Controller Area Network with Flexible Data (CAN FD) usage in XMC7000 family

About this document

Scope and purpose

This application note describes how to use Controller Area Network with Flexible Data (CAN FD) rate for Infineon XMC7000 family microcontrollers.

Intended audience

This document is for anyone who uses the XMC7000 MCU family to know Controller Area Network with Flexible Data (CAN FD) usage.

Associated part family

XMC7000 family XMC7100/XMC7200 series

Table of contents

Table of contents

About this document **1**
Table of contents **2**
1 Introduction **3**
2 Overview of CAN FD **4**
2.1 CAN FD network 4
2.2 CAN FD messages 5
2.2.1 CAN FD fields 5
2.2.2 Bit timing 6
3 CAN FD controller in XMC7000 family **9**
4 CAN FD settings **11**
4.1 CAN FD setup 11
4.2 Initialize CAN FD 11
4.2.1 Use case 14
4.2.2 Configuration for CAN FD controller 14
4.2.3 Example code to initialize CAN FD 16
4.3 Message transmission 17
4.3.1 Use case 18
4.3.2 Configuration 18
4.3.3 Example code of message transmission 19
4.4 Message reception 20
4.4.1 Message reception in dedicated Rx buffer 20
4.4.2 Message reception in Rx FIFO 0/1 21
4.4.3 Example code of message reception 23
5 Glossary **26**
References **27**
Revision history **28**
Disclaimer **29**

1 Introduction

This application note is intended for users of the Infineon XMC7000 family microcontrollers. The application note describes how to use CAN FD for Infineon XMC7000 family devices.

CAN FD is an extension of CAN (nowadays called ‘Classical CAN’). CAN FD can transmit data frames of up to 64 bytes at bit rates exceeding the 1 Mbps limit of Classical CAN. The maximum achievable bus speed in the data segment is limited only by external components such as transceivers and the particular network topology of an application. There are transceivers supporting 5 Mbps; several new products claim speeds up to 8 Mbps.

The CAN FD Controller (M_TTCAN) in XMC7000 supports Classical CAN as well as CAN FD (ISO 11898-1:2015) and Time-Triggered (TT) communication on CAN (ISO 11898-4:2004). The CAN FD Controller has been certified according to ISO 16845:2015.

This document is applicable to XMC7100/XMC7200 series devices.

2 Overview of CAN FD

This section describes the operation of CAN FD communication with an example of the CAN FD network followed by the CAN FD message format and the bit timing considerations.

2.1 CAN FD network

Figure 1 shows an example of the CAN FD network.

Two communication lines (CANH, CANL) are used in the CAN FD network to make it resilient against noise. Multiple Electronic Control Units (ECUs) can be connected to the CAN FD network; data is exchanged between the ECUs.

A receiver node converts the differential bus voltage to a digital signal by the CAN FD Transceiver; received data is handled by the CAN FD Control Logic of the microcontroller. In transmission, data is transmitted from the CAN FD Control Logic to the CAN FD Transceiver that drives a differential signal onto the CANH and CANL lines of the CAN FD network.

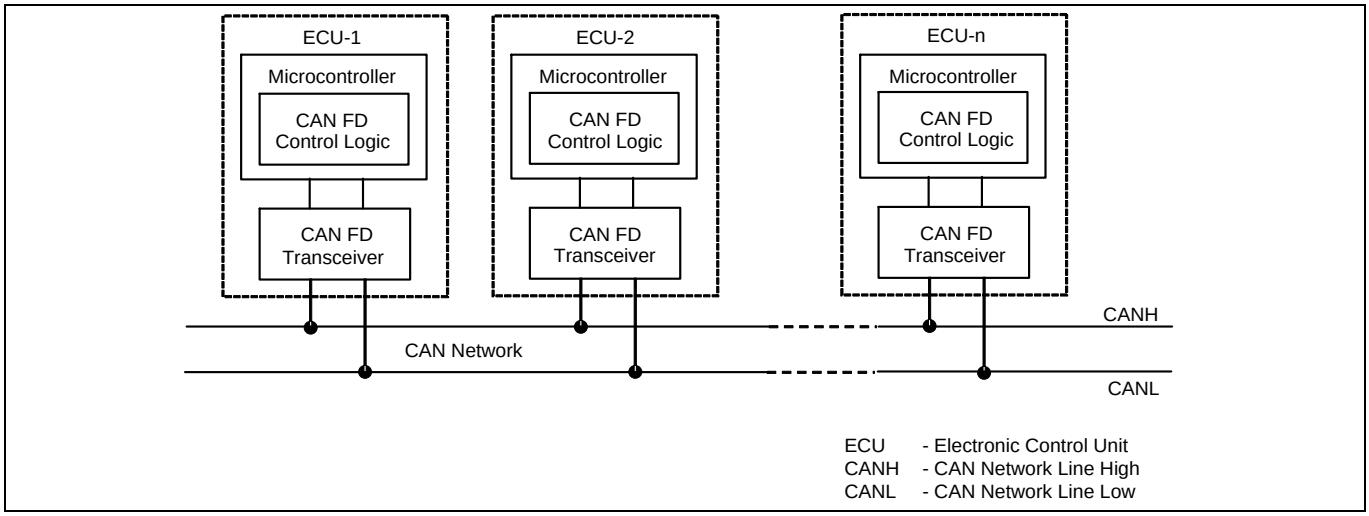


Figure 1 CAN FD network

2.2 CAN FD messages

There are four frame types: DATA FRAME, REMOTE FRAME, ERROR FRAME, and OVERLOAD FRAME. This section will explain the DATA FRAME.

Figure 2 shows the DATA FRAME formats of the Classical CAN and CAN FD message frame. As already mentioned, CAN FD is an extension of Classical CAN and both message formats are equal during the arbitration segment and after the CRC field. The differences occur in the data segment; the CAN FD frame has more data bytes and can be transmitted at higher speeds than the arbitration baud rate.

The maximum data length in Classical CAN is 8 bytes with a maximum baud rate of 1 Mbps.

CAN FD supports data lengths of up to 64 bytes with a maximum baud rate of 1 Mbps for the arbitration phase. The data communication speed can exceed the 1 Mbps limit set by Classical CAN and is only limited by external components such as transceivers and the network topology.

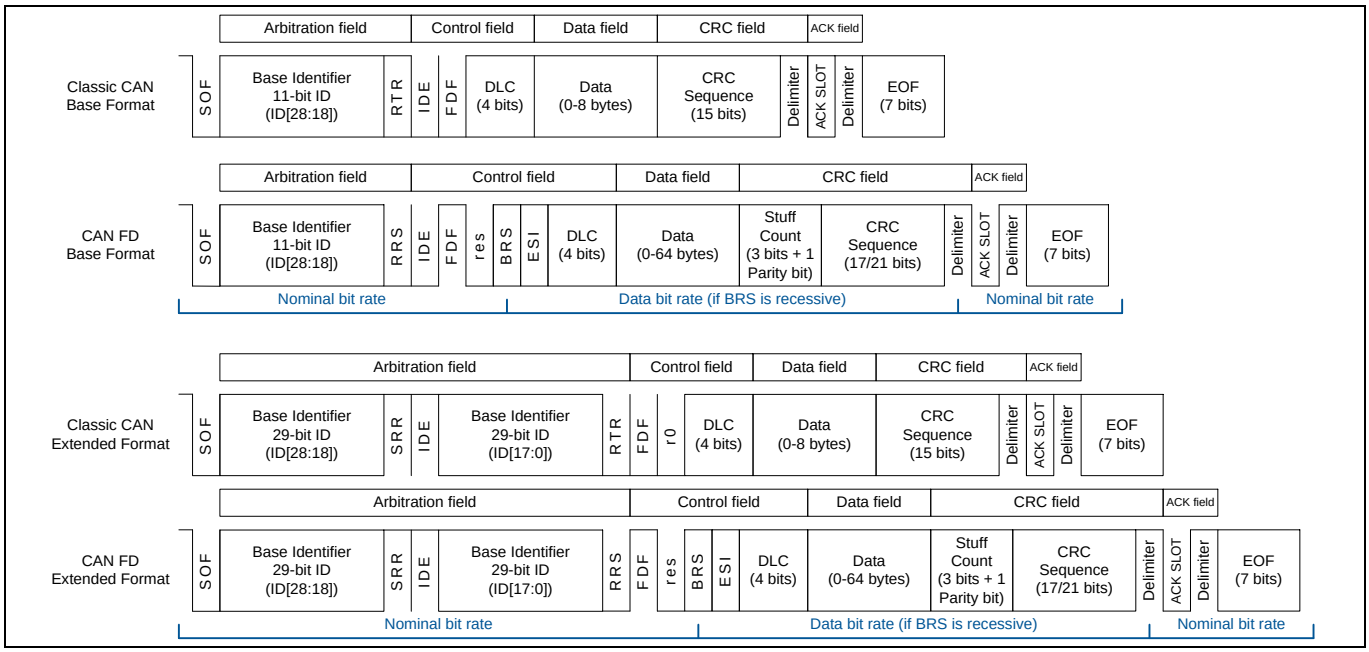


Figure 2 DATA FRAME formats

2.2.1 CAN FD fields

The fields of the CAN FD frame format include an Arbitration field, a Control field, a Data field, a CRC field, and an ACK field.

The Arbitration field contains the message ID number and determines the priority of the message among other messages from other nodes trying to start a transmission simultaneously. The message ID can be 11-bits (Base Format) or 29-bits (Extended Format), configured by the “IDE” bit.

The FD Format (FDF) indicator bit in the Control field identifies the frame type as CAN or CAN FD. The FDF bit is recessive (‘1’) for CAN FD frames and dominant (‘0’) for CAN frames. If the Bit Rate Switch (BRS) bit is recessive, the bit rate of the data field is switched to another, typically higher speed; if the BRS bit is dominant, the bit rate of the data field remains the arbitration bit rate. The Error State Indicator (ESI) bit is used for the identification of the error state of the CAN FD node. BRS and ESI bits are only available in CAN FD frames.

Overview of CAN FD

Furthermore, the Data Length Code (DLC) has four bits and it indicates how many bytes of data are transmitted. This settable range is 0–8 bytes for CAN frames and up to 64 bytes in CAN FD frames. Table 1 shows the relationship between the DLC field and the number of transmitted data bytes.

Table 1 Coding of DLC in CAN and CAN FD

DLC	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Number of data bytes in CAN	0	1	2	3	4	5	6	7	8	8	8	8	8	8	8	8
Number of data bytes in CAN FD	0	1	2	3	4	5	6	7	8	12	16	20	24	32	48	64

The Data field carries the message data and is sized by the data length set by DLC.

The CRC field consists of a CRC sequence and a CRC delimiter. For CAN frames, the CRC sequence has a fixed length of 15 bits. CAN FD frames additionally consist of a 4-bit Stuff Count at the beginning of the CRC field, followed by the CRC sequence (17 bits when the data length is 0–16 bytes; 21 bits for data lengths greater than 16 bytes). Any receiver can analyze the received data stream of a message and compare it with the transmitted CRC, and thus identify a valid or incorrectly received message.

The ACK field consists of an ACK slot and an ACK delimiter. The transmitter node sends an ACK as recessive bits and one or more receivers overwrite this with a dominant bit if message reception is successful. This helps the transmitter to determine whether the frame was received successfully or was corrupted.

The frame concludes with a flag sequence of seven recessive bits forming the end-of-frame (EOF).

2.2.2 Bit timing

The Classical CAN operation defines a single bit time for the entire message frame. The CAN FD operation defines two-bit times – nominal bit time and a data bit time. The nominal bit time is for the arbitration phase. The data bit time is equal to or shorter than nominal bit time and can be used to accelerate the data phase.

The basic construction of a bit time is shared with both nominal and data bit times. The bit time can be divided into four segments according to the CAN specifications (see Figure 3): the synchronization segment (Sync_Seg), the propagation time segment (Prop_Seg), the phase buffer segment 1 (Phase_Seg1), and the phase buffer segment 2 (Phase_Seg2). The sample point, the point of time at which the bus level is read and interpreted as the value of that respective bit, is located at the end of Phase_Seg1.

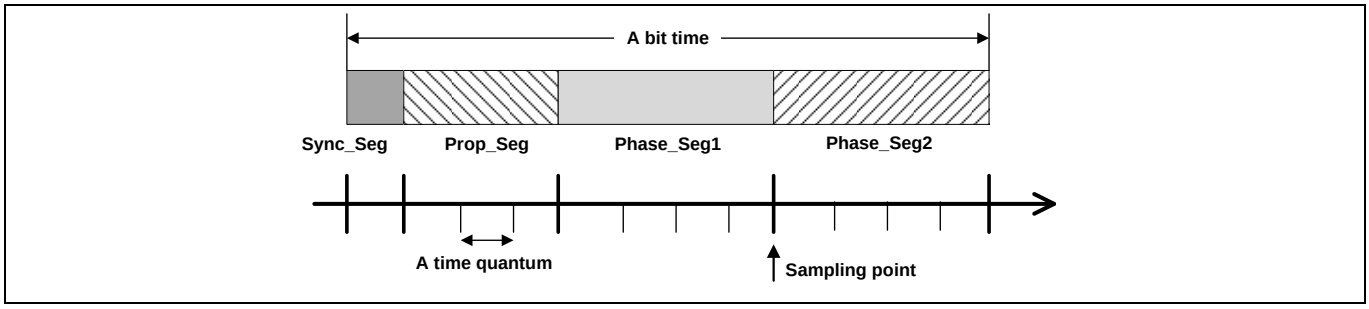


Figure 3 Bit time construction

Overview of CAN FD

Each segment consists of a programmable number of time quanta, which is a multiple of the time quantum that is defined by the CAN clock and a Prescaler. The values and Prescalers used to define these parameters differ for the nominal and data bit times, and are configured by Nominal Bit Timing & Prescaler Register (NBTP) and Data Bit Timing & Prescaler Register (DBTP) as listed in Table 2.

Table 2 CAN bit timing parameters

Parameter	Description
Time quantum tq (nominal) and tqd (data)	Time quantum. Derived by multiplying the basic unit time quanta (i.e. the CAN clock period) with the respective prescaler. The time quantum is configured by the CAN FD Controller as nominal: tq = (NBTP.NBRP[8:0] +1) × CAN clock period data: tqd = (DBTP.DBRP[4:0] + 1) × CAN clock period
Sync_Seg	Sync_Seg is fixed to one-time quantum as defined by the CAN specifications and is therefore not configurable (inherently built into the CAN FD Controller). nominal: 1 tq data: 1 tqd
Prop_Seg	Prop_Seg is the part of the bit time that is used to compensate for the physical delay times within the network. The CAN FD Controller configures the sum of Prop_Seg and Phase_Seg1 with a single parameter, i.e., nominal: Prop_Seg + Phase_Seg1 = NBTP.NTSEG1[7:0] + 1 data: Prop_Seg + Phase_Seg1 = DBTP.DTSEG1[4:0] + 1
Phase_Seg1	Phase_Seg1 is used to compensate for edge phase errors before the sampling point. Can be lengthened by the resynchronization jump width. The sum of Prop_Seg and Phase_Seg1 is configured by the CAN FD Controller as nominal: NBTP.NTSEG1[7:0] + 1 data: DBTP.DTSEG1[4:0] + 1
Phase_Seg2	Phase_Seg2 is used to compensate for edge phase errors after the sampling point. Can be shortened by the resynchronization jump width. Phase_Seg2 is configured by the CAN FD Controller as nominal: NBTP.NTSEG2[6:0] + 1 data: DBTP.DTSEG2[3:0] + 1
SJW	Resynchronization Jump Width. Used to automatically compensate timing fluctuation between nodes and adjust the length of Phase_Seg1 and Phase_Seg2. SJW will not be longer than either Phase_Seg1 or Phase_Seg2. SJW is configured by the CAN FD Controller as nominal: NBTP.NSJW[6:0] + 1 data: DBTP.DSJW[3:0] + 1

Overview of CAN FD

These relations result in the following equations for the nominal and data bit times:

Nominal bit time

$$= [Sync_Seg \ + \ Prop_Seg \ + \ Phase_Seg1 \ + \ Phase_Seg2] \times tq$$

$$= [1 \ + \ (NBTP.NTSEG1[7:0] + 1) + (NBTP.NTSEG2[6:0] + 1)] \times [(NBTP.NBRP[8:0] + 1) \times CAN\ clock\ period]$$

Example (500 kbps with sampling point of 75%)

$$= [1 + (13 + 1) + (4 + 1)] \times [(3 + 1) \times (1/40000000)] = 0.000002 \ \ (500\ kbps\ )$$

Data bit time

$$= [1 \ + \ (DBTP.DTSEG1[4:0] + 1) + (DBTP.DTSEG2[3:0] + 1)] \times [(DBTP.DBRP[4:0] + 1) \times CAN\ clock\ period]$$

Example (5 Mbps with sampling point of 62.5%)

$$= [1 + (3 + 1) + (2 + 1)] \times [(0 + 1) \times (1/40000000)] = 0.0000002 \ \ (5\ Mbps)$$

Example (2 Mbps with sampling point of 60%)

$$= [1 + (10 + 1) + (7 + 1)] \times [(0 + 1) \times (1/40000000)] = 0.0000005 \ \ (2\ Mbps)$$

3 CAN FD controller in XMC7000 family

This section provides an overview of the CAN FD Controller in the XMC7000 family.

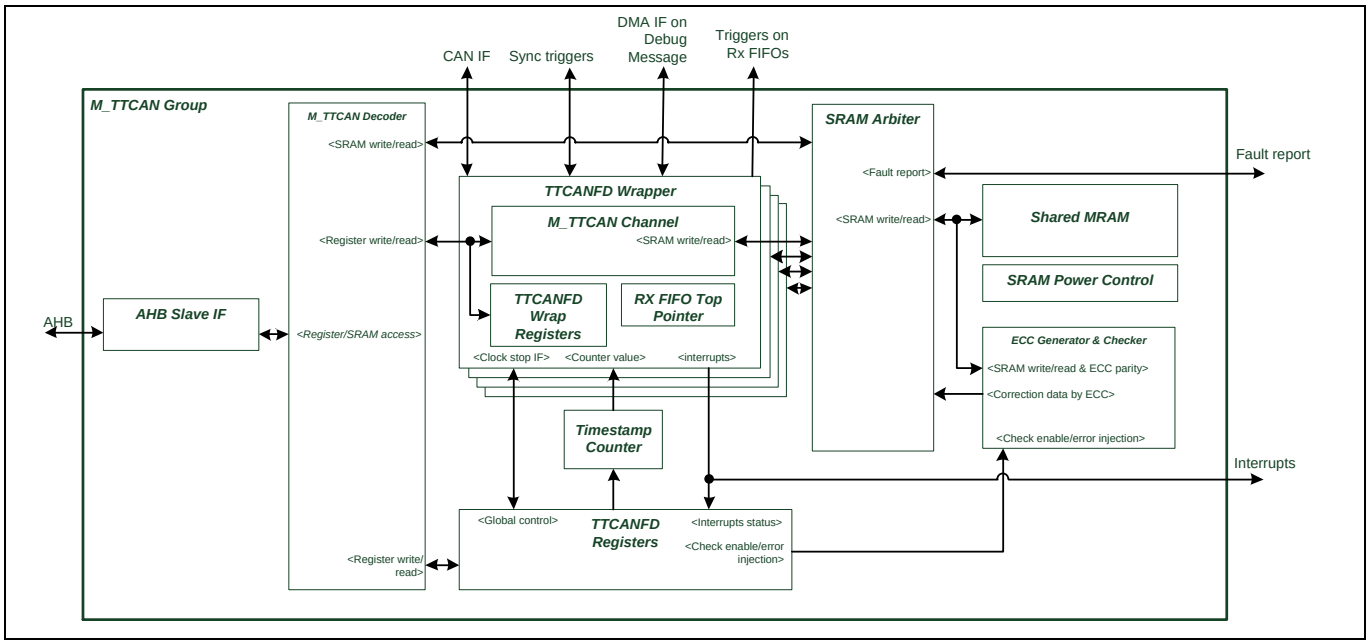


Figure 4 CAN FD controller block diagram

Figure 4 shows the block diagram of the CAN FD Controller (M_TTCAN) in XMC7000 devices. The M_TTCAN channels in XMC7000 devices are organized into groups, with each group consisting of one or more channels that share the Message RAM. The total number of available M_TTCAN groups and channels depends on the device variant. For details, see the device datasheet.

The M_TTCAN channels support Classical CAN and CAN FD operation according to ISO 11898-1:2015. M_TTCAN operation is available in Active and Sleep power modes; the IP is fully retained except the Time Stamp counter in Deep Sleep power mode.

The CAN Core, along with the Tx and Rx handlers is responsible for protocol handling; the slave interface to Memory Mapped I/O (MMIO) registers facilitates the configuration of the CAN FD Controller by the CPU. Each M_TTCAN channel has two clock inputs: cclk and hclk. The cclk is used for CAN FD operation and hclk is used for internal IP operation (for example, register accesses and Message RAM accesses).

Each M_TTCAN Group consists of one Message RAM, and this Message RAM is shared among the M_TTCAN channels belonging to that group. You should take care of distributing the Message RAM to the channels of that group and prevent any overlapping distribution. The CAN FD Controller does not check internally if any Message RAM region is overlapping for multiple channels of the group. The Message RAM is ECC protected with the single-bit error correction and double-bit error detection feature. ECC errors and out-of-range accesses to the Message RAM are reported to fault structures.

Each M_TTCAN channel consists of two interrupt lines (Interrupt 0 and Interrupt 1); you have the flexibility to route the Channel Interrupt sources to either Channel Interrupt 0 or Channel Interrupt 1. Channel Interrupt sources include the New Message received interrupt, Transmission completed interrupt, and Receive FIFO Watermark interrupt.

CAN FD controller in XMC7000 family

In addition to Channel Interrupt lines, Consolidated Interrupt 0 and Consolidated Interrupt 1 are available for each M_TTCAN Group. Consolidated Interrupt 0 is a logical OR of the Interrupt 0 lines of all channels of the group; similarly, Consolidated Interrupt 1 is the logical OR of the Interrupt 1 lines of all channels of the group. All Channel Interrupt lines and Consolidated Interrupt lines are routed to the Device Interrupt System.

To remove the software overhead for calculating an Rx pointer each time a frame is received, hardware logic is implemented. The Rx FIFO top pointer calculates the next read address and provides a single address (RXFTOPn_DATA) for each FIFO from where data can be read. This logic will also update a specific acknowledge index (RXFnA.FnAI) in the TTCAN register set so that the index is also incremented accordingly.

The following sections describe how to set up the CAN FD Controller to transmit and receive CAN FD messages.

4 CAN FD settings

This section describes how to configure CAN FD based on a use case using Peripheral Driver Library (PDL) in ModusToolbox™ provided by Infineon. See References section for the ModusToolbox™.

The CAN FD application basically has a configuration part. The configuration part mainly configures the parameter values for the desired operation. The PDL driver provides API to configure each register based on the parameter values in the configuration part.

You can use the device configurator of ModusToolbox™ to configure according to your system, and device configurator will auto-generate the CAN FD configuration code.

4.1 CAN FD setup

To set up CAN FD, do the following:

1. Initialize the CAN FD peripheral clock by configuring and assigning a clock divider to the CAN FD peripheral.
2. Enable the I/O ports used for CAN FD communication.
3. Map CAN FD system interrupt sources to available external CPU interrupts.
4. Initialize the CAN FD Controller.

For steps 1 to 3 set up, see the “Clocking System”, “Input/Output Subsystem”, and “Interrupts” sections in the architecture technical reference manual (TRM).

4.2 Initialize CAN FD

Figure 5 shows the flow to initialize the CAN FD controller. In this flow, (0) is performed in the configuration part, (1) to (9) is performed in the PDL driver part, and the Cy_CANFD_Init() implements CAN FD initialization in the cy_canfd.c file of PDL.

(0) Configure the parameter values according to the system.

(1) Set initialization register (CCCR.INIT) to “1” and stop CAN FD communication. Then, enable the Configuration Change Enable register (CCCR.CCE) to enable write access to the write-protected CAN FD configuration registers.

(2) Configure the number of elements of the message filter and the start address offset in the Message RAM with the Standard ID Filter Configuration (SIDFC) register and the Extended ID Filter Configuration register (XIDFC). Configure the Extended ID AND Mask (XIDAM) register for masking the ID bits that are not to be used for extended ID message acceptance filtering.

(3) For Rx and Tx messages, configure the element size of the Rx FIFO and start address offset in Message RAM with the Rx FIFO 0 Configuration (RXF0C) register and Rx FIFO 1 Configuration (RXF1C) register. The Rx FIFO Top pointer logic is enabled/disabled by setting the RXFTOP_CTL register.

Configure the Rx buffer start address offset in the Rx Buffer Configuration (RXBC) register and the data field size of Rx buffer or FIFO elements in the Rx Buffer/FIFO Element Size Configuration (RXESC) register.

If the application uses Tx event FIFO it must be configured in the TXEFC register. The event FIFO size, start address offset, and watermark level must be configured in this register.

CAN FD settings

Configure the number of Tx buffers and start address offset in the Message RAM with the Tx Buffer Configuration (TXBC) register. Set the size of the data field of the Tx buffer with the Tx Buffer Element Size Configuration (TXESC) register.

(4) Clear the Message RAM area intended to be allocated for this CAN FD channel. This Message RAM area will hold the Rx and Tx buffers and filter configurations for this channel.

(5) Configure the mode of operation – Classical CAN/CAN FD mode (CCCR.FDOE) and the Bit Rate Switch (CCCR.BRSE) in the CC Control Register (CCCR).

(6) Configure the Bit timing - Nominal Bit Timing & Prescaler Register (NBTP) used in the arbitration phase and the Data Bit Timing & Prescaler Register (DBTP) used in the data phase when the bit rate switch is enabled in CAN FD mode. Configure the Transmitter Delay Compensation Register (TDCR) for using higher bit rates during the CAN FD data phase.

(7) For message filters, determine the handling of received frames with message IDs that do not match any filters as set in the Global Filter Configuration (GFC) register.

Set up message filters in the address obtained by adding the start address offset (SIDFC/XIDFC) to the start address of Message RAM. Range Filter, Dual Filter, or Classic Bit Mask Filter can be configured. For details, see the Message RAM chapter in the architecture TRM.

(8) To enable Tx buffers to assert an interrupt upon transmission, configure the Tx Buffer Transmission Interrupt Enable (TXBTIE) register. Similarly, for Tx buffers to assert an interrupt upon completion of transmission cancellation, configure the Tx Buffer Cancellation Finished Interrupt Enable (TXBCIE) register. Clear the interrupt flags in the Interrupt Register (IR) and enable each interrupt in the Interrupt Enable (IE) register. The CAN FD Controller has dual interrupt lines; Interrupt Line Select (ILS) determines the line the interrupt is assigned to. Enable the interrupt line with Interrupt Line Enable (ILE).

(9) Set the Initialization register (CCCR.INIT) to ‘0’ to start the operation of CAN FD. The CAN FD channel is ready for transmitting/receiving messages once the read of CCCR.INIT results in a value of ‘0’.

Note: *Some external transceivers require to be configured (for example, via SPI interface) before they can facilitate CAN FD communication. For details, see the device datasheet of the transceiver used in your hardware.*

CAN FD settings

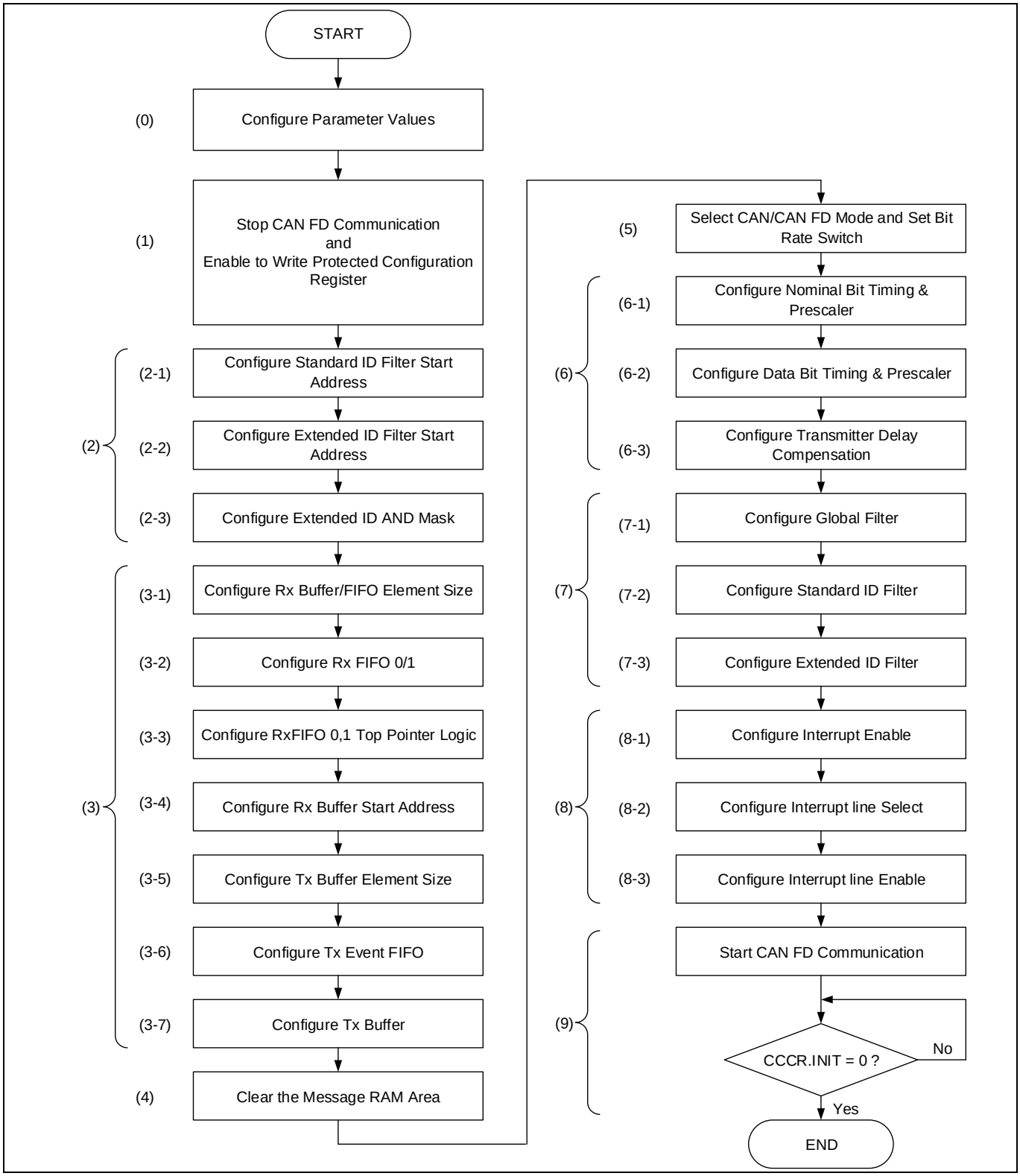


Figure 5 Example of CAN FD initialization flow

CAN FD settings

4.2.1 Use case

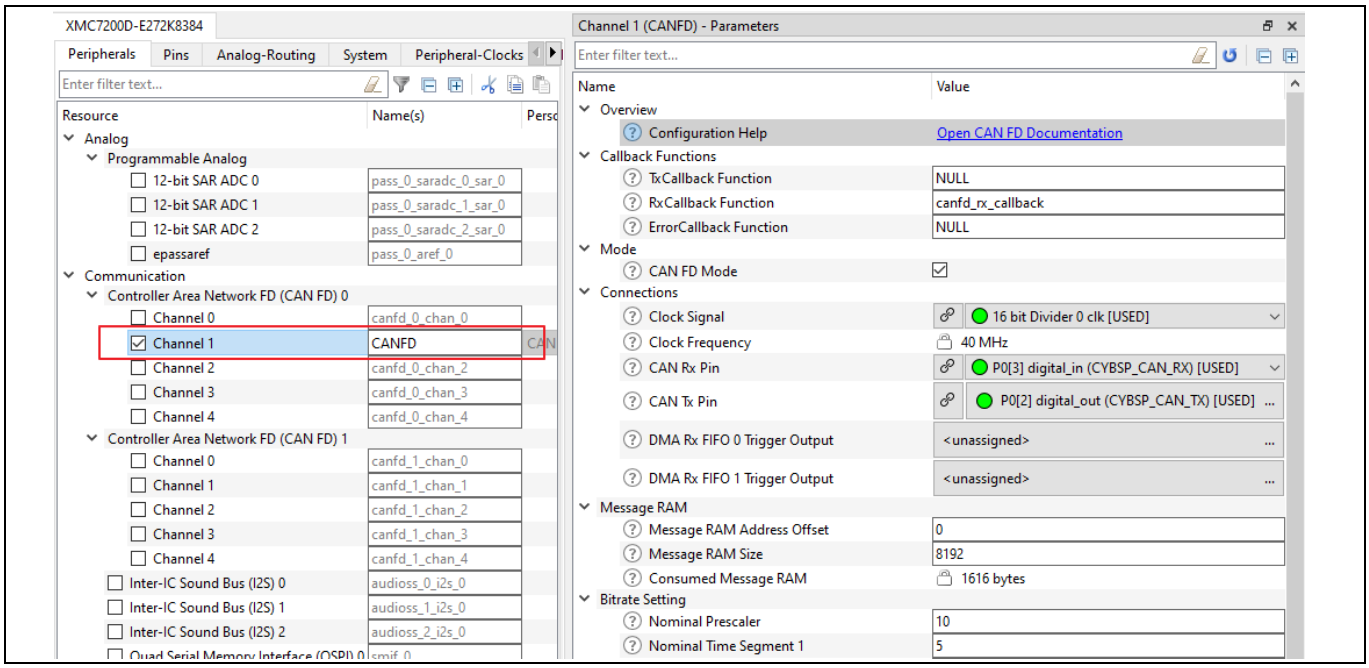
This section explains an example of CAN FD initialization using the following use case. CAN FD initialization is configured using PDL.

Use case:

- Mode: CAN FD
- CAN Instance: CAN0_CH1
- Interrupt handler: For CAN message reception
- Input Clock: 40 MHz
- Normal Bit rate (Sample point = 75%)
  - 500 kbps, 1 bit = 8 tq
  - Prescaler = 40 MHz / 500 kbps / 8 tq = 10
  - tseg1 = 5 tq, tseg2 = 2 tq, sjw = 2 tq
- Fast Bit rate (Sample point = 75%)
  - 1 Mbps, 1 bit = 8 tq
  - Prescaler = 40 MHz / 1 Mbps / 8 tq = 5
  - tseg1 = 5 tq, tseg2 = 2 tq, sjw = 2 tq
- Filter Configuration: Two Standard and Extended IDs
- Transceiver delay compensation: unused
- Rx/Tx Data Size: 64 bytes
- Number of Rx Buffers: 4

4.2.2 Configuration for CAN FD controller

Figure 6 shows the parameters of the configuration in the device configurator of ModusToolbox™ for CAN FD initialization.



CAN FD settings

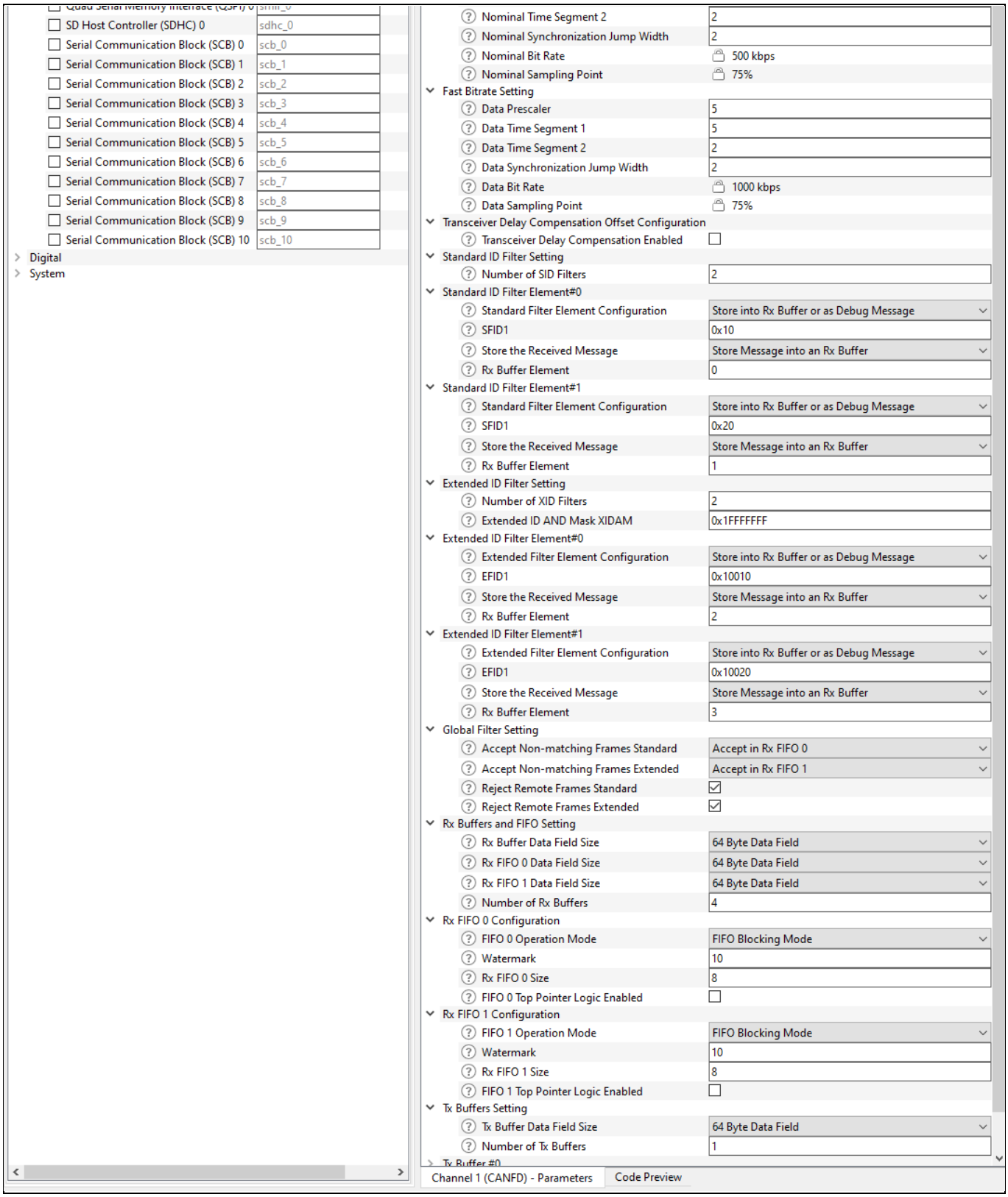


Figure 6 Example of CAN FD configuration

The device configurator of ModusToolbox™ will automatically generate the configuration code for CAN FD initialization.

4.2.3 Example code to initialize CAN FD

Code Listing 1 demonstrates an example program to initialize the CAN FD.

Code Listing 1 Example to initialize CAN FD

```
/* CAN channel number */
#define CAN_HW_CHANNEL          1

/* This structure initializes the CANFD interrupt for the NVIC */
cy_stc_sysint_t canfd_irq_cfg =
{
    .intrSrc  = (NvicMux2_IRQn << 16) | CANFD_IRQ_0, /* Source of interrupt signal */
    .intrPriority = 1U, /* Interrupt priority */
};

/* This structure initializes the button interrupt for the NVIC */
cy_stc_sysint_t button_intr_config =
{
    .intrSrc  = (NvicMux2_IRQn << 16) | CYBSP_USER_BTN_IRQ,
    .intrPriority = 0U,
};

/* This is a shared context structure, unique for each canfd channel */
cy_stc_canfd_context_t canfd_context;

/* This is the main function. It initializes the CANFD channel
* and interrupt. User button and User LED are also initialized. The main loop
* checks for the button pressed interrupt flag and when it is set, a CANFD frame
* is sent. Whenever a CANFD frame is received from other nodes, the user LED
* toggles and the received data is logged over serial terminal.
*/
int main(void)
{
    cy_rslt_t result;
    cy_en_canfd_status_t status;

    /* Initialize the device and board peripherals */
    result = cybsp_init() ;
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Initialize retarget-io for uart logging */
    result = cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX,
                                 CY_RETARGET_IO_BAUDRATE);
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }
```

CAN FD settings

Code Listing 1 Example to initialize CAN FD

```
    /* \x1b[2J\x1b[;H - ANSI ESC sequence for clear screen */
    printf("\x1b[2J\x1b[;H");
    printf("============================================================\r\n");
    printf("XMC7000 MCU: CANFD example\r\n");
    printf("============================================================\r\n\n");

    /* Hook the interrupt service routine and enable the interrupt */
    (void) Cy_SysInt_Init(&canfd_irq_cfg, &isr_canfd);
    Cy_SysInt_Init(&button_intr_config, isr_button);
    NVIC_EnableIRQ(NvicMux2_IRQn);

    /* Enable global interrupts */
    __enable_irq();

    /* Initialize CANFD Channel */
    status = Cy_CANFD_Init(CANFD_HW, CAN_HW_CHANNEL, &CANFD_config, &canfd_context);
    if (status != CY_CANFD_SUCCESS)
    {
        CY_ASSERT(0);
    }

    for(;;)
    {
        :
        :
    }
}

/* This is the interrupt handler function for the canfd interrupt */
void isr_canfd(void)
{
    /* Just call the IRQ handler with the current channel number and context */
    Cy_CANFD_IrqHandler(CANFD_HW, CAN_HW_CHANNEL, &canfd_context);
}
```

4.3 Message transmission

Figure 7 is an example of message transmission flow. This example does not use the Tx Interrupt. In this flow, (0) is performed in the configuration part, and (1) to (5) is performed in the PDL driver part.

The message is sent via the Tx buffer in the Message RAM area. Ensure that there are no pending requests (TXBRP). If there is no pending request, calculate the Tx buffer address of the Message RAM and write the control information and data of the frame to be transmitted by the CAN FD Controller. Message transmission is started by writing to the Tx Buffer Add Request (TXBAR) register.

CAN FD settings

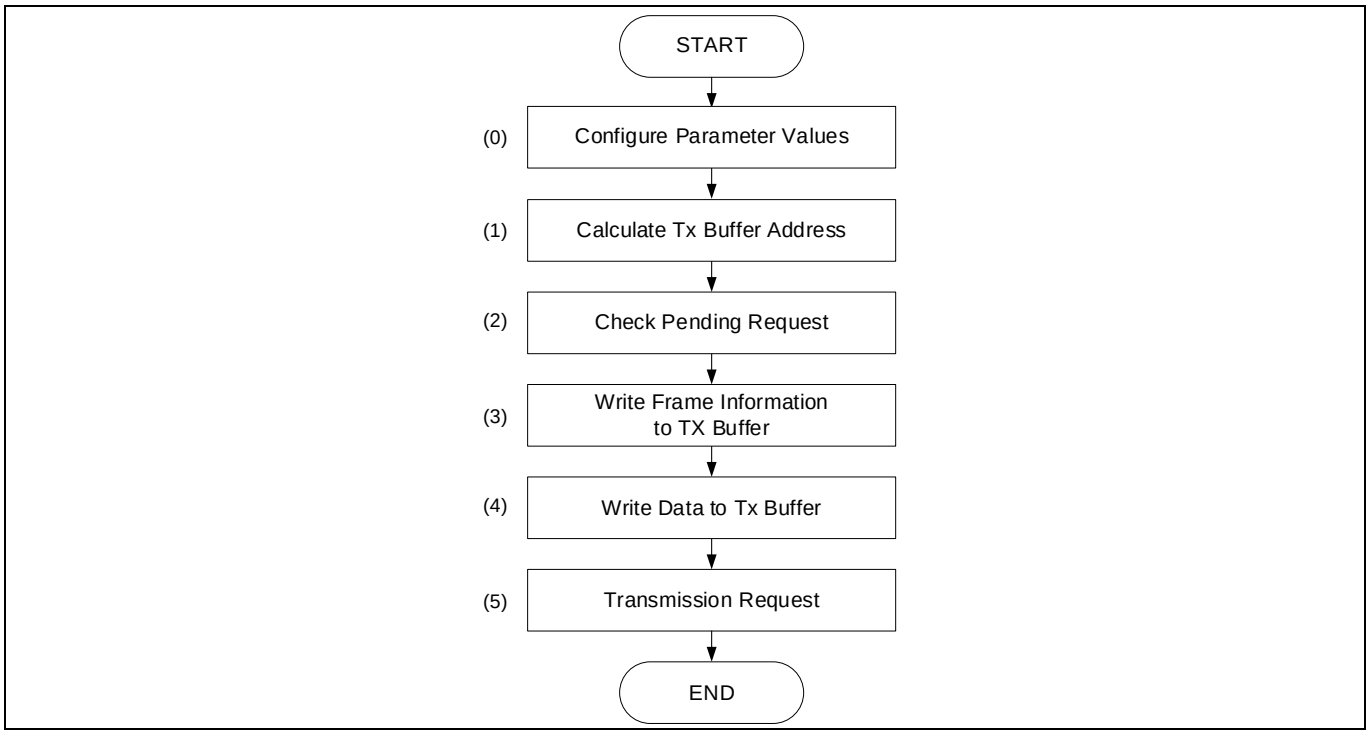


Figure 7 Example of message transmission flow

4.3.1 Use case

This section explains an example of CAN FD message transmission using the following use case and the use case discussed in 4.2.1 Use case. CAN FD message transmission is configured using PDL.

Use case:

- FD Format (FDF): 1 (Frame transmitted in CAN FD format)
  - Bit Rate Switching (BRS): 1 (CAN FD frame transmitted with bit rate switching)
- Extended Identifier (XTD): 0 (11-bit standard identifier)
- Identifier (ID): 0x525
- Data Length Code (DLC): 15

4.3.2 Configuration

Figure 8 shows the parameters of the configuration part in the device configurator of ModusToolbox™ for message transmission.

CAN FD settings

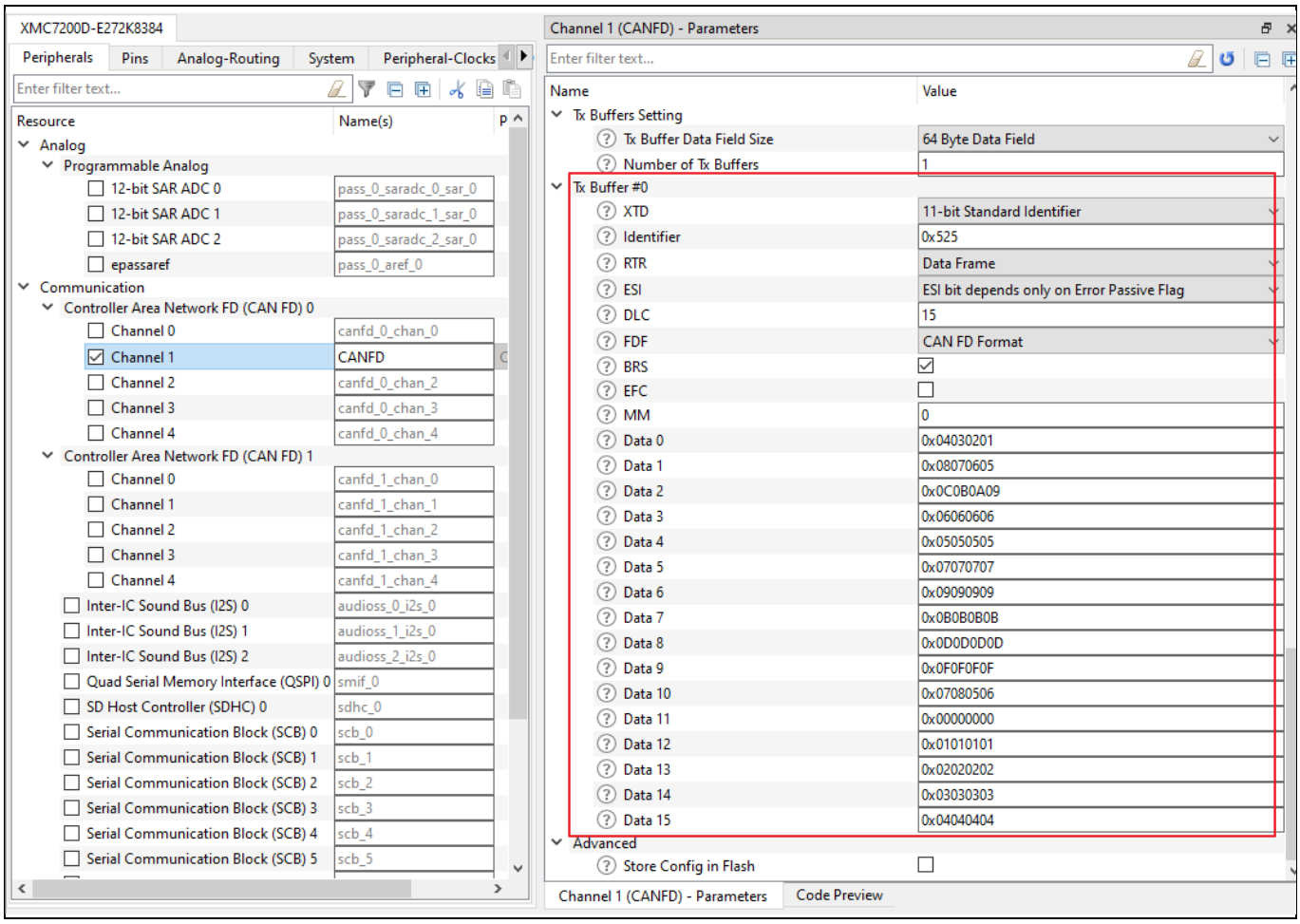


Figure 8 Example of CAN FD message transmission configurations

4.3.3 Example code of message transmission

Code Listing 2 demonstrates an example code of CAN FD message transmission. When the user button is pressed, call PDL API Cy_CANFD_UpdateAndTransmitMsgBuffer() to transfer the CAN message.

Code Listing 2 Example of message transmission

```
/* CAN Tx buffer index */
#define CAN_BUFFER_INDEX        0

int main(void)
{
   :
    for(;;)
    {
        if (ButtonIntrFlag == true)
        {
            ButtonIntrFlag = false;
            /* Sending CANFD frame */
            status = Cy_CANFD_UpdateAndTransmitMsgBuffer(CANFD_HW,
                                                         CAN_HW_CHANNEL,
```

CAN Transmission setting
CAN0 Channel 1
Message buffer 0
Transmission data

CAN FD settings

Code Listing 2 Example of message transmission

```
                                                &CANFD_txBuffer_0,
                                                CAN_BUFFER_INDEX,
                                                &canfd_context);
        }
    }
}

/* Button interrupt callback function */
void isr_button (void)
{
    uint32_t intStatus = 0;

    /* If user button falling edge detected */
    intStatus = Cy_GPIO_GetInterruptStatusMasked(CYBSP_USER_BTN_PORT, CYBSP_USER_BTN_PIN);
    if (intStatus != 0ul)
    {
        /* Set button interrupt flag */
        ButtonIntrFlag = true;
        /* Clears the triggered pin interrupt */
        Cy_GPIO_ClearInterrupt(CYBSP_USER_BTN_PORT, CYBSP_USER_BTN_PIN);
    }
}
```

4.4 Message reception

Based on the filter configuration, message reception can be done in dedicated Rx buffers or in Rx FIFO 0/1. This section describes the message reception methods.

4.4.1 Message reception in dedicated Rx buffer

Figure 9 shows an example of the message reception flow using the dedicated Rx buffer and Rx interrupt.

When a received message passes acceptance filtering and is stored in one of the dedicated Rx buffers of the Message RAM area, an interrupt occurs at this event if Rx interrupt is enabled. When the message is stored in the dedicated Rx buffer, the corresponding bits of the Interrupt Register (IR.DRX) and New Data register 1/2 (NDAT 1/2) are set. Interrupt handling involves the calculation of the absolute address of the Rx buffer in the Message RAM holding the received message and reading the received message information from the calculated address. After the message is read from the Rx buffer, the corresponding flag in the NDAT 1/2 register must be cleared to enable this Rx buffer to receive the next message.

CAN FD settings

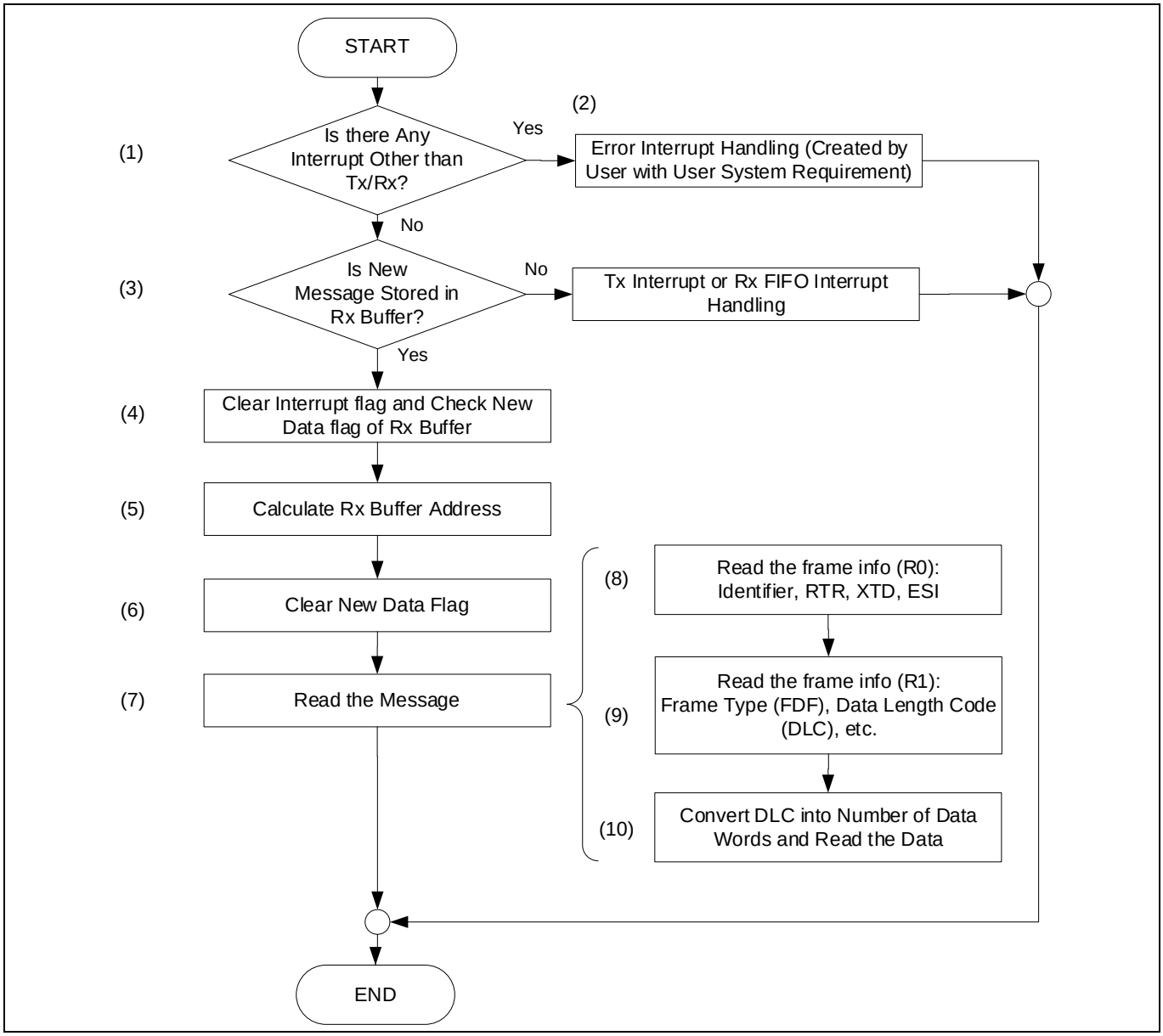


Figure 9 Example of message reception in dedicated Rx buffer flow

4.4.2 Message reception in Rx FIFO 0/1

When a received message passes the acceptance filtering and is stored in Rx FIFO 0/1 of the Message RAM area, an interrupt occurs at this event if Rx FIFO interrupts are enabled. The received message is stored in the Rx FIFO at the buffer position pointed to by the Rx FIFO Put Index; the corresponding bit in the Interrupt Register (IR.RF0N/RF1N) is set. The messages in the FIFO are always read out from the position pointed by the Rx FIFO Get Index. This is depicted in an example in Figure 10 with eight FIFO elements.

CAN FD settings

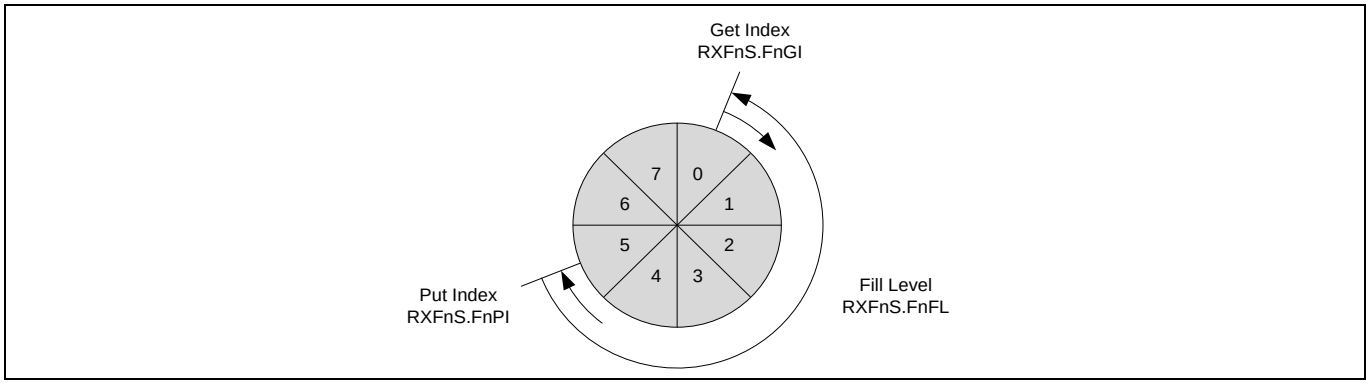


Figure 10 Example of Rx FIFO with eight elements

The conventional method of Rx FIFO Message handling involves three steps:

1. Calculating the absolute address of the buffer at the Get Index position
2. Reading the received message information
3. Acknowledging the message at the Get Index position

This method comes with the disadvantage of software overhead; to eliminate this overhead, XMC7000 implements a hardware logic on top of the Rx FIFOs. The Rx FIFO top pointer logic provides a single source register (RXFTOPn_DATA) to read out the message content from the Get Index position, thus eliminating the need for absolute address calculation. Also, the Top pointer logic takes care of acknowledging the message at the Get Index position when all words of the message are read out via the RXFTOPn_DATA register.

For example, when the Rx FIFO element size is configured to be 18 words, the RXFTOPn_DATA register must be read 18 times to read the complete message; after the 18th read, the message at the Get Index is automatically acknowledged.

Figure 11 shows an example of the message reception flow using the Rx FIFO and Rx Interrupt. The example uses only the Rx FIFO New Message Interrupt.

CAN FD settings

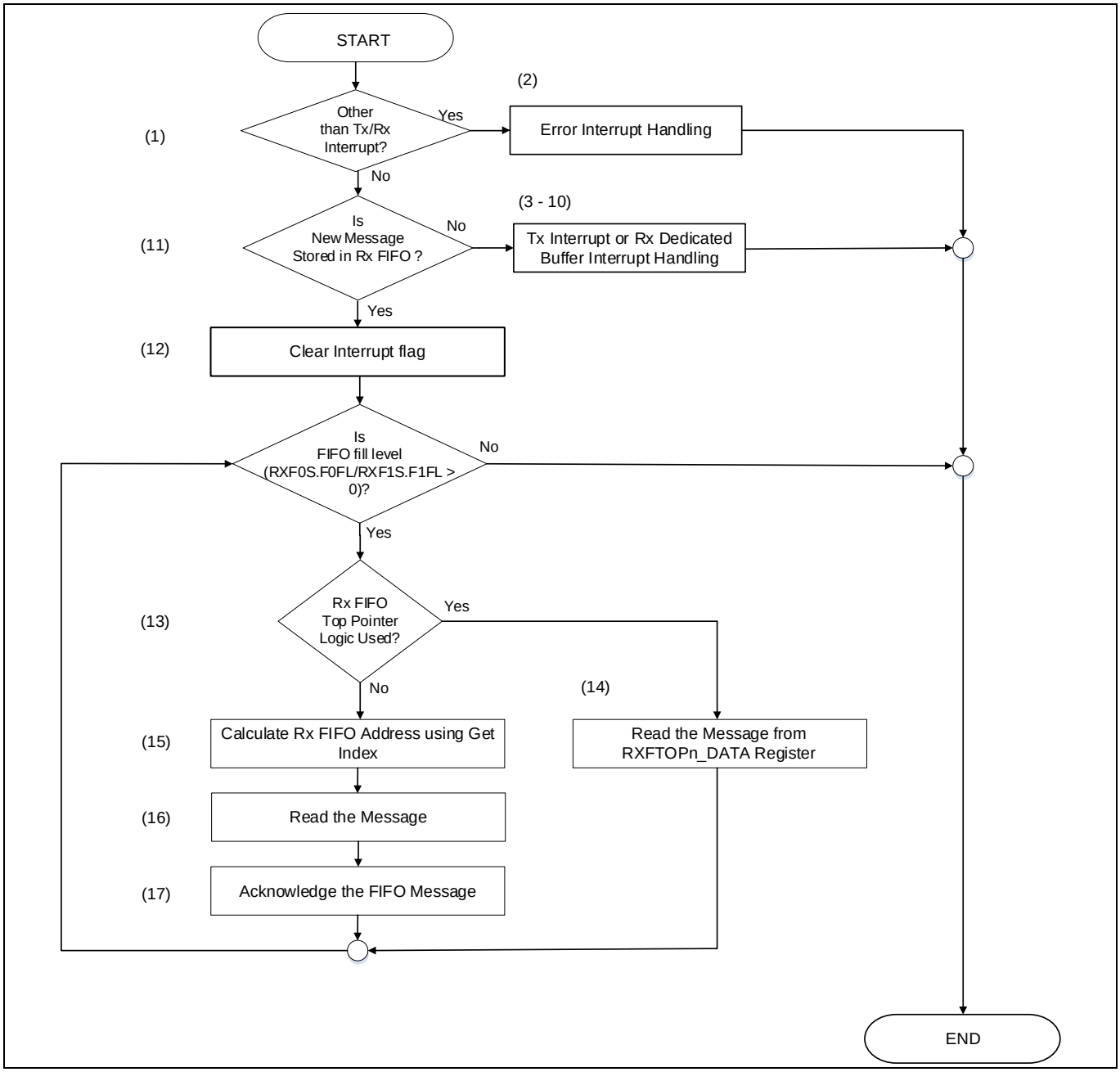


Figure 11 Example of message reception in Rx FIFO flow

4.4.3 Example code of message reception

PDL CAN FD driver implements Cy_CANFD_IrqHandler() API for the CAN FD interrupt service routine. It reads data from the dedicated Rx buffers and Rx FIFO buffers. Corresponding callback functions are called for error interrupts (Rx interrupts and Tx complete interrupt). For details, please check the CAN FD PDL driver implementation in cy_canfd.c file.

In the CAN FD initialization configuration shown in Figure 6, set “RxCallback Function” to “canfd_rx_callback”, this callback will handle the CAN FD reception.

Code Listing 3 shows CAN FD Rx callback implementation, it will print the received message by UART.

CAN FD settings

Code Listing 3 CAN FD Rx callback

```
/* This is the callback function for canfd reception */
void canfd_rx_callback (bool                        rxFIFOMsg,
                        uint8_t                     msgBufOrRxFIFONum,
                        cy_stc_canfd_rx_buffer_t*   basemsg)
{
    /* Array to hold the data bytes of the CANFD frame */
    uint8_t canfd_data_buffer[64];
    /* Variable to hold the data length of the CANFD frame */
    int canfd_datalen;
    /* Variable to hold the Identifier of the CANFD frame */
    int canfd_id;

    /* Message was received in Rx FIFO */
    if (rxFIFOMsg == true)
    {
        printf("Message received in Rx FIFO %d", msgBufOrRxFIFONum);
    }
    else /* Message was received in Rx buffer */
    {
        printf("Message received in Rx Buffer %d", msgBufOrRxFIFONum);
    }

    /* Checking whether the frame received is a data frame */
    if(CY_CANFD_RTR_DATA_FRAME == basemsg->r0_f->rtr)
    {
        /* Toggle the user LED */
        Cy_GPIO_Inv(CYBSP_USER_LED_PORT, CYBSP_USER_LED_PIN);
        /* Get the CAN DLC and ID from received message */
        canfd_datalen = canfd_dlc_to_bytes(basemsg->r1_f->fdf, basemsg->r1_f->dlc);
        canfd_id  = basemsg->r0_f->id;
        /* Print the received message by UART */
        printf(", %d bytes received from Node-%d with identifier %d\r\n\r\n",
                                                canfd_datalen,
                                                canfd_id,
                                                canfd_id);
        memcpy(canfd_data_buffer, basemsg->data_area_f, canfd_datalen);
        printf("Rx Data : ");
        for (uint8_t msg_idx = 0; msg_idx < canfd_datalen ; msg_idx++)
        {
            printf(" %d ", canfd_data_buffer[msg_idx]);
        }
        printf("\r\n");
    }
    printf("\r\n");
}

/* Convert the CAN data length code to number of bytes */
uint8_t canfd_dlc_to_bytes(cy_en_canfd_fdf_t fdf, uint32_t dlc)
{
```

CAN FD settings

Code Listing 3 CAN FD Rx callback

```
    if(CY_CANFD_CLASSIC_CAN_DATA_LENGTH >= dlc)
    {
        return dlc;
    }
    /* Standard CAN frame and DLC greater than 8 */
    if(CY_CANFD_FDF_STANDARD_FRAME == fdf)
    {
        return CY_CANFD_CLASSIC_CAN_DATA_LENGTH;
    }
    /* CAN FD frame and DLC greater than 8 */
    switch(dlc)
    {
        case 9:
            return 12;
        case 10:
            return 16;
        case 11:
            return 20;
        case 12:
            return 24;
        case 13:
            return 32;
        case 14:
            return 48;
        default:
            return 64;
    }
}
```

5 Glossary

Table 3 Glossary

Terms	Description
ACK	Acknowledgement
BRS	Bit Rate Switch
CAN	Controller Area Network
CAN FD	Controller Area Network with Flexible Data rate
CANH	CAN Network Line High
CANL	CAN Network Line Low
CRC	Cyclic Redundancy Check
DLC	Data Length Code
ECC	Error Correction Code
ECU	Electronic Control Unit
EOF	End of Frame
ESI	Error State Indicator
FDF	FD Format indicator
FIFO	First in First out
ID	Identifier
IDE	Identifier Extension
MMIO	Memory Mapped I/O
RAM	Random Access Memory
RTR	Remote Transmission Request
SOF	Start of Frame
SPI	Serial Peripheral Interface

References

The following are the XMC7000 family series datasheets and technical reference manuals. Contact Technical support to obtain more documents.

[1] Device datasheet

- XMC7200 Datasheet 32-Bit Arm® Cortex®-M7 microcontroller XMC7000 family
- XMC7100 Datasheet 32-Bit Arm® Cortex®-M7 microcontroller XMC7000 family

[2] Technical reference manual

- XMC7000 MCU family architecture technical reference manual (TRM)
- XMC7200 registers technical reference manual (TRM)
- XMC7100 registers technical reference manual (TRM)

[3] Application notes

- AN234334 - Getting started with XMC7000 MCU on ModusToolbox™ software

Revision history

Document revision	Date	Description of changes
**	2021-12-14	Initial release.
*A	2023-08-29	Deleted Table 3 and Table 4. Deleted all driver code listings in chapters 4.2.3, 4.3.3, and 4.4.3. Updated the code examples in chapter 4.2.3, 4.3.3, and 4.4.3. Updated all hyperlinks in References chapter. Removed the Other references chapter. Updated the template.

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2023-08-30

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2023 Infineon Technologies AG. All Rights Reserved.

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

002-34022 Rev. *A

Important notice

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.