message header to compare with, see TX Filter chapter for more details.
3) Set the **TX_FILTER_REFVAL0**, **TX_FILTER_REFVAL1**, **TX_FILTER_REFVAL2** and **TX_FILTER_REFVAL3** registers (in write Privileged mode) to define value or value/mask pair to perform the comparison, see TX Filter chapter for more details.

Once the MH is started (**MH_CTRL.START** = 1), the **TX_FILTER_CTRL0**, **TX_FILTER_CTRL1**, **TX_FILTER_REFVAL0**, **TX_FILTER_REFVAL1**, **TX_FILTER_REFVAL2** and **TX_FILTER_REFVAL3** are write protected.

1.4.7.14 Timeout Setting

Three different timeouts can be set in the MH to protect:

- The AXI system bus interface connected to the MH *DMA_AXI* interface
- The AXI local memory interface connected to the *MEM_AXI* interface
- The internal bus (RX/TX)_MSG used to receive and transmit data from and to the PRT

A common prescaler is used to set the reference clock for all timeout counters, see **MH_SFTY_CFG.PRESCALER[1:0]** register. According to the value defined in the **MH_SFTY_CFG.PRESCALER[1:0]** register, the timeout counter reference clock is either CLK/32, CLK/64, CLK/128 or CLK/512.

In order to set the timeout value assigned to the 3 interfaces use the following registers:

- Set the value in the **MH_SFTY_CFG.DMA_TO_VAL[7:0]** register for the *DMA_AXI* interface
- Set the value in the **MH_SFTY_CFG.MEM_TO_VAL[7:0]** register for the *MEM_AXI* interface
- Set the value in the **MH_SFTY_CFG.PRT_TO_VAL[13:0]** register for the (RX/TX)_MSG interfaces

As the CLK and prescaler are common to all timeout counters, once the CLK is defined and the prescaler set, only a defined range is possible. The table below provides the different possible range for every timeout (according to the CLK clock frequency and clock ratio).

CLK (MHz)	PRESCALER	Timeout	DMA_AXI Timeout range (us)	MEM_AXI Timeout range (us)	(RX/TX)_MSG Timeout range (us)

	Value	CLK Ratio	Step (us)	Max 255	Min 0	Max 255	Min 0	Max 16383	Min 0
80	0	32	0.4	102	0	102	0	6553.2	0
	1	64	0.8	204	0	204	0	13106.4	0
	2	128	1.6	408	0	408	0	26212.8	0
	3	512	6.4	1632	0	1632	0	104851.2	0
160	0	32	0.2000	51.00	0	51.00	0	3276.60	0
	1	64	0.4000	102.00	0	102.00	0	6553.20	0
	2	128	0.8000	204.00	0	204.00	0	13106.40	0
	3	512	3.2000	816.00	0	816.00	0	52425.60	0
320	0	32	0.1000	25.50	0	25.50	0	1638.30	0
	1	64	0.2000	51.00	0	51.00	0	3276.60	0
	2	128	0.4000	102.00	0	102.00	0	6553.20	0
	3	512	1.6000	408.00	0	408.00	0	26212.80	0

1.4.7.14.1 DMA_AXI Interface Timeout Configuration

Any transfer from or to the system bus is protected using a timeout on the AXI read and write channel. As the timeout value is common to the AXI read and write channels, the setting must ensure read and write access time are covered.
A timeout counter is implemented per DMA channel, despite using the same timeout reference value.

The timeout value is defined with the **MH_SFTY_CFG.DMA_TO_VAL[7:0]** register, the **MH_SFTY_CFG.PRESCALER[1:0]** register and depends on the CLK clock frequency:
DMA AXI timeout value (us) = (1/(CLK(MHz))*(ratio defined in **MH_SFTY_CFG.PRESCALER[1:0]**)***MH_SFTY_CFG.DMA_TO_VAL[7:0]**

There are two different configurations of the MH leading to a different computation of the timeout value according to the number of outstanding transactions programmed in the **AXI_PARAMS** register:

- The **AXI_PARAMS.AR_MAX_PEND[1:0]** and **AXI_PARAMS.AW_MAX_PEND[1:0]** bit register are both equal to 1. The timeout counter starts when the AXI read or write command is accepted and stops when the last data response is received
- The **AXI_PARAMS.AR_MAX_PEND[1:0]** or **AXI_PARAMS.AW_MAX_PEND[1:0]** bit register is greater than 1. The timeout counter starts when the first AXI read or write command is accepted and stops when there are no more outstanding transactions

Here are the equations to fulfil:
Timeout value > **AXI_PARAMS.AR_MAX_PEND[1:0]***Prdl
Timeout value > **AXI_PARAMS.AW_MAX_PEND[1:0]***Pwdl

where Prdl = Peak of the read data system latency, Pwdl = Peak of the write data system latency and (**AXI_PARAMS.AR_MAX_PEND[1:0]** and **AXI_PARAMS.AW_MAX_PEND[1:0]** are always >= 1)

If the reads or writes do not complete in time, the **DMA_TO_ERR** interrupt is triggered to the system. The **SFTY_INT_STS.DMA_AXI_RX_TO_ERR** and **SFTY_INT_STS.MEM_AXI_TX_TO_ERR** bit register provide the information about the timeout source interrupt.

1.4.7.14.2 MEM_AXI Interface Timeout Configuration

Any transfer from or to the local memory L_MEM is protected using a timeout on the AXI read and write channel. As the timeout value is common to the AXI read and write channels, the setting must ensure read and write access time are covered.
A timeout counter is implemented per DMA channel, despite using the same timeout reference value.

The timeout value is defined with the **MH_SFTY_CFG.DMA_TO_VAL[7:0]** register, the **MH_SFTY_CFG.PRESCALER[1:0]** and depends on the CLK clock frequency:
MEM AXI timeout value (us) = (1/(CLK(MHz))*(ratio defined in **MH_SFTY_CFG.PRESCALER[1:0]**)***MH_SFTY_CFG.MEM_TO_VAL[7:0]**

The computation of the timeout value is different on the AXI read and write channels:

- On the AXI write channel, the timeout counter starts when the AXI write command is accepted and stops when the last data of that command is written
- On the AXI read channel, the timeout counter starts when the first AXI read or write command is accepted and stops when there are no more outstanding transactions

Here are the equations to fulfil:
Timeout value > Prdl*2
Timeout value > Pwdl
where Prdl = Peak of the read data system latency, Pwdl = Peak of the write data system latency
If the reads or writes do not complete in time, the **MEM_TO_ERR** interrupt is triggered to the system. The **SFTY_INT_STS.MEM_AXI_RX_TO_ERR** and **SFTY_INT_STS.MEM_AXI_TX_TO_ERR** bit register provide the information about the timeout source interrupt.

1.4.7.14.3 (RX/TX)_MSG Interface Timeout Configuration

The timeout counter, one per interface, starts counting when a start of frame is received from the PRT or transmitted to the PRT (SOF codeword identified). It ends when the last timestamp word is received from the PRT or transmitted to the PRT (TS1 codeword).
The timeout value, defined in the **MH_SFTY_CFG.PRT_TO_VAL[12:0]** register, is common to RX_MSG and TX_MSG interfaces. The SW must set the timeout value greater than the longest CAN frame to support in receive and transmit.

The timeout value is defined with the **MH_SFTY_CFG.PRT_TO_VAL[7:0]** register, the **MH_SFTY_CFG.PRESCALER[1:0]** and depends on the CLK clock frequency:
PRT timeout value (us) = (1/(CLK(MHz))*(ratio defined in **MH_SFTY_CFG.PRESCALER[1:0]**)***MH_SFTY_CFG.PRT_TO_VAL[7:0]**

If the received or transmitted CAN frame does not complete in time, the **DP_TO_ERR** interrupt is triggered to the system. The **SFTY_INT_STS.DP_PRT_RX_TO_ERR** and **SFTY_INT_STS.DP_PRT_TX_TO_ERR** bit register provide the information about the timeout source interrupt.

IMPORTANT: In case of data underrun, the PRT will complete its current frame, but the MH will provide right away the next TX message. Therefore, the TX_MSG timeout is restarted while the PRT is still transmitting the previous frame. If such DU codeword is received from the PRT, **the timeout value needs to cover 2 times the largest CAN frame** (the previous one to complete + the next one waiting to be transmitted).

LIMITATIONS: Due to the range of the RX/TX timeout value, which can be programmed in the **MH_SFTY_CFG.PRT_TO_VAL[7:0]** and **MH_SFTY_CFG.PRESCALER[1:0]** registers, the factor of 2 due to the data underrun, there is a limitation regarding the CAN XL protocol. Considering a very low bitrate, a maximum clock frequency of 320MHz, a maximum payload size of 2048bytes and a bit rate for the arbitration phase equal to 0.5Mbps: Timeout value can cover CAN XL bit rate greater or equal to 1.4Mbps.

A CAN XL frame with a maximum payload size of 2048bytes, and a bit rate of 1Mbps, would mean a message duration of 18.2ms (which is not acceptable for a real time system). In order to address a bit rate of 1Mbps in CAN XL, with the MH at 320MHz, the maximum payload size must be limited to 1460bytes.
At 160MHz, there is no issue regarding the setting of the RX/TX Timeout value, whatever the bitrate, the payload size of the CAN frame or the CAN protocols.

1.4.8 PRT and *ENABLE* Signal

The PRT signalizes via *ENABLE* whether it is active and requires message handling or not. It means a message can be received or transmitted only if the *ENABLE* signal is set high by the PRT.
As soon as this *ENABLE* signal goes low, the MH must stop its activity and goes in idle state. As the MH is stopped, the active TX FIFO Queue n and RX FIFO Queue m are put on hold, it means **TX_FQ_STS0.BUSY[n]** = 1, **RX_FQ_STS0.BUSY[m]** = 1, **TX_FQ_STS0.STOP[n]** = 1 and **RX_FQ_STS0.STOP[m]** = 1. Any active TX Priority Queue slot k is discarded, **TX_PQ_STS0.BUSY[k]** = 0. Any RX message received, or TX message transmitted at that time is discarded.

- Reinitialize MH: For more details, refer to the section Full Stop in Stopping MH Procedure chapter.

1.5 PRT – Protocol Controller

1.5.1 Overview

The PRT is a CAN XL Protocol Controller that can be integrated into different CAN modules. The PRT performs CAN communication as specified in ISO 11898-1:2015 (Classical CAN and CAN FD) and in CiA610-1 (CAN XL). The bitrate can be configured to values up to 20MBit/s at a clock speed of 160MHz, depending on the resources of the message handler and on the used semiconductor technology. For the connection to the physical layer, additional transceiver hardware is required, which is connected via GPIO ports or may be integrated into the CAN module (see chapter “Transceiver Interface”).

The PRT does not provide internal buffering of frames, so that data has to be transferred by IP internal Message Busses in 32 bit slices in real-time while (de)-serialization on the CAN Bus. Thus, single data transfers at the internal Message Busses are closely time-synchronized to the schedule at the CAN bus.

1.5.2 Features

- Classical CAN and CAN FD as specified in ISO 11898-1:2015
- CAN XL as specified in CiA610-1
- Classical CAN bit rate up to 1Mbps
- Arbitration phase bit rate up to 1Mbps for CAN FD and for CAN XL
- CAN FD data phase bit rate up to 8Mbps at a clock speed of 80 MHz or 160MHz
- CAN XL data phase bit rate up to 20Mbps at a clock speed of 160MHz

- Dedicated Timebase interface

1.5.3 Block Diagram

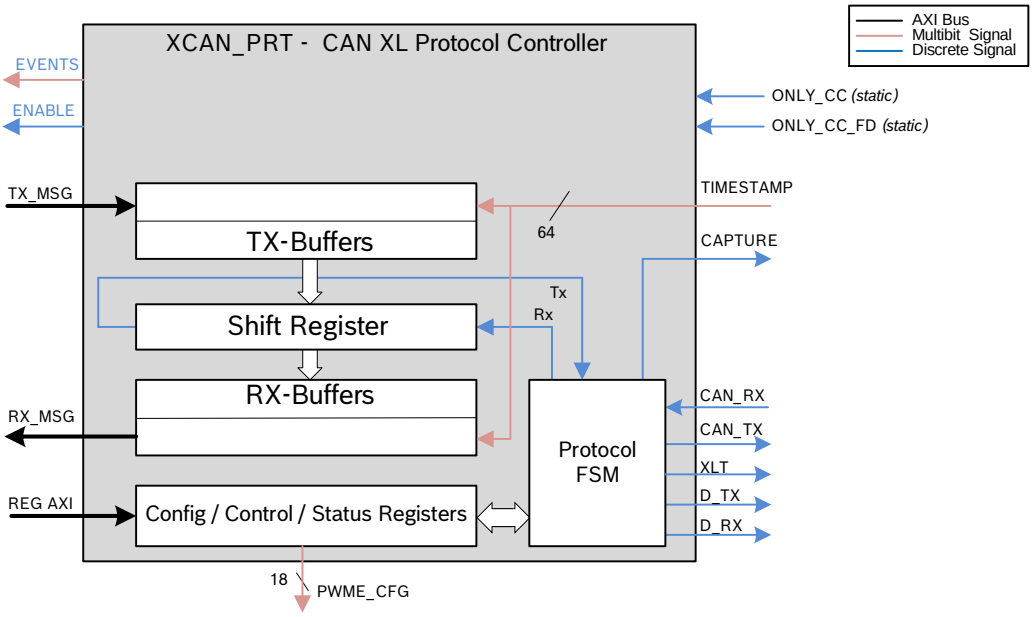

Figure: PRT Block Diagram

Figure above shows principle of the PRT's functions. When the PRT and the message handler operate in different clock domains, they are connected via CDC modules. The Time Base is captured inside a CDC module, triggered by the PRT's output signal *CAPTURE*.

The PRT consists of the CAN Protocol FSM, an Rx/Tx Shift Register, a set of interface registers for configuration, control, and status information as well as interfaces to the message handler for received messages and for messages to be transmitted. The PRT is not designed to store complete messages, there is only transient caching for two memory words for each direction during reception or transmission. A separate message handler is needed for the storage of whole messages as well as for functions like acceptance filtering, sorting of received messages into specific message buffers, and ordering the sequence of messages that are requested for transmission. Messages are streamed between message handler and PRT as sequences or 32 bit data words.

The host accesses the PRT's registers via REG_AXI, an AMBA AXI4-Lite interface, for configuration, control, and status information.

1.5.4 Software Interface

1.5.4.1 Register Map

The PRT's address map starts at the PRT's local address 0x00. All addresses are byte addresses, but the address map is organized as 32-bit words. The registers are 32 bit wide, while some register bits may be reserved. Reserved bits are always read as 0.

All registers are set to 0x00000000 after hardware reset except the two constant registers **ENDN** and **PREL**. The configuration registers are writable while the CAN communication is stopped, they are read-only while the CAN communication is started. The configuration registers are not changed by a software reset, see chapter “Software Reset”.

Address offset	Register name	Description	Access	Initial value
Status information of the PRT				
0x00	ENDN	Endianness Test Register	read-only	0x87654321
0x04	PREL	PRT Release Identification Register	read-only	0x05400000
0x08	STAT	PRT Status Register	read-only	0x00000010
Event information of the PRT				
0x20	EVNT	Event Status Flags Register	read-write	0x0000
Control of the PRT during runtime				
0x40	LOCK	Unlock Sequence Register	write-only	0x00000000
0x44	CTRL	Control Register	write-only	0x0000
0x48	FIMC	Fault Injection Module Control Register	read-write	0x0000
0x4C	TEST	Hardware Test functions register	read-write	0x00000008
Configuration of the PRT before runtime				
0x60	MODE	Operating Mode Register	read-write	0x000
0x64	NBTP	Arbitration Phase Nominal Bit Timing Register	read-write	0x00000000
0x68	DBTP	CAN FD Data Phase Bit Timing Register	read-write	0x00000000
0x6C	XBTP	XAN XL Data Phase Bit Timing Register	read-write	0x00000000
0x70	PCFG	PWME Configuration Register	read-write	0x000000

1.5.4.1.1 Register Access

The PRT registers are accessible in read/write mode through its AXI4-Lite slave interface *REG_AXI* (compliant to AMBA 4 ARM Ltd protocol, see [5]).

Any access to registers, either read or write, must use a 32bit aligned address otherwise a SLVERR is provided as a response.

When an access is performed to a non-mapped register in the address range, a SLVERR is provided as a response.
The phrase ‘SLVERR is provided as a response’ means that the REG_AXI responds with RRESP = ‘SLVERR’ respective BRESP = ‘SLVERR’. When attempting to write into any of the PRT's currently locked registers, the write does not occur, but no ‘SLVERR’ is triggered. A register Read Back is mandatory after writing to a potentially locked register, to confirm it has been correctly written.
The error is only reported on the AXI4-Lite protocol, no interrupt is triggered for such issue.

1.5.4.2 Register Description

1.5.4.2.1 status

REGISTER DESCRIPTION: Status information of the PRT
SIZE:
Register Base Address: 0x00
Register Address Range: 0x20

1.5.4.2.1.1 ENDN

Endianness Test Register

Address Offset:	0x00000000	Initial Value: 0x87654321
	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
Bit		ETV
Mode		R
Initial Value		0x876 54321

Bit 31:0 The purpose of this register is to identify the beginning of the PRT address map in a memory dump and to check the proper endianness data byte mapping when the data word is routed through different busses.

1.5.4.2.1.2 PREL

PRT Release Identification Register.

Address Offset:	0x00000004																Initial Value:								0x05400000							
	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
Bit	REL				STEP				SUBSTEP				YEAR				MON								DAY							
Mode	R				R				R				R				R								R							
Initial Value	0x0				0x5				0x4				0x0				0x0								0x0							

Bit 7:0 Define the day of the release using a binary coded decimal representation (1 being the first day of the month and so forth). This reset value is defined by the generic parameter DESIGN_TIME_STAMP_G[7:0]. If the generic parameter DESIGN_TIME_STAMP_G is not set, the default value is the one defined here

Bit 15:8 Define the month of the release using a binary coded decimal representation (1 being January and so forth). This reset value is defined by the generic parameter DESIGN_TIME_STAMP_G[15:8]. If the generic parameter DESIGN_TIME_STAMP_G is not set, the default value is the one defined here

Bit 19:16 Define the year of the release using a binary coded decimal representation (0 being 2020 and so forth...). This reset value is defined by the generic parameter DESIGN_TIME_STAMP_G[19:16]. If the generic parameter DESIGN_TIME_STAMP_G is not set, the default value is the one defined here

Bit 23:20 Sub-Step value according to Step

Bit 27:24 Step value according to Release

Bit 31:28 Release value, used to identify the main release of the XCAN_PRT.

1.5.4.2.1.3 STAT

PRT Status Register

Address Offset:	0x00000008																Initial Value:								0x00000010							
	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
Bit	TEC								RP	REC							TDCV								BO	EP	FIMA	CLKA	STP	INT	ACT	
Mode	R								R	R							R								R	R	R	R	R	R	R	
Initial Value	0x0								0x0	0x0							0x0								0x0	0x0	0x0	0x1	0x0	0x0	0x0	

Bit 1:0 The current activity of this node:
0b00: inactive state
0b01: Idle
0b10: Receiver
0b11: Transmitter

When the CAN protocol operation is stopped, ACT changes to 0b00 and INT changes to 0.
When the CAN protocol operation is started, INT is set to 1, but ACT remains at 0b00 until the CAN protocol's bus idle detection condition is met, then it changes to 0b01 and INT changes to 0.
When the CAN protocol operation is started while BO is set, the PRT remains in integrating state (INT=1 and ACT=0b00) until the Bus-Off recovery sequence is finished, then it changes to 0b01 and INT changes to 0.
When PRT detects a protocol exception event (see [1], chapter 10.9.5), ACT changes to 0b00 and INT changes to 1 until the CAN protocol's bus idle detection condition is met, then ACT changes to 0b01 and INT changes to 0.
ACT changes from 0b01 to 0b10 when the PRT has received a Start-of-Frame from the CAN bus.
ACT changes from 0b01 to 0b11 when the PRT has sent a Start-of-Frame to the CAN bus.
ACT changes from 0b11 to 0b10 when the PRT loses arbitration during a transmission.
ACT changes from 0b10 to 0b01 or from 0b11 to 0b01when the PRT detects the second bit of intermission (see [1], chapter 10.4.6.2) to be recessive.
Bit 2 This node is integrating into CAN bus traffic
Bit 3 Waiting for end of actual message after STOP command, see Starting and Stopping The Module chapter

Bit 4 The actual value of the CLOCK_ACTIVE input signal, see Starting and Stopping the Module chapter. As the clock must be active when a reset is performed, the default value should be 1.

Bit 5 Fault Injection Module Activated, see Safety Measures chapter

Bit 6 This node is in Error-Passive state. When both error counters drop below 127, or when the Bus-Off recovery sequence is finished, the EP bit is cleared.

Bit 7 This node is in Bus-Off state. This flag is set on an error condition that would have caused an increment of the Transmit Error Counter to a value beyond its 8 bit range. When the PRT enters Bus-Off state, BO is set to 1 and CAN protocol operation is stopped. When the Bus-Off recovery sequence is finished, BO is cleared.

Bit 15:8 Transmitter Delay Compensation's delay value. A software reset clears the TDV bit field to 0x00. This register shows the sum of the measured delay plus the configured offset, giving the position of the secondary sample point. It is updated for each frame transmission that includes a data phase.

Bit 22:16 The CAN protocol's Receive Error Counter. A software reset does not change the value in this register. When the Bus-Off recovery sequence is finished, the error counter REC is cleared. The REC is a 7-bit-counter, together with the Error-Passive flag EP. When the increment REC+1 or REC+8 would result in a value > 127 (carry-flag), the REC is kept unchanged, but EP is set. When EP is set but REC is below 127 and further errors are detected with an REC+1 condition, the REC will be incremented until it reaches 127. At the reception of a valid message, the REC-1 decrements the actual value of the REC by one AND clears the Error-Passive flag EP.

Bit 23 The Passive flag of the CAN protocol's Receive Error Counter. This flag is set on an error condition that would have caused an increment of the Receive Error Counter to a value beyond its 7 bit range.

Bit 31:24 The CAN protocol's Transmit Error Counter. A software reset does not change the value in this register. When the Bus-Off recovery sequence is finished, the error counter TEC is cleared. When the increment TEC+8 would result in a value > 255 (carry-flag), the TEC is kept unchanged, but BO is set. The Transmit Error Counter is decremented by one each time a CAN message has been successfully transmitted, but it is not decremented below the value 0.

1.5.4.2.2 event

REGISTER DESCRIPTION: Event information of the PRT

SIZE:

Register Base Address: 0x20

Register Address Range: 0x20

1.5.4.2.2.1 EVNT

Event Status Flags Register

The EVNT Register contains event status flags. The flags are set by the PRT when specific events occur. A software reset clears all flags. A host write access to this register, writing a 1 to a specific flag, clears that flag. When a host write access occurs concurrently with a set condition for a flag, the flag is set.

Address Offset:	0x00000000																Initial Value:		0x00000000													
	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
Bit																			ABO	IFR	USO	DU	PXE	TXF	RXF	DO	STE	FRE	AKE	B1E	B0E	CRE
Mode																			RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Initial Value																			0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

Bit 0 CRC Error

Bit 1 Bit0 Error: The PRT wanted to send a dominant bit (logical value 0), but the monitored CAN bus value was recessive. During Bus-Off recovery, B0E is also set each time a sequence of 11 recessive bits has been monitored, enabling the CPU to readily check whether the CAN bus is stuck at dominant or continuously disturbed, and to monitor the proceeding of the Bus-Off recovery sequence.

Bit 2 Bit1Error: During the transmission of a message (with the exception of the arbitration field), the PRT wanted to send a recessive bit (logical value 1), but the monitored CAN bus value was dominant.

Bit 3 Acknowledge Error

Bit 4 Form Error or the condition of CAN error counting rule f)

Bit 5 Stuff Error

Bit 6 Overflow condition in RX_MSG sequence detected

Bit 7 Frame received

Bit 8 Frame transmitted

Bit 9 Protocol Exception Event occurred

Bit 10 Underrun condition in TX_MSG sequence detected

Bit 11 Unexpected Start of Sequence during TX_MSG sequence detected

Bit 12 Invalid Frame Format requested in TX_MSG

Bit 13 TX_MSG sequence stopped by TX_MSG_WUSER code ABORT

1.5.4.2.3 control

REGISTER DESCRIPTION: Control of the PRT during runtime
SIZE:
Register Base Address: 0x40
Register Address Range: 0x20

1.5.4.2.3.1 LOCK

Unlock Sequence Register
Writing a sequence of specific data words enables the activation of control commands in the registers CTRL and FIMC. Reading this register always gives the value 0x00000000.

Address Offset:	0x00000000																Initial Value: 0x00000000															
	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
Bit	TMK																ULK															
Mode	w																w															
Initial Value	0x0																0x0															

Bit 15:0 Unlock Key
Bit 31:16 Test Mode Key

1.5.4.2.3.2 CTRL

Control Register
Writing to this register controls the CAN protocol operation. Reading this register gives the value 0x00000000.
When writing to this register, only one of the four bits TEST, SRES , STRT, or STOP may be written to 1, otherwise the write access takes no effect. The bit IMMD may be written to 1 together with the bit STOP, but not together with one of the other bits.

Address Offset:	0x00000004																Initial Value:			0x00000000											
	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
Bit																				TEST				SRES				STRT			IMMD	STOP
Mode																				W				W				W			W	W
Initial Value																				0x0				0x0				0x0			0x0	0x0

Bit 0 Stop CAN protocol operation. The Unlock Key must be used prior to write to this bit field. When not set together with bit IMMD the PRT waits for an ongoing CAN message to finish before stopping CAN protocol operation.

Bit 1 Stop CAN protocol operation immediately. The Unlock Key must be used prior to write to this bit . This bit is only effective when being set together with the bit STOP.

Bit 4 Start CAN protocol operation.

Bit 8 Software Reset. When the CAN protocol operation is stopped, the software reset of all state machines of the PRT (excluding the error-counters and the error-states) is triggered by writing 1 to CTRL.SRES. No unlocking sequence is required. A software reset will not be executed while the CAN protocol operation is started.

Bit 12 Enable Test Mode. The Test Mode Key must be used prior to write to this bit field.

1.5.4.2.3.3 FIMC

Fault Injection Module Control Register

Writing the fault injection position number requires the application of the test mode key sequence before writing to FIMC. This register must be accessed in privileged mode when supported.

Address Offset:	0x00000008																Initial Value:															0x00000000
	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
Bit																		FIP														
Mode																		RW														
Initial Value																		0x0														

Bit 14:0 Fault Injection Position. Writing to FIMC while MODE.FIME is set activates the Fault Injection Module FIM (see Safety Measures chapter). While the FIM is activated, the value of FIMC.FIP is protected from further write accesses until the FIM is de-activated again.

1.5.4.2.3.4 TEST

Hardware Test Functions Register

This register is writable after the hardware test mode functions are enabled by writing the test mode key sequence to LOCK and CTRL registers. While the hardware test mode functions are not enabled, this register is read-only. This register must be accessed in privileged mode when supported. The hardware test mode functions are disabled and cleared by the software reset of the PRT.

Address Offset:	0x0000000c																Initial Value:															0x00000008
	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
Bit					BUS_OFF	BUS_ON	E_PASSIVE	E_ACTIVE	BUS_ERR	RX_EVT	TX_EVT	IFF_RQ	RX_DO	TX_DU	USOS	ABORTED	HWT										TXC		RXD			LBCK
Mode					W	W	W	W	W	W	W	W	W	W	W	W	R										RW		R			RW
Initial Value					0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0										0x0		0x1			0x0

Bit 0 Enable the Message Loop-Back mode, see chapter Trace and Debug.

Bit 3 Bit value seen at CAN_RX. The CAN_RX input (output signal of the transceiver) is always readable through this bit.

Bit 5:4 Control the bit value driven at CAN_TX

0b00: Normal function of CAN TX

0b01: Normal function of CAN TX. CAN RX is ignored (for message look back mode)
0b10: CAN TX output set to 0
0b11: CAN TX output set to 1

Bit 15 This status flag HWT shows whether the hardware test mode functions are enabled, set to 1 means enable.

Bit 16 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared

Bit 17 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared

Bit 18 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared

Bit 19 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared

Bit 20 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared

Bit 21 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared

Bit 22 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared

Bit 23 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared

Bit 24 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared

Bit 25 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared

Bit 26 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared

Bit 27 Writing a 1 to the bit field triggers the related interrupt line, this bit is auto-cleared

1.5.4.2.4 configuration

REGISTER DESCRIPTION: Configuration of the PRT before runtime
SIZE:
Register Base Address: 0x60
Register Address Range: 0x20

1.5.4.2.4.1 MODE

Operating Mode Register
Configuration register that is writable while the CAN communication is stopped and that is read-only after the CAN communication is started. This register defines separate operating mode options. The

four configuration bits FDOE, XLOE, EFDI, and XLTR, are interrelated according to table Frame Formats defined in Operating Mode chapter.

Address Offset:	0x00000000																Initial Value:																0x00000000
	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
Bit																					FIME	EFDI	XLTR	SFS	RSTR	MON	TXP	EFBI	PXHD	TDCE	XLOE	FDOE
Mode																					RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Initial Value																					0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

Bit 0 FD Frame Format enabled. When set to 1, node is FD enabled according to ISO11898-1:2015. When set to 0, node is FD tolerant according to ISO11898-1:2015 (only Classical CAN frames used). This bit cannot be set to 1 when the static input ONLY_CC is set.

Bit 1 XL Frame Format enabled. When set to 0, node behaves according to ISO11898-1:2015, no arbitration during FDF bit. When set to 1, node behaves according to CiA610-1, arbitration during FDF bit and XLF bit. This bit cannot be set to 1 when one of the static inputs ONLY_CC or ONLY_CC_FD is set. Setting XLOE without setting FDOE is an invalid configuration.

Bit 2 Transmitter Delay Compensation Enabled as defined in [1]

Bit 3 Protocol Exception Handling Disabled

Bit 4 Edge Filtering during Bus Integration. If this bit is set, the PRT requires two consecutive dominant to to detect an edge causing the reset of the bit counter for the detection of the idle condition.

Bit 5 Transmit Pause. If this bit is set, the PRT pauses for two CAN bit times before starting the next transmission after itself has successfully transmitted a frame

Bit 6 Monitoring Mode Enabled as defined in [1]

Bit 7 Restricted Mode Enabled as defined in [1]

Bit 8 Time stamp position: Start of Frame Stamping

1: Timestamps captured at the start of a frame

0: Timestamps captured at the end of a frame

Bit 9 XL Transceiver Connected

Bit 10 Error Flag Disable, 1 means Error Signalling is disabled as defined in [2] and the error counters REC and TEC are not incremented. When this bit

is set, only CAN XL frames are transmitted and received dominant FDF or XLF bits are treated as form errors.

Bit 11 Fault Injection Module Enable, see Safety Measures chapter

1.5.4.2.4.2 NBTP

Arbitration Phase Nominal Bit Timing Register

Configuration register that is writable while the CAN communication is stopped and that is read-only after the CAN communication is started. This register defines the Nominal Bit Timing as defined in [1].

Address Offset:	0x00000004																Initial Value:								0x00000000							
	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
Bit			BRP					NTSEG1										NTSEG2								NSJW							
Mode			RW					RW										RW								RW							
Initial Value			0x0					0x0										0x0								0x0							

Bit 6:0 Nominal SJW. Valid values for the Nominal Synchronization Jump Width NSJW are 0x00-0x7F. The actual interpretation of this value is that the Nominal Synchronization Jump Width is (NSJW + 1) TQ long.

Bit 14:8 Nominal Phase_Seg2. Valid values for NTSEG2 are 0x01-0x7F. This value defines the length of Phase_Seg2(N). The actual interpretation of this value is that the phase buffer segment 2 is (NTSEG2 + 1) TQ long.

Bit 24:16 Nominal Prop_Seg and Phase_Seg1. Valid values for NTSEG1 are 0x01-0x1FF. This value defines the sum of Prop_Seg(N) and Phase_Seg1(N). The actual interpretation of this value is that these segments together are (NTSEG1 + 1) TQ long.

Bit 29:25 Bit Rate Prescaler. Valid values for the Bit Rate Prescaler BRP are 0x00-0x1F. This value defines the length of the Time Quantum TQ for all three bit time configurations. The actual interpretation of this value is that the TQ is (BRP + 1) CLK periods long

1.5.4.2.4.3 DBTP

CAN FD Data Phase Bit Timing Register

Configuration register that is writable while the CAN communication is stopped and that is read-only after the CAN communication is started. This register defines the FD Data Phase Bit Timing as defined in [1].

Address Offset:	0x00000008																Initial Value:								0x00000000							
	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
Bit	DTDCO								DTSEG1									DTSEG2								DSJW						
Mode	RW								RW									RW								RW						
Initial Value	0x0								0x0									0x0								0x0						

Bit 6:0 FD data phase SJW. Valid values for the FD data phase Synchronization Jump Width DSJW are 0x00-0x7F. The actual interpretation of this value is that the FD data phase Synchronization Jump Width is (DSJW + 1) TQ long.

Bit 14:8 FD data phase Phase_Seg2. Valid values for DTSEG2 are 0x01-0x7F. This value defines the length of Phase_Seg2(D). The actual interpretation of this value is that the phase buffer segment 2 is (DTSEG2 + 1) TQ long.

Bit 23:16 FD data phase Prop_Seg and Phase_Seg1. Valid values for DTSEG1 are 0x00-0xFF. This value defines the sum of Prop_Seg(D) and Phase_Seg1(D). The actual interpretation of this value is that these segments together are (DTSEG1 + 1) TQ long

Bit 31:24 Transmitter Delay Compensation Offset for FD frames. Valid values for the FD Transmitter Delay Compensation Offset DTDCO is 0x00-0xFF. This configuration defines the distance between the measured delay from CAN_TX to CAN_RX and the secondary sample point SSP, measured in CLK periods. This value is used when transmitting a CAN FD frame

1.5.4.2.4.4 XBTP

CAN XL Data Phase Bit Timing Register

Configuration register that is writable while the CAN communication is stopped and is read-only after the CAN communication is started. This register defines the XL Data Phase Bit Timing as defined in [2].

Address Offset:	0x0000000c																Initial Value:								0x00000000							
	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
Bit	XTDCO								XTSEG1									XTSEG2								XSJW						
Mode	RW								RW									RW								RW						
Initial Value	0x0								0x0									0x0								0x0						

Bit 6:0 XL data phase SJW. Valid values for the XL data phase Synchronization Jump Width XSJW are 0x00-0x7F. The actual interpretation of this value is that the XL data phase Synchronization Jump Width is (XSJW + 1) TQ long

Bit 14:8 XL data phase Phase_Seg2. Valid values for XTSEG2 are 0x01-0x7F. This value defines the length of Phase_Seg2(X). The actual interpretation of this value is that the phase buffer segment 2 is (XTSEG2 + 1) TQ long

Bit 23:16 XL data phase Prop_Seg and Phase_Seg1. Valid values for XTSEG1 are 0x00-0xFF. This value defines the sum of Prop_Seg(X) and Phase_Seg1(X). The actual interpretation of this value is that these segments together are (XTSEG1 + 1) TQ long

Bit 31:24 Transmitter Delay Compensation Offset for XL frames. Valid values for the XL Transmitter Delay Compensation Offset XTDCO is 0x00-0xFF. This configuration defines the distance between the measured delay from CAN_TX to CAN_RX and the secondary sample point SSP, measured in CLK periods. This value is used when transmitting a CAN XL frame.

1.5.4.2.4.5 PCFG

PWME Configuration Register

Configuration register that is writable while the CAN communication is stopped and is read-only after the CAN communication is started. This register defines the parameters needed for the PWM coding (as described in [2]) in the PWME module for CAN XL transceivers with switchable operating modes

Address Offset:	0x00000010																Initial Value:															
	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
Bit											PWMO								PWML								PWMS					
Mode											RW								RW								RW					
Initial Value											0x0								0x0								0x0					

Bit 5:0 PWM phase Short
Bit 13:8 PWM phase Long
Bit 21:16 PWM Offset

1.5.5 Functional Description

The PRT does provide several interrupt outputs that signal, with a high-pulse of one CLK period length, the occurrence of specific internal events. For test purposes, the TEST register has a Generate Interrupt Pulse function GIP so that in hardware test mode HWT an interrupt output pulse can also be triggered by writing a 1 to the corresponding TEST register bit.

Interrupt	TEST bit	*Activated when*
BUS_OFF	27	Entering Bus_Off State
BUS_ON	26	Starting CAN communication, after Starting or end of Bus_Off
E_PASSIVE	25	Switching from Error-Active to Error-Passive
E_ACTIVE	24	Switching from Error-Passive to Error-Active
BUS_ERR	23	Error detected on CAN bus or Protocol Exception Event detected
RX_EVT	22	Received a valid message
TX_EVT	21	Successfully transmitted a message
IFF_RQ	20	MH Requests a Message with Invalid Frame Format in Header
RX_DO	19	Data Overflow condition in RX_MSG sequence detected
TX_DU	18	Data Underrun condition in TX_MSG sequence detected
USOS	17	Unexpected Start of Sequence during TX_MSG sequence detected
ABORTED	16	TX_MSG sequence stopped by TX_MSG_WUSER code ABORT

The PRT outputs internal status information, optionally to be connected to a hardware debug port
SAMPLE_POINT: This is the CAN Sample Point
STAT_ACT: This is the actual 2-bit-value of register STAT.ACT (see register description)

1.5.5.1 PRT static configuration

The two inputs ONLY_CC and ONLY_CC_FD are intended to be connected to static signals (either hard-wired or OTP), thereby permanently restricting the PRT's function to older versions of the CAN protocol, see also [1].

The function of the PRT is restricted to the Classical CAN frame format (CAN FD tolerant implementation of ISO 11898-1:2015) when ONLY_CC = 1 or when the configuration bit MODE.FDOE is not set by the host.

The function of the PRT is restricted to the frame formats Classical CAN and CAN FD (full implementation of ISO 11898-1:2015) when ONLY_CC_FD = 1 or when the configuration bit MODE.XLOE is not set by the host.

With the exception of MODE.XLOE and MODE.FDOE, ONLY_CC and ONLY_CC_FD have no impact on the behavior of the other configuration registers.
They can change MODE.XLOE and MODE.FDOE to read-only.

- ONLY_CC = 1 means MODE.FDOE=0 and is not writable by Software
  MODE.XLOE=0 and is not writable by Software
- ONLY_CC_FD = 1 means MODE.XLOE=0 and is not writable by Software

1.5.5.2 Software Reset

The software reset is triggered by writing 1 to **CTRL.SRES** when the CAN protocol operation is stopped. This does not require an unlocking sequence. The software reset must not be executed when **CTRL.SRES** is written while the CAN protocol operation is started. The software reset resets all state machines of the PRT (excluding the error-counters and the error-states) and clears the following readable registers: **STAT.TDCV**, **STAT.FIMA**, **FIMC.FIP**, **TEST.HWT**, **TEST.TXC**, **TEST.LBCK**, and all flags of **EVNT**. The configuration registers are not changed by a software reset.

1.5.5.3 Operating Mode

The operating mode is defined using the **MODE** register and only when the CAN communication is stopped, otherwise the register is read only. The register **MODE** defines separate operating mode options.

The four configuration bits **FDOE**, **XLOE**, **EFDI**, and **XLTR**, and are interrelated according to the table below.
Table: Frame Formats

FDOE	XLOE	XLTR	EFDI	Description

0	0	X	0	Operating only in Classical CAN frame format. When **FDOE** is not set, the PRT shall be restricted to the Classical CAN frame format. In this case, when PXHD is set, the PRT shall accept both recessive and dominant bits as received reserved bits. When PXHD is not set, the PRT shall treat a recessive received first reserved bit as a Protocol Exception condition and shall enter the Protocol Exception State as defined in [1] and it shall set the flag EVNT.PXE. **FDOE** shall be static at 0 while the input signal **ONLY_CC** is 1.
0	1	X	X	Invalid configuration
0	X	X	1	Invalid configuration
X	0	X	1	Invalid configuration
1	0	X	0	Operating in Classical CAN and CAN FD frame format. When **FDOE** is set, the PRT shall be able to transmit and to receive Classical CAN frames and CAN FD frames as defined in [1]. When **XLOE** is not set, the PRT shall not be able to transmit or to receive CAN XL frames as defined in [2]. When **FDOE** is set but not **XLOE**, **PXHD** defines the PRT's reaction on a recessive reserved bit following the recessive FDF bit in a CAN FD frame. In this case, when **PXHD** is set, the PRT shall treat this condition as a Form Error. When **PXHD** is not set, the PRT shall treat this condition as a Protocol Exception condition and shall enter the Protocol Exception State as defined in [1] and it shall set the flag **EVNT**.**PXE**. **XLOE** shall be static at 0 while the input signals **ONLY_CC_FD** or **ONLY_CC** are at 1.
1	1	0	0	Operating in all frame formats, without XL transceiver. When **FDOE** and **XLOE** are both set and **EFDI** is not set, the PRT shall be able to transmit and to receive Classical CAN frames and CAN FD frames as defined in [1] and it shall be able to transmit and to receive CAN XL frames as defined in [2]. When both **FDOE** and **XLOE** are set, **PXHD** defines the PRT's reaction on a recessive reserved bit following the recessive XLF bit in a CAN XL frame. In this case, when **PXHD** is set, the PRT shall treat this condition as a Form Error. When **PXHD** is not set, the PRT shall treat this condition as a Protocol Exception condition and shall enter the Protocol Exception State as defined in [2] and it shall set the flag **EVNT**.**PXE**. When **EFDI** is not set, the PRT shall send error flags as defined in [1].
1	1	0	1	Operating in XL frame format only, without XL transceiver, error frames are disabled for all communication. It shall be an invalid configuration to set **EFDI** without setting both **FDOE** and **XLOE**. When **XLTR** is not set, the PRT shall not control the operating mode of the transceiver.
1	1	1	0	Invalid configuration
1	1	1	1	Operating in XL frame format only, enabling XL transceiver, error frames are disabled for all communication. When **XLTR** is set together with **FDOE** and **XLOE**, the PRT shall control the PWME to the transceiver in order to switch the operating mode of the transceiver at the beginning and at the end of the CAN XL data phase, as defined in [2]. When **EFDI** is set together with **FDOE** and **XLOE**, the PRT shall not send error flags and it shall not change its transmit error counter or its receive error counter. When an error condition occurs, the PRT shall, instead of sending an error flag, enter Protocol Exception State, like in Restricted Mode, as defined in [2].

1.5.5.4 Starting and Stopping the Module

The PRT is started by writing 1 to CTRL.STRT. This does not require an unlocking sequence. After the start command, the PRT waits for the occurrence of a sequence of 11 consecutive recessive bits (the idle condition of ISO 11898-1:1995) to finish its integration into the CAN communication on the CAN bus line. When the PRT has detected the idle condition and has no pending transmission request, it switches into idle state. When the PRT has detected the idle condition and has a pending transmission request, the PRT starts the transmission in the following bit. When the PRT sees a dominant bit on entering idle state, it immediately becomes receiver of that frame.

There are two options to stop the CAN protocol operation under software control, one that waits for the completion of an ongoing message transfer and one that stops the operation immediately.

The two options use different variants of the CTRL.STOP command. In the first variant, only the STOP bit is written to 1. In the second variant, both the CTRL.STOP bit and the CTRL.IMMD bit are written to 1 at the same time. Both variants require the application of the unlock key sequence, followed by a

write access to the CTRL register. The three consecutive write accesses may not be interrupted by other accesses via *REG_AXI*.

The first variant of the **STOP** command is asserted this way:

Step	Write data	to Register	at Address
1:	Write 0x1234	to **LOCK.ULK**	0x40
2:	Write 0x4321	to **LOCK.ULK**	0x40
3:	Write 1	to **CTRL.STOP**	0x44

The PRT's reaction to the first variant of the STOP command depends on its current activity (**STAT.ACT**). When the current activity of this node is Idle, it switches its activity to its inactive state immediately and stop all CAN operation. When the current activity is either Receiver or Transmitter, it continues that activity, and it sets the status flag **STAT.STP** to show that it is waiting for end of the actual message after a **CTRL.STOP** command. If no TX_MSG sequence is already started, the PRT clears *TX_MSG_WREADY* and keep it cleared until all CAN operation is stopped. As soon as the current reception or transmission is finished (either successfully or in failure) the PRT reports the result of that transfer to the MH, clears the status flag **STAT.STP**, clears *ENABLE*, switches its activity to its inactive state, and stops all CAN operation. The PRT does not start another reception or transmission until it is started again.

The second variant of the **STOP** command is asserted this way:

Step	Write data	to Register	at Address
1:	Write 0x1234	to **LOCK.ULK**	0x40
2:	Write 0x4321	to **LOCK.ULK**	0x40
3:	Write 1	to **CTRL.IMMD** and 1 to **CTRL.STOP**	0x44
4:	Write 1	to **CTRL.SRES**	0x44

The PRT's reaction to the second variant of the STOP command is to switch its activity to its inactive state immediately, to stop all CAN operation, and to set its *CAN_TX* output to 1 and to clear *ENABLE*. When the current activity was Transmitter, the PRT aborts that transmission. When the current activity was Receiver, it aborts that reception. Both interfaces with the MH are reset, outstanding transactions are discontinued. If this happens while an RX_MSG sequence was ongoing, this sequence is discontinued with the ABORT code. The PRT does not start another reception or transmission until it is started again.

The CAN protocol operation stops automatically on the following conditions:

- When the node enters the CAN protocol's Bus-Off state
- Unexpected Start of Sequence transaction during TX_MSG sequence detected

Note: Detecting an Unexpected Start of Sequence (USOS) indicates a mis-synchronization between PRT and MH that requires a restart.

When the CAN protocol operation is stopped, it is started by writing 1 to **CTRL.STRT**. This does not require an unlocking sequence.

When the CAN protocol operation was stopped because the PRT entered the CAN Bus_Off state, the start command (writing 1 to **CTRL.STRT**) causes the PRT to perform the CAN Bus_Off Recovery Sequence before it is again able to participate in CAN communication.

The CAN Bus_Off Recovery Sequence (see ISO 11898-1:2015) cannot be shortened by starting or stopping the PRT. If the PRT goes Bus_Off, it will set **STAT.BO** bit and it will, of its own accord, stop all bus activities. Once the PRT has been started again, the PRT clears **STAT.TEC**, **STAT.RP**, **STAT.REC**, and **STAT.EP** but it will keep **STAT.BO**.
The PRT will then wait for 129 occurrences of Bus Idle (129 * 11 consecutive recessive bits) before resuming normal operation. The PRT uses the Receive Error Counter (**STAT.REC**) to count the occurrences of Bus Idle. Additionally, each time a sequence of 11 recessive bits has been monitored, a Bit0 Error (**EVNT.B0E**) is reported, enabling the host to readily check-up whether the CAN bus is stuck at dominant or continuously disturbed and to monitor the progress of the Bus_Off recovery sequence. When the last-but-one sequence of 11 recessive bits has been monitored, **STAT.RP**, and **STAT.EP** are set and **STAT.REC** is at 0x7F. When the last sequence of 11 recessive bits has been monitored, the end of the Bus_Off recovery sequence is reached and **STAT.TEC**, **STAT.RP**, **STAT.REC**, **STAT.EP** and **STAT.BO** will all be reset. The PRT switches into idle state.

1.5.5.5 Reaction on Exceptions at the TX_MSG and RX_MSG Interfaces

1.5.5.5.1 MH Requests a Message with Invalid Frame Format in Header

This is detected when the requested transmit frame format is disabled in the configuration register (see **MODE.FDOE**, **MODE.EFDI**, or **MODE.XLOE**) or there is an internal contradiction in the header content of that frame. On the detection of this condition, the PRT ends the *TX_MSG* sequence with the response code HFI, generate a pulse on the *IFF_RQ* interrupt output and it does not transmit that message.

1.5.5.5.2 MH Intentionally Aborts TX_MSG Sequence

If the ABORT command is given with the second transaction of the *TX_MSG* sequence, the PRT does not start the transmission. If the ABORT command is given after the second transaction of the *TX_MSG* sequence, the PRT sets an internal flag that causes the FCRC bits of that transmission to be transmitted inverted. This internal flag is cleared at the end of the transmission. In both cases, the PRT generates a pulse on the *ABORTED* interrupt output and the ongoing *TX_MSG* sequence is finished.

1.5.5.5.3 Data Underrun Condition in TX_MSG Sequence Detected

This is detected when the ongoing transmission on the CAN bus needed another *TX_MSG* data word but that word was not provided in time. On the detection of this condition, the PRT continues the transmission (to avoid disturbing the message schedule on the CAN bus), but the PRT generates a

pulse on the *TX_DU* interrupt output and the PRT sets an internal flag that causes the FCRC bits of that transmission to be transmitted inverted (to avoid the acceptance of a message containing invalid data). This internal flag is cleared at the end of the transmission. In case of a Data Underrun, the ongoing *TX_MSG* sequence finishes with the code DU when the MH transfers the missing data word. After the end of the transmission, TX_MSG_WREADY is asserted again to accept the following TX_MSG Sequence (starting again with W0).

1.5.5.5.4 Unexpected Start of Sequence Detected

This is detected when the PRT receives a *TX_MSG* transaction marked as Start of Sequence before a previously started *TX_MSG* sequence has ended. This indicates that MH and PRT operate out-of-phase. On the detection of this condition, the PRT generates a pulse on the *USOS* interrupt output and the PRT stops CAN protocol operation. Afterwards, the PRT needs to be restarted under software control by writing 1 to **CTRL.STRT**.

1.5.5.5.5 Data Overflow Condition in RX_MSG Sequence Detected

This is detected when, during an ongoing reception, the MH has not acknowledged an *RX_MSG* data word in time. On the detection of this condition, the PRT generates a pulse on the *RX_DO* interrupt output and the PRT ends such an *RX_MSG* sequence via a subsequent transfer with code DO. This transfer with code DO must be acknowledged by the MH before the PRT can start a new *RX_MSG* sequence.

1.5.5.6 Controlling the Module's Clock Input

The PRT has two clock inputs, *CLK* and *CLK_AXI*. *CLK* is the clock input of the PRT excluding its REG_AXI interface, while *CLK_AXI* is the clock input of the PRT's REG_AXI module. Both clocks are synchronous to each other, driven from the same source. The difference between the two clocks is that *CLK_AXI* must be always active (to keep the REG_AXI interface operational), but *CLK* may be switched off (gated) while the PRT is stopped, e.g., when no CAN communication is needed.

The recommended clock frequency for CAN XL operation is 160 MHz.
The recommended clock frequency for CAN FD operation only is 80 MHz.
The function of the PRT does not depend on a particular duty-cycle of the clock, i.e., it reacts only on rising clock edges. The duration of the clock high pulse may vary between 10% and 90% of the clock period during operation.

The PRT's input signal *CLOCK_ACTIVE* shows whether the clock input *CLK* of the PRT is active. The actual value of the *CLOCK_ACTIVE* input signal is always readable from the status bit **STAT.CLKA**, even when *CLK* is not active. The signal *CLOCK_ACTIVE* does not control the PRT's function, it provides only status information, coming from a clock multiplexer outside of the PRT.

While *CLK* is not active, the PRT has no function and cannot be started. *CLK* must be reactivated before the PRT needs to be started again.

1.5.5.7 Transceiver Interface

The CAN bus is usually implemented as a twisted-pair bus line, its bus wires called *CAN_H* and *CAN_L*. An analog CAN transceiver device is connected to the CAN bus, interfacing between the bidirectional CAN bus wires and the CAN protocol controller's unidirectional, digital serial input and output signals. The future CAN XL transceivers will need to switch between two operating modes during the transmission of a CAN XL message. The switching control is coded into signals between protocol controller and transceiver. The coding is implemented inside a separate, dedicated PWME module that is placed between protocol controller and transceiver, see figure PRT Block Diagram.

The transceiver's *RxD* output is connected to the PRT's *CAN_RX* input. This asynchronous input signal is synchronized to the CAN clock by routing it through two synchronizer-FFs. This delay of two clock cycles is part of the input delay for the calculation of the propagation segment length of the CAN bit time. The CAN bit time configuration is only functional if the following conditions are fulfilled:
a) ((NTSEG1+1) x (BRP x CLK_period)) is larger than the transmitter loop delay
b) ((DTSEG1+1) x (BRP x CLK_period)) and ((XTSEG1+1) x (BRP x CLK_period)) are larger than the transmitter loop delay or transmitter delay compensation is enabled.

The connection from the PRT to the transceiver is routed through the PWME module (Pulse Width Modulation Encoder, specified in [2]). In CAN XL communication using a transceiver with switchable operating modes, the PWME controls the operating mode of the transceiver, to switch it into the CAN XL data phase modes for transmissions as well as receptions and back. The *PWME_CFG* configuration data consists of **PCFG.PWMO**, **PCFG.PWML**, and **PCFG.PWMS**, it is an 18-bit vector concatenating the three 6-bit vectors from **PCFG.PWMO**[5] down to **PCFG.PWMS**[0]. The appropriate switching times are signaled by the PRT via its outputs *XLT* (see [2], chapter 7.2.4), *D_TX* (see [2], chapter 7.2.5) and *D_RX* (see [2], chapter 7.2.6).

The PWME function is controlled by the following outputs of the PRT: *PWME_CFG*, *XLT*, *D_TX*, *D_RX*, and *CAN_TX*.

1.5.5.8 Hardware Timestamping

1.5.5.8.1 Timestamping Function

Timestamps are captured for each transmitted or received message, captured at either the sample point of the start of frame bit of the message or at the sample point of the bit when the message becomes valid at the end of the frame. The capture position is defined by the configuration bit **MODE.SFS**.

1.5.5.9 Trace and Debug

The hardware test mode functions are disabled by the software reset of the PRT. The status flag **TEST.HWT** shows whether the hardware test mode functions are enabled.

When the hardware test mode functions are disabled, all bits of **TEST** are cleared with the exception of **RXD**.
Enabling the hardware test mode functions (see chapter “Hardware Test Functions”) requires the application of the test mode key sequence. The test mode key sequence consists of three consecutive write accesses, not interrupted by other accesses via REG_AXI:

Step	Write data	to Register	at Address
1:	Write 0x6789	to **LOCK.TMK**	0x40
2:	Write 0x9876	to **LOCK.TMK**	0x40
3:	Write 1	to **CTRL.TEST**	0x44

The hardware test mode functions enable the host to directly control the values driven at the transceiver interface pins, to read the actual transceiver RxD output, and to transmit messages in a loop-back mode where all transmitted messages are also reported through the RX_MSG interface as received messages.

The *CAN_RX* input (output signal of the transceiver) is always readable at **RXD**
The *CAN_TX* output control **TXC** offers four options:
0b00: Normal function of *CAN_TX*
0b01: Normal function of *CAN_TX*, *CAN_RX* is ignored (for message loop-back mode)
0b10: *CAN_TX* output set to 0 and **XLT** output set to 0
0b11: *CAN_TX* output set to 1 and **XLT** output set to 0
When **LBCK** is set, the PRT operates in the message loop-back mode. In message loop-back mode, the PRT reports transmit messages (requested through the TX_MSG interface) via RX_MSG as received messages. The transmit messages are encoded and decoded bitwise inside the PRT, but a transmitted message is treated as successfully transmitted even if it does not get ACK. When the host sets **LBCK** to 1, it also sets **TXC** to either 0b01 or to 0b11 to control whether messages transmitted in the message loop-back mode are visible at the transceiver pins. In the message loop-back mode with **TXC** set to a value > 0b00, the actual *CAN_RX* input (output signal of the transceiver) is ignored by the PRT.

When the host sets **TXC**=0b11 in the message loop-back mode, the PRT keeps the *CAN_TX* output at 1 and loops back its internal serial output signal to its internal serial input signal. With this configuration, the loopback transmission does not disturb a CAN bus system connected to its transceiver.

When the host sets **TXC**=0b01 in the message loop-back mode, the PRT drives the frame bits at its *CAN_TX* output. In this case, the PRT loops back its internal serial output signal to its internal serial input signal, so it is not able to perform an arbitration or to react on bit errors on the CAN bus.

If TXC is set as zero b00, the loop-back test may be disturbed by errors on the CAN_RX input.

If TXC is set as 0b01, the loop-back test operates independently from the CAN_RX input, the Loopback transmission can be monitored at the CAN_TX output.
If TXC is set as 0b11, the loop-back test operates independently from the CAN_RX input, the Loopback transmission cannot be monitored at the Tx-pin. This is intended for a self-test in the field, not disturbing the CAN bus.

1.5.6 Application Information

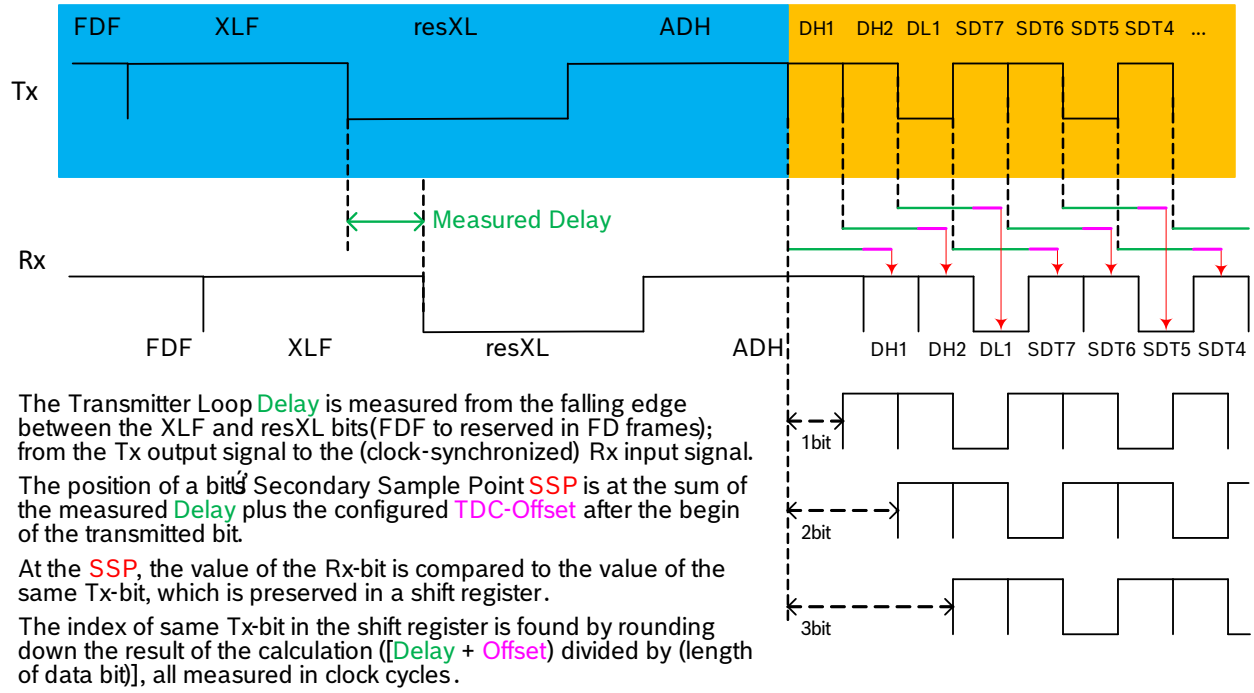

Figure: Overview of Transmitter Delay Compensation

The Transmitter Delay Compensation TDC is needed for applications where the bit rate in the CAN FD or CAN XL data phase is so high (and therefore the data bit time so short) that the transmitter loop delay prevents the node from a meaningful bit error check at the Sample-Point.
The transmitter loop delay is the time from the CAN_TX output at the start of the transmitted bit through (if used) the PWM Encoder, then through the transceiver to the CAN bus and back through the transceiver to the CAN_RX input and through the input synchronization.
Any two CAN nodes may have a systematic phase-shift to each other by the amount of the sum of the transmitter loop delay and the bus line delay between the two nodes.
The CAN arbitration mechanism and the acknowledge signaling function require that the time from the start of a bit at the synchronization segment to the bit's Sample-Point must be at least twice that systematic phase shift. This defines the required length of the bit's propagation segment. The two phase-buffer segments and the resynchronization mechanism remain to compensate for the oscillator frequency differences between the nodes.
In Classical CAN and in arbitration phase bit timing, the propagation segment is necessary and raises the minimum length of the bit time and therefore limits the achievable bit rate.

The CAN arbitration mechanism and the acknowledge signaling function are not used in the data phase of CAN FD or CAN XL, so in that phase no propagation segment is needed. When error signaling is enabled, the transmitter still needs to check for bit errors while operating in the data phase, this means that the time from the start of a bit at the synchronization segment to the bit's Sample-Point must be longer than the transmitter loop delay. For this calculation, the phase-buffer segment before the Sample-Point is included since here only the bits generated from the node's own local clock are regarded, no clock tolerances need to be considered.

When the transmitter loop delay is longer than the time from the start of the bit to the Sample-Point, the TDC mechanism needs to be enabled and the bit error check is then delayed to the time of the Secondary-Sample-Point, where the delayed input signal has arrived.

As shown in the TDC overview figure, the position of the Secondary-Sample-Point is not inside the transmitted bit, but inside one of the following Tx-bits. The TDC measures the delay in each transmitted FD or XL frame before it switches into the data phase. There may be small differences between successive measurement results due to changes in voltage, temperature, or input synchronization jitter. The measured delay (green line in the figure) shows the phase shift between the start of a transmitted bit at the Tx-output to the start of the same bit seen at the Rx-input of the PRT. The TDC offset (magenta line in the figure) needs to be configured to place the SSP at a position inside the same bit seen at the Rx-input. When the TDC detects a bit error at the SSP position, this information will be processed by the PRT at the following regular Sample-Point. The optimum position of the SSP inside that bit depends on the analyzed properties of the physical layer. The length of the recessive and the dominant bits may become asymmetric due to signal reflections and the different driving strengths. When transceivers are used that drive more symmetric signal shapes (e.g., CAN SIC or CAN SIC XL types), the position of the SSP should be in the middle of the received bit. The TDC offset configuration must always be below the length of a data-phase bit time.

Usual CAN SIC transceivers have a maximum Tx-input to Rx-output delay of 190ns. Together with a digital delay of three clock periods, this results in a maximum transmitter loop delay of 209ns (160 MHz clock) or 228ns (80 MHz clock). Other transceivers, especially those including galvanic isolation, have longer delay times and require TDC even at lower bit rates.

CAN SIC transceivers are specified for bit rates up to 8 MBit/s (125ns bit time). For higher bit rates, CAN SIC XL transceivers are needed that operate in a dedicated XL mode during the data phase. TDC is not needed for these higher bit rates because the CAN XL protocol specifies that error signaling, and bit error checking are disabled when the transceiver is switched into the XL mode. So 125ns is the shortest bit time to be considered for the TDC mechanism.

At the Secondary-Sample-Point, the TDC mechanism compares the actual value of the Rx-input signal with a stored value of the Tx-output signal. For this purpose, the TDC stores the last nine transmitted bits in a shift register. To select the correct shift register cell (or the current Tx-bit), to be compared to the Rx-input value at the SSP, the TDC divides the SSP position (STAT.TDCV, number of clock periods after the start of the bit) by the length of one bit time in the data phase (calculated number of clock periods from the configuration of NBTP.BRP and DBTP or XBTP). This limits the transmitter loop

delay that can be compensated to at most 10 bit times. With the shortest bit time of 125ns (8 MBit/s), this is 1250ns. At 5 MBit/s, the limit is 2000ns.

The second limitation for the maximum loop delay compensation comes from the calculation range. The TDC operates with a resolution of clock cycles in the range of 8 bit, so the latest SSP position is 255 clock cycles after the start of the Tx-bit, 1593ns (160 MHz clock) or 3187ns (80 MHz clock). The maximum distance between two SSPs is also 255 clock cycles, so the maximum length of a data phase bit may not exceed the number 255. When TDC is used, NBTP.BRP must be configured to a value of 0x00 or 0x01.
The length of a CAN FD data bit time is (NBTP.BRP * (DBTP.TSEG1 + DBTP.TSEG2 + 3)) clock cycles.
The length of a CAN XL data bit time is (NBTP.BRP * (XBTP.TSEG1 + XBTP.TSEG2 + 3)) clock cycles.

The CAN XL protocol specification requires that node shall be able to compensate transmitter delays of at least 95 clock cycles, which is 593,75ns (160 MHz clock) or 1187.5ns (80 MHz clock).

When the transmitter loop delay is indeed at 95 clock cycles (upper range as required by protocol specification), this results in an upper limit for the configuration range of the TDC offset (DBTP.DTDCO or XBTP.XTDCO) to the number 160 (255 - 95). In systems with shorter transmitter loop delays, the TDC offset may be configured to a higher value. The TDC mechanism can compensate longer transmitter loop delays than 95 clock cycles, as long as the sum of the measured delay and the configured TDC offset does not exceed the number 255.

1.6 PWME – Pulse Width Modulation Encoder

1.6.1 Overview

PWME is the Pulse Width Modulation Encoder build in the X_CAN.

1.6.2 Features

- PWM encoding as specified in [2]

1.6.3 Block Diagram

Figure “PWME overview” shows the PWME and its interfaces with the CAN XL Protocol Controller and the CAN transceiver.

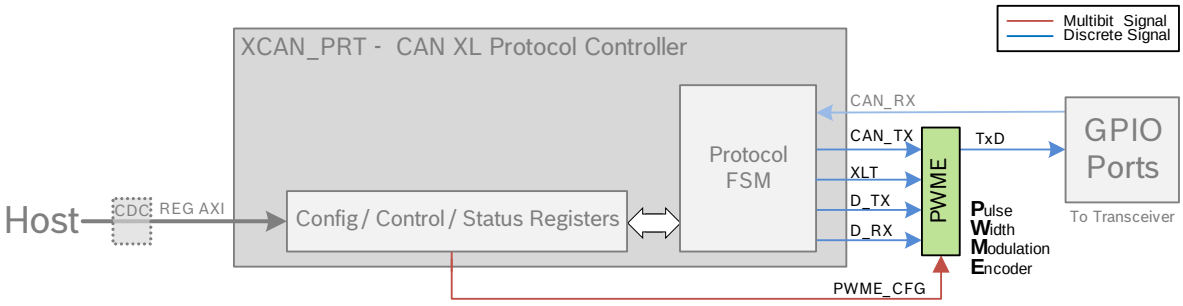

Figure: PWME overview

1.6.4 Software interface

1.6.4.1 PWME Configuration (PWME_CFG)

The *PWME_CFG* contains the parameters needed for the PWM encoding (as defined in [2]) in the PWME module. The *PWME_CFG* signal may not change its value while the PRT is started, i.e., while CAN frames can be received and transmitted.

Bits	Config	Description
17:12	**PWMO**[5:0]	PWM Offset
11:6	**PWML**[5:0]	PWM phase Long
5:0	**PWMS**[5:0]	PWM phase Short

Valid values for the PWM phase Short **PWMS** are 0x00-0x3F. The actual interpretation of this value is that the PWM short phase length is (**PWMS** + 1) clock cycles long.

Valid values for the PWM phase Long **PWML** are 0x00-0x3F. The actual interpretation of this value is that the PWM long phase length is (**PWML** + 1) clock cycles long.

The PWM symbol length is the sum of PWM short phase length and PWM long phase length (**PWMS** + **PWML** + 2) clock cycles.

Valid values for the PWM Offset **PWMO** are 0x00-0x3F. PWMO shall always be smaller than the PWM symbol length (**PWMO** < **PWMS** + **PWML** + 2).

1.6.5 Functional description

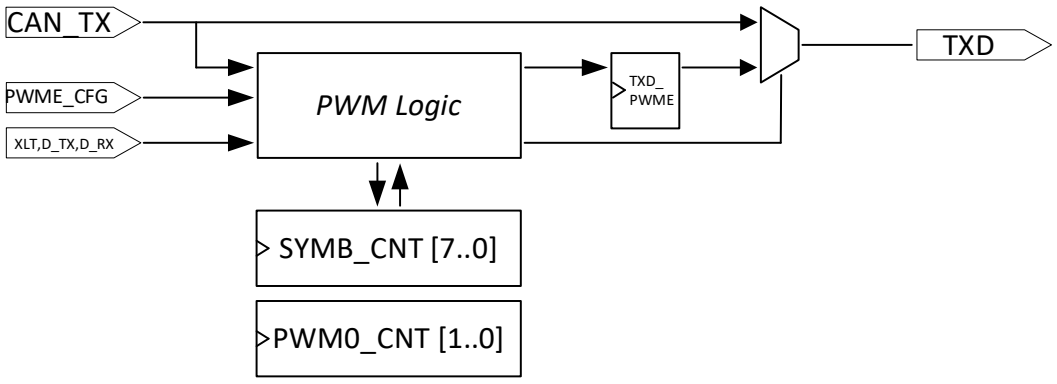

Figure: PWME Block Diagram
PWME implements the PWM encoding as specified in [2]. When transceiver mode switching is enabled, the PWME encodes the *CAN_TX* input signal during a CAN XL frame's data phase and during ADH bit, to generate the PWM encoded output signal *TXD*.

The output of the PWM Logic is registered by the Flip Flop TXD_PWME.
An active Reset sets this Flip Flop TXD_PWME to one.
All Flip Flops in the PWME change their value with the rising edge of *CLK*.

1.6.5.1 Transparent Mode

While *XLT* is passive or both *D_RX* and *D_TX* are passive, the PWME interface behaves transparent between *CAN_TX* input and *TXD* output.

This Mode is independent of Reset.

1.6.5.2 PWM encoded Mode

While *XLT* is active, the *TXD* output is PWM encoded for a transmitting node while *D_TX* is active and for the receiving node while *D_RX* active.

The PWM encoded *TXD* output has one CLK cycle delay (internal processing delay) relative to the bit boundaries on *CAN_TX* input.

1.6.5.2.1 Transmitting Node

When the PWME detects an edge from passive to active on *D_TX* and if *XLT* is active, then the PWME drives a LOW level on *TXD* for one PWM Offset time.

With expiration of the PWM Offset time the *TXD* output drives 2 consecutive PWM_0 symbols. From there onwards all following PWM symbols follow the *CAN_TX* input.

When *D_TX* is passive or *XLT* is passive the PWME switches back to transparent behavior regardless of the actual PWM phase.

1.6.5.2.2 Receiving Node

When the PWME detects an edge from passive to active on *D_RX* and if *XLT* is active, then the PWME drives consecutive PWM_1 symbols without a leading Offset time.

When *D_RX* is passive or *XLT* is passive the PWME switches back to transparent behavior regardless of the actual PWM phase.

1.7 IRC - Interrupt Controller

1.7.1 Overview

The X_CAN IP is equipped with a central interrupt controller (IRC). It captures all events of the MH and PRT and can be configured for each event individually to interrupt the HOST CPU.
The events are organized in two categories, i.e., Functional Events and Error Events. Functional Events can trigger the IRC output *FUNC_INT*. Error Events can trigger the IRC outputs *ERR_INT* and *SAFETY_INT*.

1.7.2 Software Interface

1.7.2.1 Register Map

Address offset	Register name	Description	Access	Initial value
MH and PRT capture event registers				
0x00	FUNC_RAW	Functional raw event status register	read-only	0x0
0x04	ERR_RAW	Error raw event status register	read-only	0x0
0x08	SAFETY_RAW	Safety raw event status register	read-only	0x0
IRC control register				
0x10	FUNC_CLR	Functional raw event clear register	write-only	0x0
0x14	ERR_CLR	Error raw event clear register	write-only	0x0
0x18	SAFETY_CLR	Safety raw event clear register	write-only	0x0
0x20	FUNC_ENA	Functional raw event enable register	read-write	0x0
0x24	ERR_ENA	Error raw event enable register	read-write	0x0
0x28	SAFETY_ENA	Safety raw event enable register	read-write	0x0
Hardware configuration of the IRC.				
0x30	CAPTURING_MODE	IRC configuration register	read-only	0x7

Auxiliary				
0x40	HDP	Hardware Debug Port control register	read-only	0x7

1.7.2.2 Register Description

1.7.2.2.1 Event

REGISTER DESCRIPTION: This address block provides registers to capture events of the MH and the PRT. The events are organized in three categories (one register for each category): Functional relevant, functional error relevant and safety relevant.
SIZE:
Register Base Address: 0x00
Register Address Range: 0x10

1.7.2.2.1.1 FUNC_RAW

Functional raw event status register. This register provides information about the occurrence of functional relevant events inside the MH and the PRT. A flag is set when the related event is detected, independent of FUNC_ENA. The flags remain set until the Host CPU clears them by writing a 1 to the corresponding bit position at register FUNC_CLR.

Address Offset:	0x00000000																Initial Value:																0x00000000
	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
Bit					PRT RX EVT	PRT TX EVT	PRT BUS ON	PRT E ACTIVE		MH STATS IRQ	MH RX ABORT IRQ	MH TX ABORT IRQ	MH TX FILTER IRQ	MH RX FILTER IRQ	MH STOP IRQ	MH TX PQ IRQ	MH RX FQ7 IRQ	MH RX FQ6 IRQ	MH RX FQ5 IRQ	MH RX FQ4 IRQ	MH RX FQ3 IRQ	MH RX FQ2 IRQ	MH RX FQ1 IRQ	MH RX FQ0 IRQ	MH TX FQ7 IRQ	MH TX FQ6 IRQ	MH TX FQ5 IRQ	MH TX FQ4 IRQ	MH TX FQ3 IRQ	MH TX FQ2 IRQ	MH TX FQ1 IRQ	MH TX FQ0 IRQ
Mode					R	R	R	R		R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R	R
Initial Value					0x0	0x0	0x0	0x0		0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

Bit 0 MH interrupt of the TX FIFO Queue 0. This interrupt is triggered when an invalid TX descriptor is fetched from this TX FIFO Queue, a TX message from that FIFO Queue is sent (if set in TX descriptor), or a TX message of that TX FIFO Queue is skipped, see description of TX_FQ_IRQ[7:0] in MH section.

Bit 1 MH interrupt of the TX FIFO Queue 1. Refer to the description of the MH_TX_FQ0_IRQ

Bit 2 MH interrupt of the TX FIFO Queue 2. Refer to the description of the MH_TX_FQ0_IRQ

Bit 3 MH interrupt of the TX FIFO Queue 3. Refer to the description of the MH_TX_FQ0_IRQ

Bit 4 MH interrupt of the TX FIFO Queue 4. Refer to the description of the MH_TX_FQ0_IRQ

Bit 5 MH interrupt of the TX FIFO Queue 5. Refer to the description of the MH_TX_FQ0_IRQ

Bit 6 MH interrupt of the TX FIFO Queue 6. Refer to the description of the MH_TX_FQ0_IRQ

Bit 7 MH interrupt of the TX FIFO Queue 7. Refer to the description of the MH_TX_FQ0_IRQ

Bit 8 MH interrupt of the RX FIFO Queue 0. This interrupt is triggered when an invalid RX descriptor is fetched from this RX FIFO Queue, or an RX message is received (if set in RX descriptor) in this RX FIFO Queue, see description of RX_FQ_IRQ[7:0] in MH section.

Bit 9 MH interrupt of the RX FIFO Queue 1. Refer to the description of the MH_RX_FQ0_IRQ

Bit 10 MH interrupt of the RX FIFO Queue 2. Refer to the description of the MH_RX_FQ0_IRQ

Bit 11 MH interrupt of the RX FIFO Queue 3. Refer to the description of the MH_RX_FQ0_IRQ

Bit 12 MH interrupt of the RX FIFO Queue 4. Refer to the description of the MH_RX_FQ0_IRQ

Bit 13 MH interrupt of the RX FIFO Queue 5. Refer to the description of the MH_RX_FQ0_IRQ

Bit 14 MH interrupt of the RX FIFO Queue 6. Refer to the description of the MH_RX_FQ0_IRQ

Bit 15 MH interrupt of the RX FIFO Queue 7. Refer to the description of the MH_RX_FQ0_IRQ

Bit 16 Interrupt of TX Priority Queue. Any TX message sent from the TX Priority Queue can be configured to trigger this interrupt. The SW would then need to look at the MH register TX_PQ_INT_STS to identify which slot has generated the interrupt and for which reason.

Bit 17 The interrupt is triggered when the PRT is stopped. The MH finishes its task and switches to idle mode.

Bit 18 In order to track RX filtering results, an interrupt can be triggered when the comparison between a RX message header and a defined filter is successful.

Bit 19 The interrupt is triggered when the TX filter is enabled, and a TX message is rejected.

Bit 20 This interrupt line is triggered when the MH needs to abort a TX message being sent to the PRT.

Bit 21 This interrupt line is triggered when the MH needs to abort a RX message being received from PRT.
Bit 22 One of the RX/TX counters have reached the threshold.
Bit 24 PRT switched from Error-Passive to Error-Active state.
Bit 25 PRT started CAN communication, after start or end of BusOff
Bit 26 PRT transmitted a valid CAN message
Bit 27 PRT received a valid CAN message

1.7.2.2.1.2 ERR_RAW

Error raw event status register. This register provides information about the occurrence of functional error relevant events inside the MH and the PRT. A flag is set when the related event is detected, independent of ERR_ENA. The flags remain set until the Host CPU clears them by writing a 1 to the corresponding bit position at register ERR_CLR.

Address Offset:	0x00000004																Initial Value:															0x00000000
	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
Bit				TOP_MUX_TO_ERR					PRT_BUS_OFF	PRT_E_PASSIVE	PRT_BUS_ERR	PRT_IFF_RQ	PRT_RX_DO	PRT_TX_DU	PRT_USOS	PRT_ABORTED			MH_MEM_TO_ERR	MH_WR_RESP_ERR	MH_RD_RESP_ERR	MH_DMA_CH_ERR	MH_DMA_TO_ERR	MH_DP_TO_ERR	MH_DP_DO_ERR	MH_DP_SEQ_ERR	MH_DP_PARITY_ERR	MH_AP_PARITY_ERR	MH_DESC_ERR	MH_REG_CRC_ERR	MH_MEM_SFTY_ERR	MH_RX_FILTER_ERR	
Mode				R					R	R	R	R	R	R	R	R			R	R	R	R	R	R	R	R	R	R	R	R	R	R
Initial Value				0x0					0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0			0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

Bit 0 MH RX filtering has not finished in time, i.e. current RX filtering has not been completed before next incoming RX message requires RX filtering.
Bit 1 MH detected error in L_MEM. This interrupt is triggered when either the MEM_SFTY_CE or MEM_SFTY_UE input signal is active. The Message Handler provides the information, which signal was active, see flags MH:SFTY_INT_STS.MEM_SFTY_CE and MH:SFTY_INT_STS.MEM_SFTY_UE.
Bit 2 MH detected CRC error at the register bank. See also description of REG_CRC_ERR in MH section.
Bit 3 CRC error detected on RX/TX descriptor or RX/TX descriptor not expected detected. A status flag can define if it is on TX or RX path, see SFTY_INT_STS register.
Bit 4 MH detected parity error at address pointers, used to manage the MH Queues (RX/TX FIFO Queues and TX Priority Queues). See also description of AP_PARITY_ERR in MH section.

Bit 5 MH detected parity error at RX message data (received from PRT and written to AXI system bus) respective parity error detected at TX message data (read from AXI system bus and transferred to PRT).
Associated information provided by MH register ERR_INT_STS, e.g. if RX message or TX message was affected.
Bit 6 MH detected an incorrect sequence at RX_MSG respective TX_MSG interfaces located between MH and PRT. Associated information provided by MH register ERR_INT_STS, e.g. if RX or TX interface was affected.
Bit 7 MH detected a data overflow at RX buffer, see description of DP_DO_ERR in MH section.
Bit 8 MH detected timeout at TX_MSG interface located between MH and PRT, see description of DP_TO_ERR in MH section.
Bit 9 MH detected timeout at DMA_AXI interface, see description of DMA_TO_ERR in MH section.
Bit 10 MH detected routing error, i.e. data received or sent are not properly routed to or from DMA channel interfaces, see description of DMA_CH_ERR in MH section.
Bit 11 MH detected a bus error caused by a read access to S_MEM respective L_MEM, see description of RESP_ERR in MH section.
Bit 12 MH detected a bus error caused by a write access to S_MEM respective L_MEM, see description of RESP_ERR in MH section.
Bit 13 MH detected timeout at local memory interface MEM_AXI, see description of MEM_TO_ERR in MH section.
Bit 16 PRT detected stop of TX_MSG sequence by TX_MSG_WUSER code ABORT.
Bit 17 PRT detected unexpected Start of Sequence during TX_MSG sequence.
Bit 18 PRT detected underrun condition at TX_MSG sequence.
Bit 19 PRT detected overflow condition at RX_MSG sequence.
Bit 20 PRT detected invalid Frame Format at TX_MSG.
Bit 21 PRT detected error on the CAN Bus.
Bit 22 PRT switched from Error-Active to Error-Passive state.
Bit 23 PRT entered Bus_Off state.
Bit 28 Timeout at top-level multiplexer for HOST_AXI detected.

1.7.2.2.1.3 SAFETY_RAW

Safety raw event status register. This register provides information about the occurrence of safety relevant events inside the MH and the PRT. A flag is set when the related event is detected, independent of SAFETY_ENA. The flags remain set until the Host CPU clears them by writing a 1 to the corresponding bit position at register SAFETY_CLR.

Address Offset:	0x00000008																Initial Value:															
	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
Bit				TOP_MUX_TO_ERR					PRT_BUS_OFF	PRT_E_PASSIVE	PRT_BUS_ERR	PRT_IFF_RQ	PRT_RX_DO	PRT_TX_DU	PRT_USOS	PRT_ABORTED			MH_MEM_TO_ERR	MH_WR_RESP_ERR	MH_RD_RESP_ERR	MH_DMA_CH_ERR	MH_DMA_TO_ERR	MH_DP_TO_ERR	MH_DP_DO_ERR	MH_DP_SEQ_ERR	MH_DP_PARITY_ERR	MH_AP_PARITY_ERR	MH_DESC_ERR	MH_REG_CRC_ERR	MH_MEM_SFTY_ERR	MH_RX_FILTER_ERR		
Mode				R					R	R	R	R	R	R	R	R			R	R	R	R	R	R	R	R	R	R	R	R	R	R
Initial Value				0x0					0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0			0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

Bit 0 MH RX filtering has not finished in time, i.e. current RX filtering has not been completed before next incoming RX message requires RX filtering.

Bit 1 MH detected error in L_MEM. This interrupt is triggered when either the MEM_SFTY_CE or MEM_SFTY_UE input signal is active. The Message Handler provides the information, which signal was active, see flags MH:SFTY_INT_STS.MEM_SFTY_CE and MH:SFTY_INT_STS.MEM_SFTY_UE.

Bit 2 MH detected CRC error at the register bank. See also description of REG_CRC_ERR in MH section.

Bit 3 CRC error detected on RX/TX descriptor or RX/TX descriptor not expected detected. A status flag can define if it is on TX or RX path, see SFTY_INT_STS register.

Bit 4 MH detected parity error at address pointers, used to manage the MH Queues (RX/TX FIFO Queues and TX Priority Queues). See also description of AP_PARITY_ERR in MH section.

Bit 5 MH detected parity error at RX message data (received from PRT and written to AXI system bus) respective parity error detected at TX message data (read from AXI system bus and transferred to PRT).

Associated information provided by MH register ERR_INT_STS, e.g. if RX message or TX message was affected.

Bit 6 MH detected an incorrect sequence at RX_MSG respective TX_MSG interfaces located between MH and PRT. Associated information provided by MH register ERR_INT_STS, e.g. if RX or TX interface was affected.

Bit 7 MH detected a data overflow at RX buffer, see description of DP_DO_ERR in MH section.

Bit 8 MH detected timeout at TX_MSG interface located between MH and PRT, see description of DP_TO_ERR in MH section.

Bit 9 MH detected timeout at DMA_AXI interface, see description of DMA_TO_ERR in MH section.

Bit 10 MH detected routing error, i.e. data received or sent are not properly routed to or from DMA channel interfaces, see description of DMA_CH_ERR in MH section.
Bit 11 MH detected a bus error caused by a read access to S_MEM respective L_MEM, see description of RESP_ERR in MH section.
Bit 12 MH detected a bus error caused by a write access to S_MEM respective L_MEM, see description of RESP_ERR in MH section.
Bit 13 MH detected timeout at local memory interface MEM_AXI, see description of MEM_TO_ERR in MH section.
Bit 16 PRT detected stop of TX_MSG sequence by TX_MSG_WUSER code ABORT.
Bit 17 PRT detected unexpected Start of Sequence during TX_MSG sequence.
Bit 18 PRT detected underrun condition at TX_MSG sequence.
Bit 19 PRT detected overflow condition at RX_MSG sequence.
Bit 20 PRT detected invalid Frame Format at TX_MSG.
Bit 21 PRT detected error on the CAN Bus.
Bit 22 PRT switched from Error-Active to Error-Passive state.
Bit 23 PRT entered Bus_Off state.
Bit 28 Timeout at top-level multiplexer for HOST_AXI detected.

1.7.2.2.2 Control

REGISTER DESCRIPTION: This address block provides the registers for controlling the IRC.
SIZE:
Register Base Address: 0x10
Register Address Range: 0x20

1.7.2.2.2.1 FUNC_CLR

Functional raw event clear register. Writing a 1 to a certain bit position clears the corresponding bit of register FUNC_RAW. Writing a '0' has no effect.

Address Offset:	0x00000000																Initial Value:															
	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
Bit					PRT_RX_EVT	PRT_TX_EVT	PRT_BUS_ON	PRT_E_ACTIVE		MH_STATS_IRQ	MH_RX_ABORT_IRQ	MH_TX_ABORT_IRQ	MH_TX_FILTER_IRQ	MH_RX_FILTER_IRQ	MH_STOP_IRQ	MH_TX_PQ_IRQ	MH_RX_FQ7_IRQ	MH_RX_FQ6_IRQ	MH_RX_FQ5_IRQ	MH_RX_FQ4_IRQ	MH_RX_FQ3_IRQ	MH_RX_FQ2_IRQ	MH_RX_FQ1_IRQ	MH_RX_FQ0_IRQ	MH_TX_FQ7_IRQ	MH_TX_FQ6_IRQ	MH_TX_FQ5_IRQ	MH_TX_FQ4_IRQ	MH_TX_FQ3_IRQ	MH_TX_FQ2_IRQ	MH_TX_FQ1_IRQ	MH_TX_FQ0_IRQ
Mode					W	W	W	W		W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W
Initial Value					0x0	0x0	0x0	0x0		0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

Bit 0 Clear bit of FUNC_RAW.MH_TX_FQ0_IRQ by writing 1
Bit 1 Clear bit of FUNC_RAW.MH_TX_FQ1_IRQ by writing 1
Bit 2 Clear bit of FUNC_RAW.MH_TX_FQ2_IRQ by writing 1
Bit 3 Clear bit of FUNC_RAW.MH_TX_FQ3_IRQ by writing 1
Bit 4 Clear bit of FUNC_RAW.MH_TX_FQ4_IRQ by writing 1
Bit 5 Clear bit of FUNC_RAW.MH_TX_FQ5_IRQ by writing 1
Bit 6 Clear bit of FUNC_RAW.MH_TX_FQ6_IRQ by writing 1
Bit 7 Clear bit of FUNC_RAW.MH_TX_FQ7_IRQ by writing 1
Bit 8 Clear bit of FUNC_RAW.MH_RX_FQ0_IRQ by writing 1
Bit 9 Clear bit of FUNC_RAW.MH_RX_FQ1_IRQ by writing 1
Bit 10 Clear bit of FUNC_RAW.MH_RX_FQ2_IRQ by writing 1
Bit 11 Clear bit of FUNC_RAW.MH_RX_FQ3_IRQ by writing 1
Bit 12 Clear bit of FUNC_RAW.MH_RX_FQ4_IRQ by writing 1
Bit 13 Clear bit of FUNC_RAW.MH_RX_FQ5_IRQ by writing 1
Bit 14 Clear bit of FUNC_RAW.MH_RX_FQ6_IRQ by writing 1
Bit 15 Clear bit of FUNC_RAW.MH_RX_FQ7_IRQ by writing 1
Bit 16 Clear bit FUNC_RAW.MH_TX_PQ_IRQ by writing 1
Bit 17 Clear bit FUNC_RAW.MH_STOP_IRQ by writing 1
Bit 18 Clear bit FUNC_RAW.MH_RX_FILTER_IRQ by writing 1
Bit 19 Clear bit FUNC_RAW.MH_TX_FILTER_IRQ by writing 1
Bit 20 Clear bit FUNC_RAW.MH_TX_ABORT_IRQ by writing 1
Bit 21 Clear bit FUNC_RAW.MH_RX_ABORT_IRQ by writing 1
Bit 22 Clear bit FUNC_RAW.MH_STATS_IRQ by writing 1
Bit 24 Clear bit FUNC_RAW.PRT_E_ACTIVE by writing 1
Bit 25 Clear bit FUNC_RAW.PRT_BUS_ON by writing 1
Bit 26 Clear bit FUNC_RAW.PRT_TX_EVT by writing 1
Bit 27 Clear bit FUNC_RAW.PRT_RX_EVT by writing 1

1.7.2.2.2.2 ERR_CLR

Error raw event clear register. Writing a 1 to a certain bit position clears the corresponding bit of register ERR_RAW. Writing a '0' has no effect.

Address Offset:	0x00000004																Initial Value:															
	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
Bit				TOP_MUX_TO_ERR					PRT_BUS_OFF	PRT_E_PASSIVE	PRT_BUS_ERR	PRT_IFF_RQ	PRT_RX_DO	PRT_TX_DU	PRT_USOS	PRT_ABORTED			MH_MEM_TO_ERR	MH_WR_RESP_ERR	MH_RD_RESP_ERR	MH_DMA_CH_ERR	MH_DMA_TO_ERR	MH_DP_TO_ERR	MH_DP_DO_ERR	MH_DP_SEQ_ERR	MH_DP_PARITY_ERR	MH_AP_PARITY_ERR	MH_DESC_ERR	MH_REG_CRC_ERR	MH_MEM_SFTY_ERR	MH_RX_FILTER_ERR
Mode				W					W	W	W	W	W	W	W	W			W	W	W	W	W	W	W	W	W	W	W	W	W	W
Initial Value				0x0					0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0			0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

Bit 0 Clear bit ERR_RAW.MH_RX_FILTER_ERR by writing 1
Bit 1 Clear bit ERR_RAW.MH_MEM_SFTY_ERR by writing 1
Bit 2 Clear bit ERR_RAW.MH_REG_CRC_ERR by writing 1
Bit 3 Clear bit ERR_RAW.MH_DESC_ERR by writing 1
Bit 4 Clear bit ERR_RAW.MH_AP_PARITY_ERR by writing 1
Bit 5 Clear bit ERR_RAW.MH_DP_PARITY_ERR by writing 1
Bit 6 Clear bit ERR_RAW.MH_DP_SEQ_ERR by writing 1
Bit 7 Clear bit ERR_RAW.MH_DP_DO_ERR by writing 1
Bit 8 Clear bit ERR_RAW.MH_DP_TO_ERR by writing 1
Bit 9 Clear bit ERR_RAW.MH_DMA_TO_ERR by writing 1
Bit 10 Clear bit ERR_RAW.MH_DMA_CH_ERR by writing 1
Bit 11 Clear bit ERR_RAW.MH_RD_RESP_ERR by writing 1
Bit 12 Clear bit ERR_RAW.MH_WR_RESP_ERR by writing 1
Bit 13 Clear bit ERR_RAW.MH_MEM_TO_ERR by writing 1
Bit 16 Clear bit ERR_RAW.PRT_ABORTED by writing 1
Bit 17 Clear bit ERR_RAW.PRT_USOS by writing 1
Bit 18 Clear bit ERR_RAW.PRT_TX_DU by writing 1
Bit 19 Clear bit ERR_RAW.PRT_RX_DO by writing 1
Bit 20 Clear bit ERR_RAW.PRT_IFF_RQ by writing 1
Bit 21 Clear bit ERR_RAW.PRT_BUS_ERR by writing 1
Bit 22 Clear bit ERR_RAW.PRT_E_PASSIVE by writing 1
Bit 23 Clear bit ERR_RAW.PRT_BUS_OFF by writing 1
Bit 28 Clear bit ERR_RAW.TOP_MUX_TO_ERR by writing 1

1.7.2.2.2.3 SAFETY_CLR

Safety raw event clear register. Writing a 1 to a certain bit position clears the corresponding bit of register SAFETY_RAW. Writing a '0' has no effect.

Address Offset:	0x00000008																Initial Value:															
	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
Bit				TOP_MUX_TO_ERR					PRT_BUS_OFF	PRT_E_PASSIVE	PRT_BUS_ERR	PRT_IFF_RQ	PRT_RX_DO	PRT_TX_DU	PRT_USOS	PRT_ABORTED			MH_MEM_TO_ERR	MH_WR_RESP_ERR	MH_RD_RESP_ERR	MH_DMA_CH_ERR	MH_DMA_TO_ERR	MH_DP_TO_ERR	MH_DP_DO_ERR	MH_DP_SEQ_ERR	MH_DP_PARITY_ERR	MH_AP_PARITY_ERR	MH_DESC_ERR	MH_REG_CRC_ERR	MH_MEM_SFTY_ERR	MH_RX_FILTER_ERR	
Mode				W					W	W	W	W	W	W	W	W			W	W	W	W	W	W	W	W	W	W	W	W	W	W
Initial Value				0x0					0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0			0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

Bit 0	Clear bit SAFETY_RAW.MH_RX_FILTER_ERR by writing 1
Bit 1	Clear bit SAFETY_RAW.MH_MEM_SFTY_ERR by writing 1
Bit 2	Clear bit SAFETY_RAW.MH_REG_CRC_ERR by writing 1
Bit 3	Clear bit SAFETY_RAW.MH_DESC_ERR by writing 1
Bit 4	Clear bit SAFETY_RAW.MH_AP_PARITY_ERR by writing 1
Bit 5	Clear bit SAFETY_RAW.MH_DP_PARITY_ERR by writing 1
Bit 6	Clear bit SAFETY_RAW.MH_DP_SEQ_ERR by writing 1
Bit 7	Clear bit SAFETY_RAW.MH_DP_DO_ERR by writing 1
Bit 8	Clear bit SAFETY_RAW.MH_DP_TO_ERR by writing 1
Bit 9	Clear bit SAFETY_RAW.MH_DMA_TO_ERR by writing 1
Bit 10	Clear bit SAFETY_RAW.MH_DMA_CH_ERR by writing 1
Bit 11	Clear bit SAFETY_RAW.MH_RD_RESP_ERR by writing 1
Bit 12	Clear bit SAFETY_RAW.MH_WR_RESP_ERR by writing 1
Bit 13	Clear bit SAFETY_RAW.MH_MEM_TO_ERR by writing 1
Bit 16	Clear bit SAFETY_RAW.PRT_ABORTED by writing 1
Bit 17	Clear bit SAFETY_RAW.PRT_USOS by writing 1
Bit 18	Clear bit SAFETY_RAW.PRT_TX_DU by writing 1
Bit 19	Clear bit SAFETY_RAW.PRT_RX_DO by writing 1
Bit 20	Clear bit SAFETY_RAW.PRT_IFF_RQ by writing 1
Bit 21	Clear bit SAFETY_RAW.PRT_BUS_ERR by writing 1
Bit 22	Clear bit SAFETY_RAW.PRT_E_PASSIVE by writing 1
Bit 23	Clear bit SAFETY_RAW.PRT_BUS_OFF by writing 1
Bit 28	Clear bit SAFETY_RAW.TOP_MUX_TO_ERR by writing 1

1.7.2.2.2.4 FUNC_ENA

Functional raw event enable register. Any bit in the FUNC_ENA register enables the corresponding bit in the FUNC_RAW to trigger the interrupt line FUNC_INT. The interrupt line gets active high, when at least one RAW/ENA pair is 1, e.g. FUNC_RAW.MH_TX_FQ_IRQ = FUNC_ENA.MH_TX_FQ_IRQ = 1

Address Offset:	0x00000010																Initial Value:	0x00000000													
	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
Bit					PRT_RX_EVT	PRT_TX_EVT	PRT_BUS_ON	PRT_E_ACTIVE		MH_STATS_IRQ	MH_RX_ABORT_IRQ	MH_TX_ABORT_IRQ	MH_TX_FILTER_IRQ	MH_RX_FILTER_IRQ	MH_STOP_IRQ	MH_TX_PQ_IRQ	MH_RX_FQ7_IRQ	MH_RX_FQ6_IRQ	MH_RX_FQ5_IRQ	MH_RX_FQ4_IRQ	MH_RX_FQ3_IRQ	MH_RX_FQ2_IRQ	MH_RX_FQ1_IRQ	MH_RX_FQ0_IRQ	MH_TX_FQ7_IRQ	MH_TX_FQ6_IRQ	MH_TX_FQ5_IRQ	MH_TX_FQ4_IRQ	MH_TX_FQ3_IRQ	MH_TX_FQ2_IRQ	MH_TX_FQ1_IRQ	MH_TX_FQ0_IRQ
Mode					RW	RW	RW	RW		RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Initial Value					0x0	0x0	0x0	0x0		0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

Bit 0	Enable FUNC_RAW.MH_TX_FQ0_IRQ to activate FUNC_INT
Bit 1	Enable FUNC_RAW.MH_TX_FQ1_IRQ to activate FUNC_INT
Bit 2	Enable FUNC_RAW.MH_TX_FQ2_IRQ to activate FUNC_INT
Bit 3	Enable FUNC_RAW.MH_TX_FQ3_IRQ to activate FUNC_INT
Bit 4	Enable FUNC_RAW.MH_TX_FQ4_IRQ to activate FUNC_INT
Bit 5	Enable FUNC_RAW.MH_TX_FQ5_IRQ to activate FUNC_INT
Bit 6	Enable FUNC_RAW.MH_TX_FQ6_IRQ to activate FUNC_INT
Bit 7	Enable FUNC_RAW.MH_TX_FQ7_IRQ to activate FUNC_INT
Bit 8	Enable FUNC_RAW.MH_RX_FQ0_IRQ to activate FUNC_INT
Bit 9	Enable FUNC_RAW.MH_RX_FQ1_IRQ to activate FUNC_INT
Bit 10	Enable FUNC_RAW.MH_RX_FQ2_IRQ to activate FUNC_INT
Bit 11	Enable FUNC_RAW.MH_RX_FQ3_IRQ to activate FUNC_INT
Bit 12	Enable FUNC_RAW.MH_RX_FQ4_IRQ to activate FUNC_INT
Bit 13	Enable FUNC_RAW.MH_RX_FQ5_IRQ to activate FUNC_INT
Bit 14	Enable FUNC_RAW.MH_RX_FQ6_IRQ to activate FUNC_INT
Bit 15	Enable FUNC_RAW.MH_RX_FQ7_IRQ to activate FUNC_INT
Bit 16	Enable FUNC_RAW.MH_TX_PQ_IRQ to activate FUNC_INT
Bit 17	Enable FUNC_RAW.MH_STOP_IRQ to activate FUNC_INT
Bit 18	Enable FUNC_RAW.MH_RX_FILTER_IRQ to activate FUNC_INT
Bit 19	Enable FUNC_RAW.MH_TX_FILTER_IRQ to activate FUNC_INT
Bit 20	Enable FUNC_RAW.MH_TX_ABORT_IRQ to activate FUNC_INT
Bit 21	Enable FUNC_RAW.MH_RX_ABORT_IRQ to activate FUNC_INT
Bit 22	Enable FUNC_RAW.MH_STATS_IRQ to activate FUNC_INT
Bit 24	Enable FUNC_RAW.PRT_E_ACTIVE to activate FUNC_INT

Bit 25 Enable FUNC_RAW.PRT_BUS_ON to activate FUNC_INT
Bit 26 Enable FUNC_RAW.PRT_TX_EVT to activate FUNC_INT
Bit 27 Enable FUNC_RAW.PRT_RX_EVT to activate FUNC_INT

1.7.2.2.2.5 ERR_ENA

Error raw event enable register. Any bit in the ERR_ENA register enables the corresponding bit in the ERR_RAW to trigger the interrupt line ERR_INT. The interrupt line gets active high, when at least one RAW/ENA pair is 1, e.g. ERR_RAW.MH_TX_FQ_IRQ = ERR_ENA.MH_TX_FQ_IRQ = 1

Address Offset:	0x00000014																Initial Value:															
	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
Bit				TOP_MUX_TO_ERR					PRT_BUS_OFF	PRT_E_PASSIVE	PRT_BUS_ERR	PRT_IFF_RQ	PRT_RX_DO	PRT_TX_DU	PRT_USOS	PRT_ABORTED			MH_MEM_TO_ERR	MH_WR_RESP_ERR	MH_RD_RESP_ERR	MH_DMA_CH_ERR	MH_DMA_TO_ERR	MH_DP_TO_ERR	MH_DP_DO_ERR	MH_DP_SEQ_ERR	MH_DP_PARITY_ERR	MH_AP_PARITY_ERR	MH_DESC_ERR	MH_REG_CRC_ERR	MH_MEM_SFTY_ERR	MH_RX_FILTER_ERR
Mode				RW					RW	RW	RW	RW	RW	RW	RW	RW			RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Initial Value				0x0					0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0			0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

Bit 0 Enable ERR_RAW.MH_RX_FILTER_ERR to activate ERR_INT
Bit 1 Enable ERR_RAW.MH_MEM_SFTY_ERR to activate ERR_INT
Bit 2 Enable ERR_RAW.MH_REG_CRC_ERR to activate ERR_INT
Bit 3 Enable ERR_RAW.MH_DESC_ERR to activate ERR_INT
Bit 4 Enable ERR_RAW.MH_AP_PARITY_ERR to activate ERR_INT
Bit 5 Enable ERR_RAW.MH_DP_PARITY_ERR to activate ERR_INT
Bit 6 Enable ERR_RAW.MH_DP_SEQ_ERR to activate ERR_INT
Bit 7 Enable ERR_RAW.MH_DP_DO_ERR to activate ERR_INT
Bit 8 Enable ERR_RAW.MH_DP_TO_ERR to activate ERR_INT
Bit 9 Enable ERR_RAW.MH_DMA_TO_ERR to activate ERR_INT
Bit 10 Enable ERR_RAW.MH_DMA_CH_ERR to activate ERR_INT
Bit 11 Enable ERR_RAW.MH_RD_RESP_ERR to activate ERR_INT
Bit 12 Enable ERR_RAW.MH_WR_RESP_ERR to activate ERR_INT
Bit 13 Enable ERR_RAW.MH_MEM_TO_ERR to activate ERR_INT
Bit 16 Enable ERR_RAW.PRT_ABORTED to activate ERR_INT
Bit 17 Enable ERR_RAW.PRT_USOS to activate ERR_INT
Bit 18 Enable ERR_RAW.PRT_TX_DU to activate ERR_INT
Bit 19 Enable ERR_RAW.PRT_RX_DO to activate ERR_INT
Bit 20 Enable ERR_RAW.PRT_IFF_RQ to activate ERR_INT
Bit 21 Enable ERR_RAW.PRT_BUS_ERR to activate ERR_INT

Bit 22 Enable ERR_RAW.PRT_E_PASSIVE to activate ERR_INT
Bit 23 Enable ERR_RAW.PRT_BUS_OFF to activate ERR_INT
Bit 28 Enable ERR_RAW.TOP_MUX_TO_ERR to activate ERR_INT

1.7.2.2.2.6 SAFETY_ENA

Safety raw event enable register. Any bit in the SAFETY_ENA register enables the corresponding bit in the SAFETY_RAW to trigger the interrupt line SAFETY_INT. The interrupt line gets active high, when at least one RAW/ENA pair is 1, e.g. SAFETY_RAW.MH_TX_FQ_IRQ = SAFETY_ENA.MH_TX_FQ_IRQ = 1

Address Offset:	0x00000018																Initial Value:															
	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
Bit				TOP_MUX_TO_ERR					PRT_BUS_OFF	PRT_E_PASSIVE	PRT_BUS_ERR	PRT_IFF_RQ	PRT_RX_DO	PRT_TX_DU	PRT_USOS	PRT_ABORTED			MH_MEM_TO_ERR	MH_WR_RESP_ERR	MH_RD_RESP_ERR	MH_DMA_CH_ERR	MH_DMA_TO_ERR	MH_DP_TO_ERR	MH_DP_DO_ERR	MH_DP_SEQ_ERR	MH_DP_PARITY_ERR	MH_AP_PARITY_ERR	MH_DESC_ERR	MH_REG_CRC_ERR	MH_MEM_SFTY_ERR	MH_RX_FILTER_ERR
Mode				RW					RW	RW	RW	RW	RW	RW	RW	RW			RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Initial Value				0x0					0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0			0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0

Bit 0 Enable SAFETY_RAW.MH_RX_FILTER_ERR to activate SAFETY_INT
Bit 1 Enable SAFETY_RAW.MH_MEM_SFTY_ERR to activate SAFETY_INT
Bit 2 Enable SAFETY_RAW.MH_REG_CRC_ERR to activate SAFETY_INT
Bit 3 Enable SAFETY_RAW.MH_DESC_ERR to activate SAFETY_INT
Bit 4 Enable SAFETY_RAW.MH_AP_PARITY_ERR to activate SAFETY_INT
Bit 5 Enable SAFETY_RAW.MH_DP_PARITY_ERR to activate SAFETY_INT
Bit 6 Enable SAFETY_RAW.MH_DP_SEQ_ERR to activate SAFETY_INT
Bit 7 Enable SAFETY_RAW.MH_DP_DO_ERR to activate SAFETY_INT
Bit 8 Enable SAFETY_RAW.MH_DP_TO_ERR to activate SAFETY_INT
Bit 9 Enable SAFETY_RAW.MH_DMA_TO_ERR to activate SAFETY_INT
Bit 10 Enable SAFETY_RAW.MH_DMA_CH_ERR to activate SAFETY_INT
Bit 11 Enable SAFETY_RAW.MH_RD_RESP_ERR to activate SAFETY_INT
Bit 12 Enable SAFETY_RAW.MH_WR_RESP_ERR to activate SAFETY_INT
Bit 13 Enable SAFETY_RAW.MH_MEM_TO_ERR to activate SAFETY_INT
Bit 16 Enable SAFETY_RAW.PRT_ABORTED to activate SAFETY_INT
Bit 17 Enable SAFETY_RAW.PRT_USOS to activate SAFETY_INT
Bit 18 Enable SAFETY_RAW.PRT_TX_DU to activate SAFETY_INT
Bit 19 Enable SAFETY_RAW.PRT_RX_DO to activate SAFETY_INT
Bit 20 Enable SAFETY_RAW.PRT_IFF_RQ to activate SAFETY_INT
Bit 21 Enable SAFETY_RAW.PRT_BUS_ERR to activate SAFETY_INT

Bit 22 Enable SAFETY_RAW.PRT_E_PASSIVE to activate SAFETY_INT
Bit 23 Enable SAFETY_RAW.PRT_BUS_OFF to activate SAFETY_INT
Bit 28 Enable SAFETY_RAW.TOP_MUX_TO_ERR to activate SAFETY_INT

1.7.2.2.3 Configuration

REGISTER DESCRIPTION: Hardware configuration of the IRC
SIZE:
Register Base Address: 0x30
Register Address Range: 0x10

1.7.2.2.3.1 CAPTURING_MODE

IRC configuration register. This register shows the hardware configuration of the IRC concerning the capturing mode of the event inputs. The IP internal events signals coming from the MH and the PRT require an 'edge sensitive' capturing. That is why the value of this register is 0x7 and cannot be changed.

Address Offset:	0x00000000																Initial Value:															
	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
Bit																														SAFETY	ERR	FUNC
Mode																														R	R	R
Initial Value																														0x1	0x1	0x1

Bit 0 Capturing mode of FUNC_RAW register.
0 = Level sensitive
1 = Edge sensitive
Bit 1 Capturing mode of ERR RAW register.
0 = Level sensitive
1 = Edge sensitive
Bit 2 Capturing mode of SAFETY RAW register.
0 = Level sensitive
1 = Edge sensitive

1.7.2.2.4 Auxiliary

REGISTER DESCRIPTION: Auxiliary registers
SIZE:

Register Base Address: 0x40
Register Address Range: 0xB0

1.7.2.2.4.1 HDP

Hardware Debug Port control register

Address Offset:	0x00000000																Initial Value:															0x00000000
	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
Bit																																HDP_SEL
Mode																																RW
Initial Value																																0x0

Bit 0 Select the driver of the Hardware Debug Port. See also chapter HDP.
0 = Message Handler
1 = Protocol Controller

1.7.3 Functional Description

The following table lists the categories and shows the related IRC registers together with the dedicated IRC outputs:

Category	IRC Registers	IRC Output
Functional Event	**FUNC_RAW, FUNC_CLR, FUNC_ENA**	*FUNC_INT*
Error Event	**ERR_RAW, ERR_CLR, ERR_ENA**	*ERR_INT*
	SAFETY_RAW, SAFETY_CLR, SAFETY_ENA	*SAFETY_INT*

For each category, three registers are implemented: xxx_RAW, xxx_CLR, and xxx_ENA. To capture the events, the xxx_RAW registers are used. These registers provide information about the occurrence of events inside the MH and the PRT. A flag is set when the related event occurred, independent of xxx_ENA. The flags remain set until the HOST CPU clears them by writing a 1 to the corresponding bit position at register xxx_CLR.

The xxx_ENA registers control on bit level, whether a certain bit in the xxx_RAW register can activate the interrupt line xxx_INT. The interrupt line xxx_INT gets active high, when at least one RAW/ENA pair is 1, e.g., *ERR_INT* gets active high, when **ERR_RAW.MH_RX_FILTER_ERR** = **ERR_ENA.MH_RX_FILTER_ERR** = 1.

Details provided by register definitions.

1.8 Clock Domains and Resets

1.8.1 Clock Domains

The X_CAN IP is split into three clock domains.

- HOST Clock Domain
- CAN Clock Domain
- TIMEBASE Clock Domain

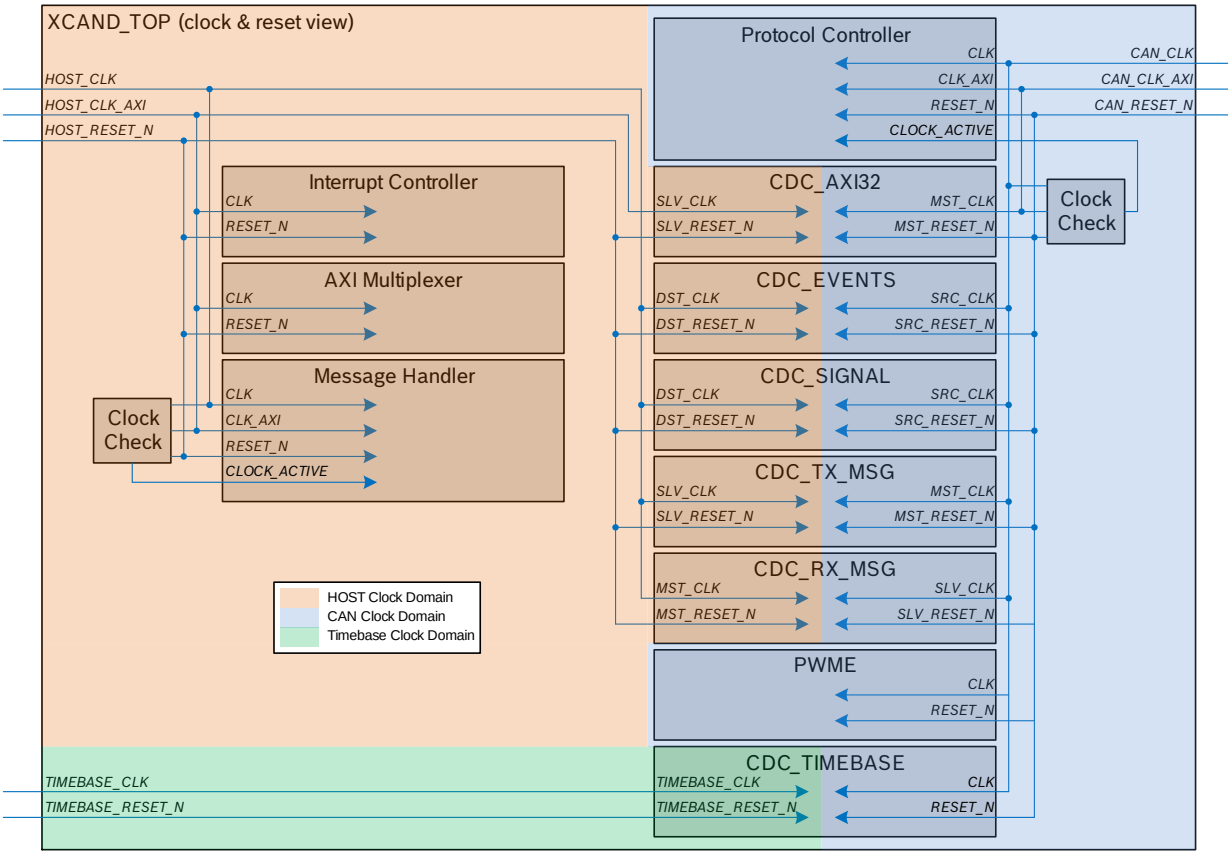

Figure: XCAND_TOP clock & reset

These are required clock frequencies for the clock domains HOST, CAN and TIMEBASE:

HOST: min. CAN frequency, typ. 160MHz, max. 320MHz

Customers may also use other frequencies within this range.

Note that the frequency of the MH clock impacts the performance of the message handling, e.g., message filtering.

CAN: 80MHz for Classical CAN and CAN FD operation only.

CAN: 160MHz for Classical CAN, CAN FD and CAN XL operation.

Other frequencies are not recommended to be used as this will lead to problems with respect to interoperability with other CAN devices.

TIMEBASE: min. 80MHz, typ. 160MHz, max. 320MHz
Customers may also use other frequencies within this range.
All clock domains can be supplied with clocks which are asynchronous to each other. The X_CAN IP internally handles all clock domain crossings, fully transparent for the user.

1.8.1.1 Behavior While Not Clocked

Here below is the X_CAN behavior when its clocks are off:
- Accessing the *HOST_AXI* when the HOST_CLK_AXI is inactive will result in a hang-up.
- Accessing the PRT through the *HOST_AXI* interface, when the CAN_CLK_AXI is inactive, will result in a timeout.

1.8.2 Resets

1.8.2.1 Behavior While Reset Active

Here below is the X_CAN behavior when its resets are active, or its clocks are off:
- Accessing the *HOST_AXI* while reset (HOST_RESET_N=0) will result in a hang-up.
- Accessing the PRT through the *HOST_AXI* interface while reset (CAN_RESET_N=0) will result in a timeout.

1.9 Application Information

1.9.1 Bit Rate and Performance

The bit rates (nominal bit rate, FD Data bit rate, XL Data bit rate) supported by the X_CAN IP depend on the system parameters: host clock frequency, CAN clock frequency, number of RX filter elements, and latency to the memories L_MEM and S_MEM.
An excel-sheet [6] is provided with the X_CAN IP to check if a specific combination of bit rates (nominal bit rate, FD Data bit rate, XL Data bit rate) and S_MEM/L_MEM latencies is functional under specific system parameters. Here below is an extract of a configuration set in the excel-sheet [6].

The X_CAN IP module is set to support:
- Nominal bit rate: 0.8 Mbit/s
- FD Data bit rate: 8 Mbit/s
- XL Data bit rate: 20 Mbit/s (CAN_CLK frequency: 160 MHz)
- RX Filter elements used: 128 (all use two comparisons of 32bit)
- L_MEM data read latency up to 6 clock cycles
- L_MEM shared by <=2 X_CAN instances
- SYS_MEM latency: <=5.5 us

The excel-sheet [6] is providing the following results

- HOST_CLK frequency: >=153 MHz
- Maximum system latency: <= 5.7 us

1.9.2 Time Stamping Offset

Due to internal propagation delays for the timestamping in the Protocol Controller (1 CAN clock cycle) and the clock domain crossing for the time capturing (n clock cycles, depending on the clock frequency ratio between the CAN and the TIMBASE domain) a certain offset has to be considered to derive the correct time.

These are the formulas for the offset:
TS_offset_max = 2 CAN_CLK period + 3 TIMEBASE_CLK period
TS_offset_min = 2 CAN_CLK period + 2 TIMEBASE_CLK period

This is the formula for the corrected timestamp:
Corrected Timestamp = Timestamp in TX/RX descriptor - TS_offset

The following table shows offsets for typical clock frequencies:
The first two columns contain the clock frequencies of the CAN_CLK and the TIMEBASE_CLK, measured in MHz. The third and fourth column contain the minimum and maximum TS_offset of the captured timestamp in multiple of TIMEBASE_CLK cycles, stored in the TX/RX descriptors. The last two columns show the interpretation of the TS_offset in nano seconds.

CAN_CLK [MHz]	TIMEBASE_CLK [MHz]	TS_offset_min [TIMEBASE_CLK cycles]	TS_offset_max [TIMEBASE_CLK cycles]	TS_offset_min [ns]	TS_offset_max [ns]
80	80	4	5	50,00	62,50
80	160	6	7	37,50	43,75
80	240	8	9	33,33	37,50
80	320	10	11	31,25	34,38
160	80	3	4	37,50	50,00
160	160	4	5	25,00	31,25
160	320	6	7	18,75	21,88

1.10 Detailed Design Information

1.10.1 Memory needs

The X_CAN needs a Local Memory (L_MEM) which is a synchronous RAM with 32 bit data, connected via *MEM_AXI* interface. The L_MEM stores RX filter elements, Header Descriptors for TX FIFO queues and TX Priority Queue.

For example, one X_CAN consumes 4 kB of memory space when using 128 filter elements (5x32 bit words per Filter element) and 256 reference pairs (value and mask) and 8 TX FIFO queues and 32 TX Priority queue slots.

The X_CAN provides a 16 bit address for the RAM to be able to address more than 4 kB. This additional address space is foreseen e.g., for using more filters and to support a Cluster Architecture where multiple X_CAN share one bigger L_MEM.

The data load at the L_MEM is typically very high and has to be considered for the X_CAN configuration. It is recommended to use [6] for cross-check.

1.11 Glossary

Term/Acronym	Meaning
ATPG	Automatic Test Pattern Generation/Generator
BCD	Binary Coded Decimal
CAN	Controller Area Network
CAN CC	CAN classic (a.k.a Classical CAN)
CAN FD	CAN flexible data rate (includes CC)
CAN XL	CAN extended data Length (includes CC and FD)
CDC	Clock Domain Crossing
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check used for data consistency check
DMA	Direct Memory Access
DST	Destination (data destination)
E_MEM	External Memory accessible off chip
EMI	ElectroMagnetic Interference
FF(s)	Flip-Flop(s)
FIXED	The data are read/written from/to a fixed address, useful for FIFO accesses. See [5] for more details
FM_PLL	Phase Lock Loop with Frequency Modulation to spread the noise spectrum
GPIO	General Purpose Input / Output
HDP	Hardware Debug Port
HOST	This is the CPU which is hosting the X_CAN
HW	Hardware
INCR	Successive data are read/written using an incremental address. See [5] for more details
IP	Intellectual property e.g., the X_CAN

IRC	Interrupt Controller
IRQ	Interrupt Request
ISO	International Standardization Organization
L_MEM	Local Memory accessible on chip
LSB	Least Significant Bit
MEM	Memory
MESSAGE	The information package to be transported on the CAN bus
MH	Message Handler
MSB	Most Significant Bit
MSG	Message, see MESSAGE
MUX	Multiplexer
NA	Not Applicable
OTP	One-time Programmable memory
OUTSTANDING	Refer to AXI protocol capability to issue/receive multiple transactions. - for an AXI slave: the number of outstanding transactions is the number of commands a slave is able to receive without blocking the response. - for an AXI master: the number of outstanding transactions is the number of commands a master is able to send without waiting for the response. Outstanding capability for read and write transactions are independent.
PARITY	Parity bit used for data consistency check
PLL	Phase Locked Loop
PRT	Protocol Controller
PWM	Pulse Width Modulation
PWME	Pulse Width Modulation Encoder
PWML	PWM long phase length
PWMO	PWM time offset parameter
PWMS	PWM short phase length
QUEUE	Buffer e.g., for Messages
RTL	Register Transfer Level (design abstraction level)
RX	Receive, e.g., reception of a frame on the CAN bus
S_MEM	System Memory. This memory could be an on-chip RAM or an E_MEM
SW	Software
SRAM	On chip RAM
SRC	Source (data source)
SW	Acronym to define CPU executing code
TX	Transmit, e.g., transmission of a frame on the CAN bus
TX SCAN	Process used to select the TX message having the highest priority before sending if to the PRT
VHDL	VHSIC Hardware Description Language
VHSIC	Very High-Speed Integrated Circuit
WORD	Data word used by X_CAN architecture = 32 bit = DWORD
WRAP	The data can be read/written using a wrap address strategy. See [5] for more details

X_CAN	Family of CAN IP supporting CAN XL protocol
XCAND	X_CAN with embedded DMA

1.12 References

This document refers to the following documents:

Ref Author(s)	Title
[1] ISO	ISO 11898-1:2015 CAN data link layer and physical signaling
[2] CAN in Automation	CiA 610-1 CAN XL Protocol Specification, V1.0.0, 2021-11-30
[3] Not Applicable	
[4] Not Applicable	
[5] AXI4	ARM IHI 0022E (ID022613)
[6] ME-IC/PAY	calc_min_host_clk_freq_and_max_sys_latency.xlsx Revision 1.12

1.13 Revision History

Version	Date	Description
0.1	02 Aug 2021	TOP Revision History: 0.0.4 MH Revision History: 0.0.59 PRT Revision History: 1.0.11 PWM Revision History: 1.0.3 IP Version: Beta (for internal use only)
1.1	08 Nov 2021	TOP Revision History: 0.0.9 MH Revision History: 0.1.17 PRT Revision History: 1.0.19 PWM Revision History: 1.0.5 IP Version: PreFreeze (for internal use only)
2.0	20 Dec 2021	TOP Revision History: 1.0.0 MH Revision History: 1.0.0 PRT Revision History: 1.0.29 PWM Revision History: 1.0.6 IP Version: FreezeRC (for internal use only)
2.3	02 Mar 2022	TOP Revision History: 1.0.6 MH Revision History: 1.0.11 PRT Revision History: 1.0.38 PWM Revision History: 1.0.6

Version	Date	Description
		IP Version: FreezeRC2 (for internal use only)
2.4	04 Apr 2022	TOP Revision History: 1.0.6 MH Revision History: 1.0.15 PRT Revision History: 1.0.40 PWM Revision History: 1.0.6 IP Version: FreezeRC3 (for internal use only)
2.8	05 May 2022	TOP Revision History: 1.0.8 MH Revision History: 1.0.18 PRT Revision History: 1.0.43 PWM Revision History: 1.0.6 IP Version: FreezeRC4 (for internal use only)
2.10	21 May 2022	TOP Revision History: 1.0.8 MH Revision History: 1.0.19 PRT Revision History: 1.0.43 PWM Revision History: 1.0.6 IP Version: FreezeRC5 (for internal use only)
3.2	19 Jul 2022	IP Version: R1.0.0
3.3	10 Aug 2022	IP Version: R1.0.1 • When a new RX frame is received, while processing the current RX frame (could be during RX filtering or afterwards), the new RX frame is discarded. In the previous release, the current RX frame is discarded, when a new RX frame starts while the RX filtering of the current RX frame is still in progress.
3.5	02 Nov 2022	IP Version R1.0.2 • The END bit was described and mentioned in the “TX descriptor Description” table but was missing in the “TX FIFO Queue Descriptor Overview” table in section TX Descriptor in MH chapter
3.8	28 Dec 2023	IP Version R1.1.0 • Add support of 20Mbps for CAN XL • Correct TX Priority Queue Descriptor Overview and TX FIFO Queue Descriptor Overview tables (PLSRC bit field was not at the right position and the size of the SIZE bit field was not correct, change from 8 to 9) • Minor correction on RX FIFO Queue schematics on MH section • Complete and add details on X_CAN feature list • Add details on RX Filter match computation in RX Filter section • Add details on Rolling Counter (RC[4:0]) in RX/TX descriptor • Add note on TX statistics counter in case of arbitration lost

Version	Date	Description
3.9	28 Feb 2024	IP Version R1.1.0 • Change organisation name in documents and references

1.14 Disclaimer

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.
DISCLAIMER: IN NO EVENT, UNLESS REQUIRED BY LAW OR AGREED TO IN WRITING, SHALL THE INTELLECTUAL PROPERTY OWNERS, COPYRIGHT HOLDERS OR ANY PERSON BE LIABLE FOR ANY LOSS, EXPENSE OR DAMAGE, OF ANY TYPE OR NATURE ARISING OUT OF THE USE OF, OR INABILITY TO USE THIS SPECIFICATION, SOFTWARE RELATED THERETO, CODE AND/OR PROGRAM RELATED THERETO, INCLUDING, BUT NOT LIMITED TO, CLAIMS, SUITS OR CAUSES OF ACTION INVOLVING ALLEGED INFRINGEMENT OF COPYRIGHTS, PATENTS, TRADEMARKS, TRADE SECRETS, OR UNFAIR COMPETITION.
INDEMNIFICATION: TO THE MAXIMUM EXTEND PERMITTED BY LAW YOU AGREE TO INDEMNIFY AND HOLD HARMLESS THE INTELLECTUAL PROPERTY OWNERS, COPYRIGHT HOLDERS AND CONTRIBUTORS, AND EMPLOYEES, AND ANY PERSON FROM AND AGAINST ALL CLAIMS, LIABILITIES, LOSSES, CAUSES OF ACTION, DAMAGES, JUDGMENTS, AND EXPENSES, INCLUDING THE REASONABLE COST OF ATTORNEYS' FEES AND COURT COSTS, FOR INJURIES OR DAMAGES TO THE PERSON OR PROPERTY OF THIRD PARTIES, INCLUDING, WITHOUT LIMITATIONS, CONSEQUENTIAL, DIRECT AND INDIRECT DAMAGES AND ANY ECONOMIC LOSSES, THAT ARISE OUT OF OR IN CONNECTION WITH YOUR USE, MODIFICATION, OR DISTRIBUTION OF THIS SPECIFICATION, SOFTWARE RELATED THERETO, CODE AND/OR PROGRAM RELATED THERETO, ITS OUTPUT, OR ANY ACCOMPANYING DOCUMENTATION.
GOVERNING LAW: THE RELATIONSHIP BETWEEN YOU AND ROBERT BOSCH GMBH SHALL BE GOVERNED SOLELY BY THE LAWS OF THE FEDERAL REPUBLIC OF GERMANY. THE STIPULATIONS OF INTERNATIONAL CONVENTIONS REGARDING THE INTERNATIONAL SALE OF GOODS SHALL NOT BE APPLICABLE. THE EXCLUSIVE LEGAL VENUE SHALL BE DUESSELDORF, GERMANY.
MANDATORY LAW SHALL BE UNAFFECTED BY THE FOREGOING PARAGRAPHS.
INTELLECTUAL PROPERTY OWNERS/COPYRIGHT OWNERS/CONTRIBUTORS: ROBERT BOSCH GMBH, ROBERT BOSCH PLATZ 1, 70839 GERLINGEN, GERMANY AND ITS LICENSORS.