CAN FD usage in TRAVEO™ T2G 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 TRAVEO™ T2G family microcontrollers.

Intended audience

This document is intended for anyone who uses Infineon TRAVEO™ T2G MCUs for CAN FD.

Associated part family

TRAVEO™ T2G family CYT2/CYT3/CYT4/CYT6 series

Table of contents

1 Introduction

The application note describes how to use CAN FD for Infineon TRAVEO™ T2G 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 for 8 Mbps.

The CAN FD controller (M_TTCAN) in TRAVEO™ T2G 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 CYT2/CYT3/CYT4/CYT6 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.

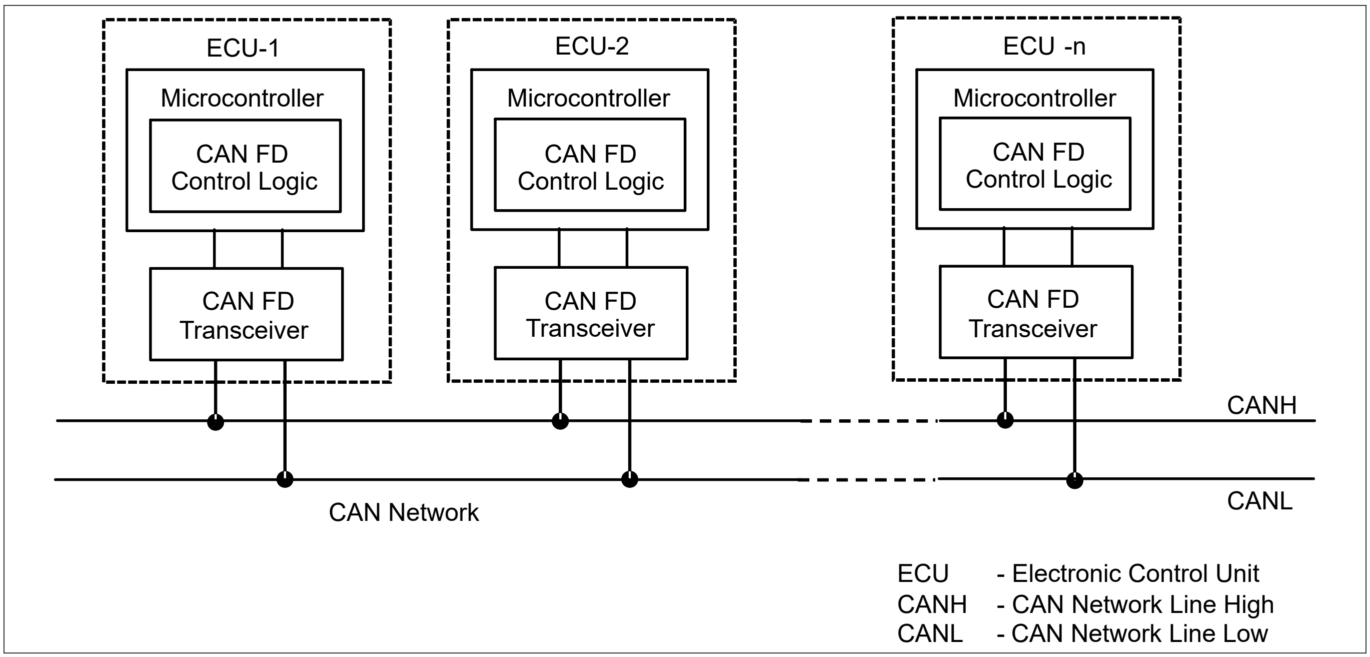

Figure 1 CAN FD network

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 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 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.

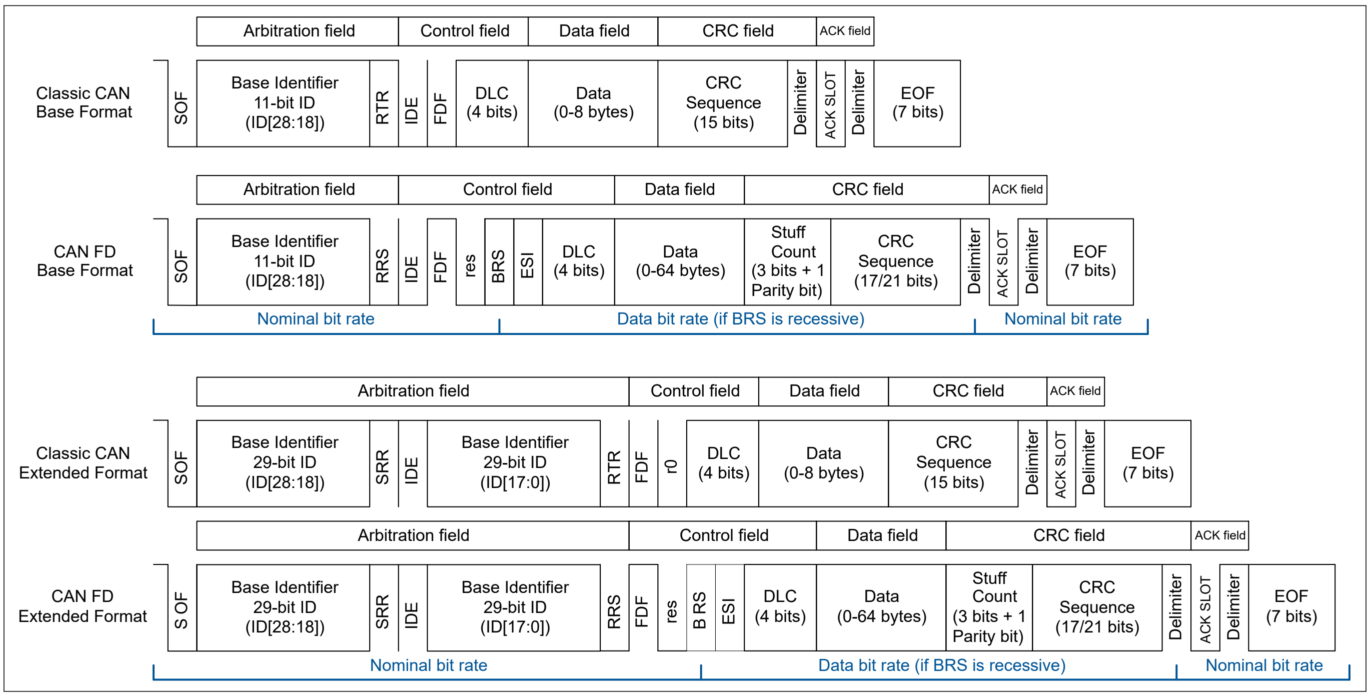

Figure 2 **DATA FRAME formats**

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 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.

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 fixed length of 15 bits. CAN FD frames additionally consist of a 4-bit Stuff Count at the beginning of 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 therefore, 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 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.

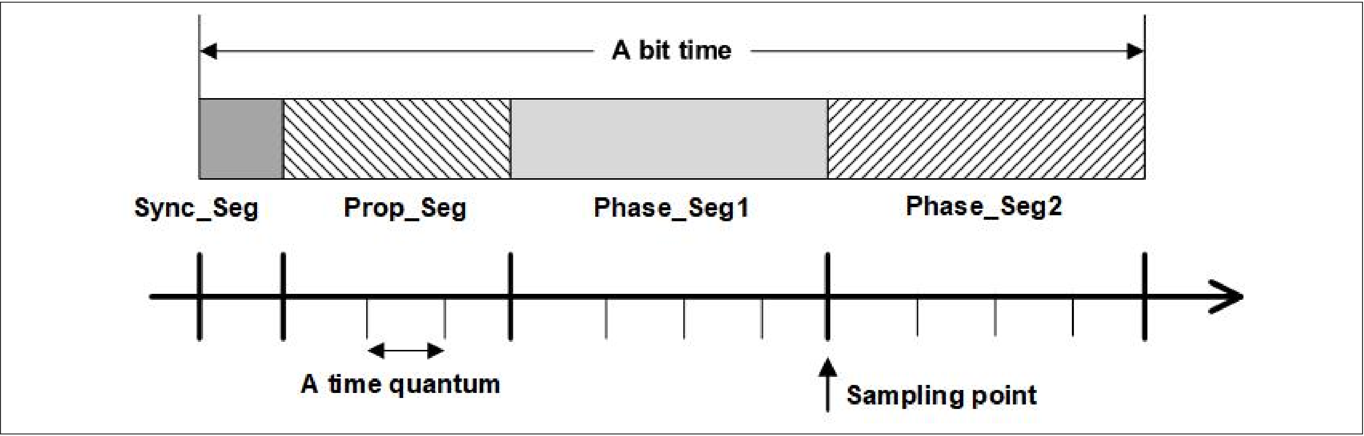

Figure 3 Bit time construction

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

(table continues...)

Table 2 (continued) CAN bit timing parameters

Parameter	Description
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

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×tq

=1 + NBTP.NTSEG17:0+1+NBTP.NTSEG26:0+1×NBTP.NBRP8:0+1×CAN clock period

Example (500 kbps with sampling point of 75%)

=1+13+1+4+1×3+1×140000000=0.000002 500 kbps

Data bit time

=1 + DBTP.DTSEG14:0+1+DBTP.DTSEG23:0+1×DBTP.DBRP4:0+1×CAN clock period

Example (5 Mbps with sampling point of 62.5%)

=1+3+1+2+1×0+1×140000000=0.0000002 5 Mbps

Example (2 Mbps with sampling point of 60%)

=1+10+1+7+1×0+1×140000000=0.0000005 2 Mbps

3 CAN FD controller in TRAVEO™ T2G family

This section provides an overview of the CAN FD controller in TRAVEO™ T2G family.

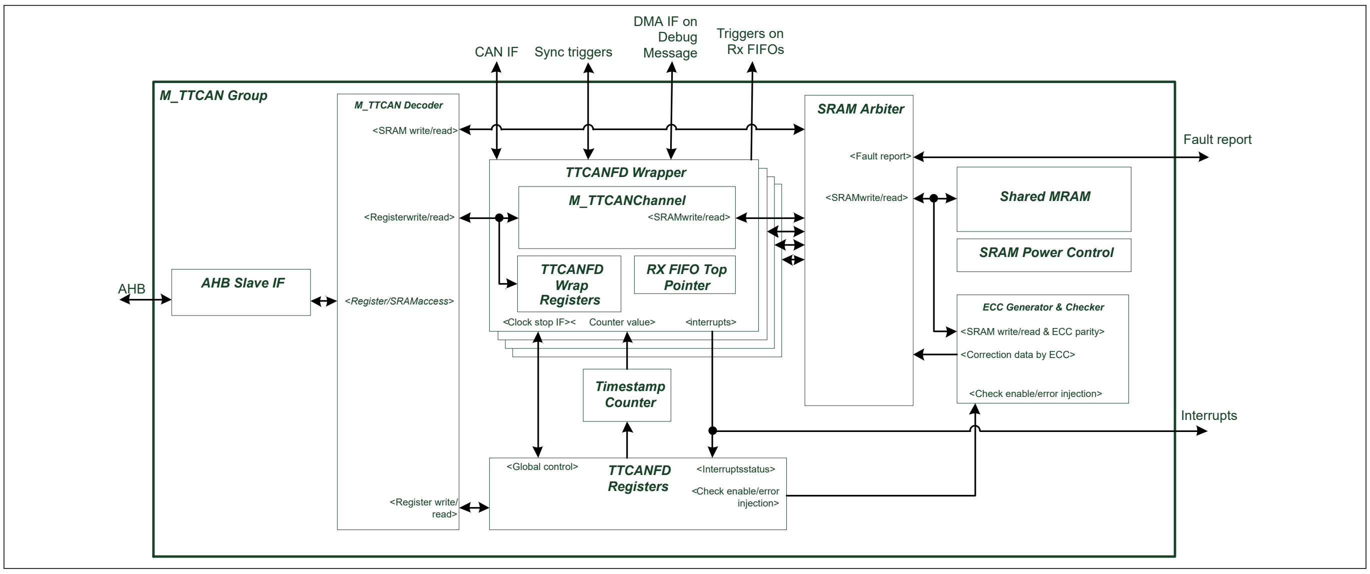

Figure 4 **CAN FD controller block diagram**

Figure 4 shows the block diagram of the CAN FD controller (M_TTCAN) in TRAVEO™ T2G devices. The M_TTCAN channels in TRAVEO™ T2G 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.

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.FnA) 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 Sample Driver Library (SDL) provided by Infineon. The code snippets in this application note are part of SDL. See Other references for the SDL.

SDL basically has a configuration part and a driver part. The configuration part mainly configures the parameter values for the desired operation. The driver part configures each register based on the parameter values in the configuration part.

You can configure the configuration part according to your system.

4.1 CAN FD setup

Do the following to set up CAN FD:

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, and (1) to (9) are performed in the driver part.

(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.

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 architectureTRM.

(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.*

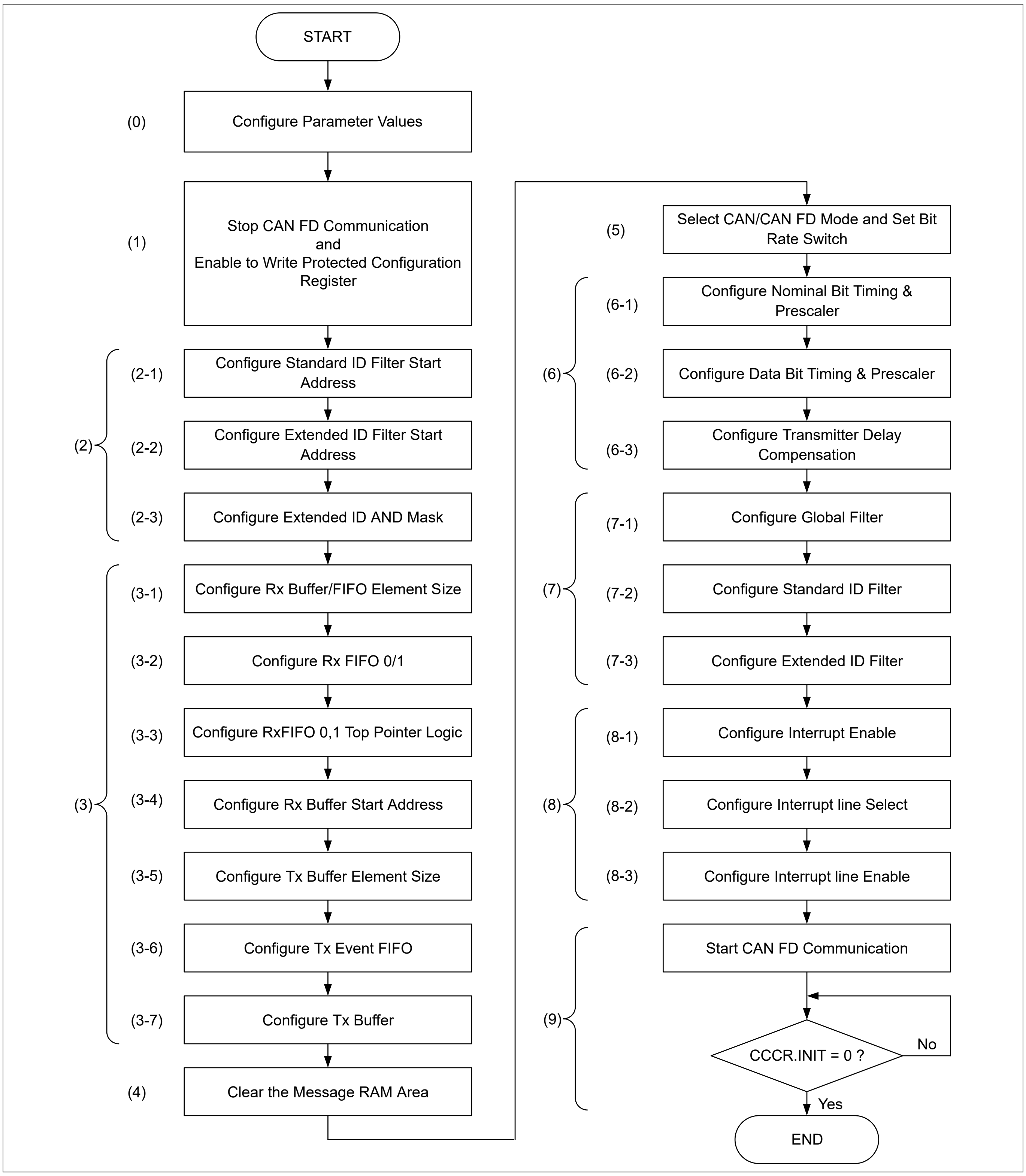
START
(0) Configure Parameter Values
(1) Stop CAN FD Communication and Enable to Write Protected Configuration Register
(2)
(2-1) Configure Standard ID Filter Start Address
(2-2) Configure Extended ID Filter Start Address
(2-3) Configure Extended ID AND Mask
(3)
(3-1) Configure Rx Buffer/FIFO Element Size
(3-2) Configure Rx FIFO 0/1
(3-3) Configure RxFIFO 0,1 Top Pointer Logic
(3-4) Configure Rx Buffer Start Address
(3-5) Configure Tx Buffer Element Size
(3-6) Configure Tx Event FIFO
(3-7) Configure Tx Buffer
(4) Clear the Message RAM Area
(5) Select CAN/CAN FD Mode and Set Bit Rate Switch
(6)
(6-1) Configure Nominal Bit Timing & Prescaler
(6-2) Configure Data Bit Timing & Prescaler
(6-3) Configure Transmitter Delay Compensation
(7)
(7-1) Configure Global Filter
(7-2) Configure Standard ID Filter
(7-3) Configure Extended ID Filter
(8)
(8-1) Configure Interrupt Enable
(8-2) Configure Interrupt line Select
(8-3) Configure Interrupt line Enable
(9)
Start CAN FD Communication
CCCR.INIT = 0 ?
No
Yes
END

Figure 5 **Example of CAN FD initialization flow**

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 SDL.

Use case:

- Mode: CAN FD
- CAN Instance: CAN0_CH0
- 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 Tx event FIFO/Buffer: 4

4.2.2 Configuration for CAN FD controller

Table 3 lists the parameters of the configuration part in SDL for CAN FD initialization.

Table 3 List of CAN FD initialization parameters

Output pin		Description	Initial setting
Can_Cfg.txCallback		Set Interrupt handler address for each event. No handling, if set to NULL.	NULL
Can_Cfg.rxCallback			CAN_RxMsgCallback
Can_Cfg.rxFifoWithTopCallback			NULL
Can_Cfg.statusCallback			NULL
Can_Cfg.errorCallback			NULL
Can_Cfg.canFDMode		Select configure mode True: CAN FD mode, False: Classic CAN mode	true
Can_Cfg.bitrate		Normal bit rate setting.	–
	.prescaler	Set value by which the oscillator frequency is divided for generating the bit time quanta. The setting value is actual value minus 1.	10u–1u
	.timeSegment1	Set normal time segment 1. The setting value is actual value minus 1.	5u–1u
	.timeSegment2	Set normal time segment 2. The setting value is actual value minus 1.	2u–1u
	.syncJumpWidth	Set normal (Re)Synchronization Jump Width. The setting value is actual value minus 1.	2u–1u
Can_Cfg.fastBitrate		Fast bit rate setting. It should be needed if you set CAN FD mode.	–

(table continues...)

Table 3 (continued) List of CAN FD initialization parameters

Output pin		Description	Initial setting
	.prescaler	Set value by which the oscillator frequency is divided for generating the bit time quanta. The setting value is actual value minus 1.	5u–1u
	.timeSegment1	Set time segment 1. The setting value is actual value minus 1.	5u–1u
	.timeSegment2	Set time segment 2. The setting value is actual value minus 1.	2u–1u
	.syncJumpWidth	Set (Re)Synchronization Jump Width. The setting value is actual value minus 1.	2u–1u
Can_Cfg.tdcConfig		Transmitter Delay Compensation setting. It should be needed if you set CAN FD mode.	–
	.tdcEnabled	Set transmitter delay compensation enable. True: Enable, False: Disable	false
	.tdcOffset	Set transmitter delay compensation offset.	0
	.tdcFilterWindow	Set transmitter delay compensation filter window length.	0
Can_Cfg.sidFilterConfig		Set standard message ID filters	–
	.numberOfSIDFilters	Number of standard message ID filters	sizeof(stdIdFilter) / sizeof(stdIdFilter[0])
	.sidFilter	Set standard message ID filter address	stdIdFilter
Can_Cfg.extidFilterConfig		Set extended message ID filters	–
	.numberOfEXTIDFilters	Number of extended message ID filters	sizeof(extIdFilter) / sizeof(extIdFilter[0])
	.extidFilter	Set extended message ID filter address	extIdFilter
	.extIDANDMask	Set value to be ANDed with the Message ID of a received frame for acceptance filtering of extended frames.	0x1fffffff
Can_Cfg.globalFilterConfig		Global Filter Setting	–
	.nonMatchingFramesStandard	Defines how received messages that do not match any element of the filter list are treated. Accept in Rx FIFO 0, Accept in Rx FIFO 1, Reject	CY_CANFD_ACCEPT_IN_RXFIFO_0
	.nonMatchingFramesExtended		CY_CANFD_ACCEPT_IN_RXFIFO_1
	.rejectRemoteFramesStandard	Reject Remote Frames True: Reject all remote frames, False: Filter remote frames	true
	.rejectRemoteFramesExtended		true
Can_Cfg.rxBufferDataSize		Set Rx event FIFO size.	CY_CANFD_BUFFER_DATA_SIZE_64

(table continues...)

Table 3 (continued) List of CAN FD initialization parameters

Output pin		Description	Initial setting
Can_Cfg.rxFifo1DataSize		Set Rx data buffer size	CY_CANFD_BUFFER_DATA_SIZE_64
Can_Cfg.rxFifo0DataSize		Set Rx FIFO1 size	CY_CANFD_BUFFER_DATA_SIZE_64
Can_Cfg.txBufferDataSize		Set Tx buffer data size	CY_CANFD_BUFFER_DATA_SIZE_64
Can_Cfg.rxFifo0Config		Configure Rx FIFO 0	–
	.mode	FIFO 0 Operation Mode Blocking mode, Overwrite mode	CY_CANFD_FIFO_MODE_BLOCKING
	.watermark	Set level for Rx FIFO 0 watermark interrupt	10u
	.numberOfFifoElements	Set number of Rx FIFO 0 elements	8u
	.topPointerLogicEnabled	It enables the FIFO top pointer logic to set the FIFO Top Address and message word counter. True: Enable, False: Disable	false
Can_Cfg.rxFifo1Config		Configure Rx FIFO 0	–
	.mode	FIFO 1 Operation Mode Blocking mode, Overwrite mode	CY_CANFD_FIFO_MODE_BLOCKING
	.watermark	Set level for Rx FIFO 1 watermark interrupt	10u
	.numberOfFifoElements	Set number of Rx FIFO 1 elements	8u
	.topPointerLogicEnabled	It enables the FIFO top pointer logic to set the FIFO Top Address and message word counter. True: Enable, False: Disable	false
Can_Cfg.noOfRxBuffers		Set number of Tx event FIFO	4u
Can_Cfg.noOfTxBuffers		Set number of dedicated Tx buffers	4u

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

Code Listing 1 Example to initialize CAN FD in configuration part

```
/* Standard ID Filter configuration */
static const cy_stc_id_filter_t stdIdFilter[] =
{
    /*  Standard ID filter.  */
    CANFD_CONFIG_STD_ID_FILTER_CLASSIC_RXBUFF(0x010u, 0u),    /* ID=0x010, store into RX buffer
Idx0 */
    CANFD_CONFIG_STD_ID_FILTER_CLASSIC_RXBUFF(0x020u, 1u),    /* ID=0x020, store into RX buffer
Idx1 */
};

/* Extended ID Filter configration */
static const cy_stc_extid_filter_t extIdFilter[] =
{
    /*  Extended ID filter.  */
    CANFD_CONFIG_EXT_ID_FILTER_CLASSIC_RXBUFF(0x10010u, 2u),  /* ID=0x10010, store into RX
buffer Idx2 */
    CANFD_CONFIG_EXT_ID_FILTER_CLASSIC_RXBUFF(0x10020u, 3u),  /* ID=0x10020, store into RX
buffer Idx3 */
};

/* CAN configuration */
/*  Configure interrupt handler for each event. Registers CAN message reception event. Others
are NULL  */
cy_stc_canfd_config_t canCfg =
{
    .txCallback     = NULL,                                  // Unused.
    .rxCallback     = CAN_RxMsgCallback,
    .rxFifoWithTopCallback = NULL,                           //CAN_RxFifoWithTopCallback,
    .statusCallback = NULL,                                  // Un-supported now
    .errorCallback  = NULL,                                  // Un-supported now

    .canFDMode      = true,                                  // Use CANFD mode
    // 40 MHz
    .bitrate        =                                        // Nominal bit rate settings
(sampling point = 75%)
    {
         /*  Normal bit rate setting. Prescaler = 10, tseg1 = 4, tseg2 = 1, sjw = 1. Set to
minus 1.   */
        .prescaler      = 10u - 1u,                          // cclk/10, When using 500kbps,
1bit = 8tq
        .timeSegment1   = 5u - 1u,                           // tseg1 = 5tq
        .timeSegment2   = 2u - 1u,                           // tseg2 = 2tq
        .syncJumpWidth  = 2u - 1u,                           // sjw   = 2tq
    },

    .fastBitrate    =                                        // Fast bit rate settings (sampling
point = 75%)
    {
         /*  Fast bit rate setting. Prescaler = 5, tseg1 = 4, tseg2 = 1, sjw = 1. Set to minus
1.   */
        .prescaler      = 5u - 1u,                           // cclk/5, When using 1Mbps, 1bit =
8tq
        .timeSegment1   = 5u - 1u,                          // tseg1 = 5tq,
        .timeSegment2   = 2u - 1u,                          // tseg2 = 2tq
        .syncJumpWidth  = 2u - 1u,                          // sjw   = 2tq
    },
    .tdcConfig      =                                       // Transceiver delay compensation,
unused.
    {
         /*  Configure Transmitter Delay Compensation. Set tdcEnabled to false (this is
unused).  */
        .tdcEnabled     = false,
        .tdcOffset      = 0,
        .tdcFilterWindow= 0,
    },
    .sidFilterConfig    =                                   // Standard message ID filters
setting.
    {
        .numberOfSIDFilters = sizeof(stdIdFilter) / sizeof(stdIdFilter[0]),
        .sidFilter          = stdIdFilter,
    },
    .extidFilterConfig  =                                   // Extended message ID filters
setting.
    {
        .numberOfEXTIDFilters   = sizeof(extIdFilter) / sizeof(extIdFilter[0]),
        .extidFilter            = extIdFilter,
        .extIDANDMask           = 0x1fffffff,               // No pre filtering.
    },
    .globalFilterConfig =                                   // Global filter setting.
    {
        .nonMatchingFramesStandard = CY_CANFD_ACCEPT_IN_RXFIFO_0,  // Reject none match IDs
        .nonMatchingFramesExtended = CY_CANFD_ACCEPT_IN_RXFIFO_1,  // Reject none match IDs
        .rejectRemoteFramesStandard = true,                        // No remote frame
        .rejectRemoteFramesExtended = true,                        // No remote frame
    },
     /*  Configure FIFO and data buffer size: 64 bytes.  */
    .rxBufferDataSize = CY_CANFD_BUFFER_DATA_SIZE_64,
    .rxFifo1DataSize  = CY_CANFD_BUFFER_DATA_SIZE_64,
    .rxFifo0DataSize  = CY_CANFD_BUFFER_DATA_SIZE_64,
    .txBufferDataSize = CY_CANFD_BUFFER_DATA_SIZE_64,
    .rxFifo0Config    = // RX FIFO0, unused.
    {
    /*  Configure Rx FIFO 0. Set topPointerLogicEnabled to false (RxFIFO 0 is unused).  */
        .mode = CY_CANFD_FIFO_MODE_BLOCKING,
        .watermark = 10u,
        .numberOfFifoElements = 8u,
        .topPointerLogicEnabled = false,
    },
    .rxFifo1Config    =                                      // RX FIFO1, unused.
    {
         /*  Configure Rx FIFO 1.  */
        .mode = CY_CANFD_FIFO_MODE_BLOCKING,
        .watermark = 10u,
        .numberOfFifoElements = 8u,
        .topPointerLogicEnabled = false,                    // true,
    },
    .noOfRxBuffers  = 4u,
    .noOfTxBuffers  = 4u,         /*  Configure Tx event FIFO and Buffer.
    Set to 4.
    */
};

int main(void)
{
  :
  /*  CAN Channel setting. CAN0 Channel 0.  */
Cy_CANFD_Init(CY_CANFD_TYPE, &canCfg);
  :
}
```

4.2.3 Configuration for Message RAM

This section shows that the configuration of CAN Message RAM and the overall message RAM size can be different for each TRAVEO™ device. You need to specify the size allocated per channel under each CAN macro. As part of SDL, a configuration is provided as an example, which can be modified based on your requirement for the respective application.

CYT2B7 has a 24-KB Message RAM per CAN macro. Code Listing 2 shows an example configuration which allocates 8 KB per channel. You can change this code to allocate 10 KB + 10 KB + 4 KB.

Code Listing 2 Example configuration of Message RAM

```
/** Offset of CAN FD Message RAM (MRAM). Total 24k MRAM per CAN FD instance is shared by all
CAN FD channels
 *  within an instance and allocation for each channel is done by user. Below shown is example
allocation */

/** Defining MRAM size (in bytes) per channel for CAN0 */
#define CY_CANFD0_0_MSGRAM_SIZE         ((CANFD0_MRAM_SIZE*1024)/CANFD0_CAN_NR)
#define CY_CANFD0_1_MSGRAM_SIZE         ((CANFD0_MRAM_SIZE*1024)/CANFD0_CAN_NR)
#define CY_CANFD0_2_MSGRAM_SIZE         ((CANFD0_MRAM_SIZE*1024)/CANFD0_CAN_NR)
```

4.2.4 Example code to initialize CAN FD in driver part

Code Listing 3 demonstrates an example program to initialize CAN FD in the driver part.

The following description will help you understand the register notation of the driver part of SDL:

- pstcCanFDChMTTCAN->unCCCR.u32Register is the CANFDx_CHy_CCCR register mentioned in the registers TRM. Other registers are also described in the same manner. “x” signifies the CAN FD instance and “y” signifies the channel number of CAN FD instance.
- Performance improvement measures

For register setting performance improvement, the SDL writes a complete 32-bit data to the register. Each bit field is generated in advance in a bit writable buffer and written to the register as the final 32-bit data.

```
unRXESC.stcField.u3RBDS = pstcConfig->rxBufferDataSize;
unRXESC.stcField.u3F1DS = pstcConfig->rxFifo1DataSize;  /* 1. Generate 32-bit data on the 
buffer. */
unRXESC.stcField.u3F0DS = pstcConfig->rxFifo0DataSize;
pstcCanFDChMTTCAN->unRXESC.u32Register = unRXESC.u32Register;  /* 2. Write to register as 
complete 32-bit data. */
```

See cyip_canfd.h under hdr/rev_x for more information on the union and structure representation of registers.

Code Listing 3 Example to initialize CAN FD in driver part

```
cy_en_canfd_status_t Cy_CANFD_Init(cy_pstc_canfd_type_t pstcCanFD, const cy_stc_canfd_config_t*
pstcConfig)
{
// Local variable declarations
cy_stc_canfd_context_t* pstcCanFDContext;
uint32_t*               pu32Adrs;
uint32_t                u32Count;
uint32_t                u32SizeInWord;

cy_stc_id_filter_t*     pstcSIDFilter;
cy_stc_extid_filter_t*  pstcEXTIDFilter;
cy_stc_canfd_msgram_config_t stcMsgramConfig;
volatile stc_CANFD_CH_M_TTCAN_t* pstcCanFDChMTTCAN;

/* Shadow data to avoid RMW and speed up HW access */
/*  Set data on the buffer to ‘0’ for performance improvement.  */
un_CANFD_CH_SIDFC_t unSIDFC = { 0 };
un_CANFD_CH_XIDFC_t unXIDFC = { 0 };
un_CANFD_CH_XIDAM_t unXIDAM = { 0 };
un_CANFD_CH_RXF0C_t unRXF0C = { 0 };
un_CANFD_CH_RXF1C_t unRXF1C = { 0 };
un_CANFD_CH_RXBC_t  unRXBC  = { 0 };
un_CANFD_CH_TXEFC_t unTXEFC = { 0 };
un_CANFD_CH_TXBC_t  unTXBC  = { 0 };
un_CANFD_CH_CCCR_t  unCCCR  = { 0 };
un_CANFD_CH_NBTP_t  unNBTP  = { 0 };
un_CANFD_CH_DBTP_t  unDBTP  = { 0 };
un_CANFD_CH_TDCR_t  unTDCR  = { 0 };
un_CANFD_CH_GFC_t   unGFC   = { 0 };
un_CANFD_CH_RXESC_t unRXESC = { 0 };
un_CANFD_CH_TXESC_t unTXESC = { 0 };
un_CANFD_CH_IE_t    unIE    = { 0 };
un_CANFD_CH_ILS_t   unILS   = { 0 };
un_CANFD_CH_ILE_t   unILE   = { 0 };
un_CANFD_CH_RXFTOP_CTL_t unRXFTOP_CTL = { 0 };

/* Check for NULL pointers */
if ( pstcCanFD  == NULL ||
     pstcConfig == NULL ||
     /*  Check if configuration parameter values are valid.  */
     ((pstcConfig->sidFilterConfig.numberOfSIDFilters != 0) && (pstcConfig-
>sidFilterConfig.sidFilter == NULL)) ||((pstcConfig->extidFilterConfig.numberOfEXTIDFilters !=
0) && (pstcConfig->extidFilterConfig.extidFilter == NULL))
      )
   {
return CY_CANFD_BAD_PARAM;
   }


/* Get pointer to internal data structure */
pstcCanFDContext = Cy_CANFD_GetContext(pstcCanFD);
```

```
/* Check for NULL */
if (pstcCanFDContext == NULL)
  {
return CY_CANFD_BAD_PARAM;
  }
/* Set notification callback functions */
/*  Configure interrupt handler for callback events.  */
pstcCanFDContext->canFDInterruptHandling.canFDTxInterruptFunction  = pstcConfig->txCallback;
pstcCanFDContext->canFDInterruptHandling.canFDRxInterruptFunction  = pstcConfig->rxCallback;
pstcCanFDContext->canFDInterruptHandling.canFDRxWithTopPtrInterruptFunction = pstcConfig-
>rxFifoWithTopCallback;
pstcCanFDContext->canFDNotificationCb.canFDStatusInterruptFunction = pstcConfig->statusCallback;
pstcCanFDContext->canFDNotificationCb.canFDErrorInterruptFunction  = pstcConfig->errorCallback;

/* Get the pointer to M_TTCAN of the CAN FD channel */
/*  Get base address of CANx channel register.  */
pstcCanFDChMTTCAN = &pstcCanFD->M_TTCAN;

/* Set CCCR.INIT to 1 and wait until it will be updated. */
pstcCanFDChMTTCAN->unCCCR.u32Register = 0x1ul;
while(pstcCanFDChMTTCAN->unCCCR.stcField.u1INIT != 1)
    {
    }

    /* Cancel protection by setting CCE */
    /*  (1) Stop CAN FD communication and enable to write to Protected Configuration Register.
*/
    pstcCanFDChMTTCAN->unCCCR.u32Register = 0x3ul;


    /* Standard ID filter */
    /*  (2-1) Configure Standard ID filter.  */
    unSIDFC.stcField.u8LSS = pstcConfig->sidFilterConfig.numberOfSIDFilters;  // Number of SID
filters
    unSIDFC.stcField.u14FLSSA = stcMsgramConfig.offset >> 2;    // Start address (word) of SID
filter configuration in message RAM
    pstcCanFDChMTTCAN->unSIDFC.u32Register = unSIDFC.u32Register;

    /* Extended ID filter */
    unXIDFC.stcField.u7LSE = pstcConfig->extidFilterConfig.numberOfEXTIDFilters; // Number of
ext id filters
    unXIDFC.stcField.u14FLESA = pstcCanFDChMTTCAN->unSIDFC.stcField.u14FLSSA +
      (pstcConfig->sidFilterConfig.numberOfSIDFilters * SIZE_OF_SID_FILTER_IN_WORD); // Start
address (word) of ext id filter configuration in message RAM
      /*  (2-2) Configure Extended ID Filter.Start address is placed under Standard ID filter
area.  */
    pstcCanFDChMTTCAN->unXIDFC.u32Register = unXIDFC.u32Register;

    /* Extended ID AND Mask */
    unXIDAM.stcField.u29EIDM = pstcConfig->extidFilterConfig.extIDANDMask;
    /*  (2-3) Configure Extended ID Mask.  */
    pstcCanFDChMTTCAN->unXIDAM.u32Register = unXIDAM.u32Register;
    /* Configuration of Rx Buffer and Rx FIFO */
    unRXESC.stcField.u3RBDS = pstcConfig->rxBufferDataSize;
    unRXESC.stcField.u3F1DS = pstcConfig->rxFifo1DataSize;
    unRXESC.stcField.u3F0DS = pstcConfig->rxFifo0DataSize;
    /*  (3-1) Configure Rx Buffer/FIFO Element size.  */
    pstcCanFDChMTTCAN->unRXESC.u32Register = unRXESC.u32Register;

    /* Rx FIFO 0 */
    unRXF0C.stcField.u1F0OM = pstcConfig->rxFifo0Config.mode;
    unRXF0C.stcField.u7F0WM = pstcConfig->rxFifo0Config.watermark;
    unRXF0C.stcField.u7F0S = pstcConfig->rxFifo0Config.numberOfFifoElements;
    unRXF0C.stcField.u14F0SA = pstcCanFDChMTTCAN->unXIDFC.stcField.u14FLESA +
    /*  (3-2) Configure Rx FIFO 0. Start address is placed under Extend ID filter area.  */
      (pstcConfig->extidFilterConfig.numberOfEXTIDFilters * SIZE_OF_EXTID_FILTER_IN_WORD);
    pstcCanFDChMTTCAN->unRXF0C.u32Register = unRXF0C.u32Register;

    /* Rx FIFO 1 */
    unRXF1C.stcField.u1F1OM = pstcConfig->rxFifo1Config.mode;
    unRXF1C.stcField.u7F1WM = pstcConfig->rxFifo1Config.watermark;
    unRXF1C.stcField.u7F1S = pstcConfig->rxFifo1Config.numberOfFifoElements;
    unRXF1C.stcField.u14F1SA = pstcCanFDChMTTCAN->unRXF0C.stcField.u14F0SA +
    /*  (3-2) Configure Rx FIFO Start address is placed under Rx FIFO 0 area.  */
      (pstcConfig->rxFifo0Config.numberOfFifoElements * (2 + dataBufferSizeInWord[pstcConfig-
>rxFifo0DataSize]));
    pstcCanFDChMTTCAN->unRXF1C.u32Register = unRXF1C.u32Register;

    /* Rx FIFO 0,1 Top pointer logic config */
    /*  (3-3) Configure RxFIFO 0,1 Top pointer logic.  */
    unRXFTOP_CTL.stcField.u1F0TPE = (pstcConfig->rxFifo0Config.topPointerLogicEnabled == false)
? 0 : 1;
    unRXFTOP_CTL.stcField.u1F1TPE = (pstcConfig->rxFifo1Config.topPointerLogicEnabled == false)
? 0 : 1;
    pstcCanFD->unRXFTOP_CTL.u32Register = unRXFTOP_CTL.u32Register;

    /* Rx buffer */
    /*  (3-4) Configure Rx Buffer. Start address is placed under Rx FIFO 1 area.  */
    unRXBC.stcField.u14RBSA = pstcCanFDChMTTCAN->unRXF1C.stcField.u14F1SA +
      (pstcConfig->rxFifo1Config.numberOfFifoElements * (2 + dataBufferSizeInWord[pstcConfig-
>rxFifo1DataSize]));
    pstcCanFDChMTTCAN->unRXBC.u32Register = unRXBC.u32Register;

    /* Configuration of Tx Buffer and Tx FIFO/Queue */
    unTXESC.stcField.u3TBDS = pstcConfig->txBufferDataSize;
    /*  (3-5) Configure Tx Buffer Element Size.  */
    pstcCanFDChMTTCAN->unTXESC.u32Register = unTXESC.u32Register;

    /* Tx FIFO/QUEUE (not use) */
    unTXEFC.stcField.u6EFWM = 0;  /* Watermark interrupt disabled */
    unTXEFC.stcField.u6EFS = 0;   /* Tx Event FIFO disabled */
    /*  (3-6) Configure Tx Event FIFO. Start address is placed under Tx Buffer area.  */
    unTXEFC.stcField.u14EFSA = pstcCanFDChMTTCAN->unRXBC.stcField.u14RBSA +
      (pstcConfig->noOfRxBuffers * (2 + dataBufferSizeInWord[pstcConfig->rxBufferDataSize]));
    pstcCanFDChMTTCAN->unTXEFC.u32Register = unTXEFC.u32Register;

    /* Tx buffer */
    unTXBC.stcField.u1TFQM = 0;   /* Tx FIFO operation */
    unTXBC.stcField.u6TFQS = 0;   /* No Tx FIFO/Queue */
    unTXBC.stcField.u6NDTB = pstcConfig->noOfTxBuffers;   /* Number of Dedicated Tx Buffers */
    /*  (3-7) Configure Tx Buffer. Start address is placed under Tx FIFO area.  */
    unTXBC.stcField.u14TBSA = pstcCanFDChMTTCAN->unTXEFC.stcField.u14EFSA +
      (10 * SIZE_OF_TXEVENT_FIFO_IN_WORD);    // Reserving memory for 10 TxEvent Fifo elements
for easy future use
    pstcCanFDChMTTCAN->unTXBC.u32Register = unTXBC.u32Register;

    /* Initialize message RAM area(Entire region zeroing) */
    pu32Adrs = (uint32_t *)(((uint32_t)pstcCanFD & 0xFFFF0000ul) +
(uint32_t)CY_CANFD_MSGRAM_START + stcMsgramConfig.offset);
    u32SizeInWord = stcMsgramConfig.size >> 2;   /*  (4) Clear the Message RAM area.  */
    for(u32Count = 0; u32Count < u32SizeInWord; u32Count++)
    {
        *pu32Adrs++ = 0ul;
    }

    /* Configuration of CAN bus */
    /* CCCR register */
    unCCCR.stcField.u1TXP = 0;     /* Transmit pause disabled */
    unCCCR.stcField.u1BRSE = ((pstcConfig->canFDMode == true) ? 1 : 0);    /* Bit rate switch */
    unCCCR.stcField.u1FDOE = ((pstcConfig->canFDMode == true) ? 1 : 0);    /* FD operation */
    unCCCR.stcField.u1TEST = 0;    /* Normal operation */
    unCCCR.stcField.u1DAR = 0;     /* Automatic retransmission enabled */
    unCCCR.stcField.u1MON_ = 0;    /* Bus Monitoring Mode is disabled */
    unCCCR.stcField.u1CSR = 0;     /* No clock stop is requested */
    unCCCR.stcField.u1ASM = 0;     /* Normal CAN operation. */
    /*  (5) Select CAN/CAN FD Mode and Set Bit Rate Switch.  */
    pstcCanFDChMTTCAN->unCCCR.u32Register = unCCCR.u32Register;

    /* Nominal Bit Timing & Prescaler Register */
    unNBTP.stcField.u9NBRP = pstcConfig->bitrate.prescaler;
    unNBTP.stcField.u8NTSEG1 = pstcConfig->bitrate.timeSegment1;
    unNBTP.stcField.u7NTSEG2 = pstcConfig->bitrate.timeSegment2;
    unNBTP.stcField.u7NSJW = pstcConfig->bitrate.syncJumpWidth;
    /* (6-1) Configure Nominal Bit Timing & Prescaler. */
    pstcCanFDChMTTCAN->unNBTP.u32Register = unNBTP.u32Register;


    if(pstcConfig->canFDMode == true)
    {
        /* Data Bit Timing & Prescaler */
        unDBTP.stcField.u5DBRP = pstcConfig->fastBitrate.prescaler;
        unDBTP.stcField.u5DTSEG1 = pstcConfig->fastBitrate.timeSegment1;
        /*  (6-2) Configure Data Bit Timing & Prescaler.This configuration is only for CAN FD
mode.  */
        unDBTP.stcField.u4DTSEG2 = pstcConfig->fastBitrate.timeSegment2;
        unDBTP.stcField.u4DSJW = pstcConfig->fastBitrate.syncJumpWidth;
        unDBTP.stcField.u1TDC = ((pstcConfig->tdcConfig.tdcEnabled == true) ? 1 : 0); /*
Transceiver Delay Compensation enabled */
        pstcCanFDChMTTCAN->unDBTP.u32Register = unDBTP.u32Register;

        /* Transmitter Delay Compensation */
        /*  (6-3) Configure Transmitter Delay Compensation. This configuration is only for CAN
FD mode.  */
        unTDCR.stcField.u7TDCO = pstcConfig->tdcConfig.tdcOffset;       /* Transmitter Delay
Compensation Offset */
        unTDCR.stcField.u7TDCF = pstcConfig->tdcConfig.tdcFilterWindow; /* Transmitter Delay
Compensation Filter Window Length */
        pstcCanFDChMTTCAN->unTDCR.u32Register = unTDCR.u32Register;
    }

    /* Configuration of Global Filter */
    unGFC.stcField.u2ANFS = pstcConfig->globalFilterConfig.nonMatchingFramesStandard;
    /*  (7-1) Configure Global Filter.  */
    unGFC.stcField.u2ANFE = pstcConfig->globalFilterConfig.nonMatchingFramesExtended;
    unGFC.stcField.u1RRFS = ((pstcConfig->globalFilterConfig.rejectRemoteFramesStandard ==
true) ? 1 : 0);
    unGFC.stcField.u1RRFE = ((pstcConfig->globalFilterConfig.rejectRemoteFramesExtended ==
true) ? 1 : 0);
    pstcCanFDChMTTCAN->unGFC.u32Register = unGFC.u32Register;

    /* Standard Message ID Filters */
    /*  (7-2) Configure Standard ID Filter.  */
    pstcSIDFilter = (cy_stc_id_filter_t *)(((uint32_t)pstcCanFD & 0xFFFF0000ul) +
                                       (uint32_t)CY_CANFD_MSGRAM_START      +
                                       (pstcCanFDChMTTCAN->unSIDFC.stcField.u14FLSSA <<
2u));
    for(u32Count = 0; u32Count < pstcConfig->sidFilterConfig.numberOfSIDFilters; u32Count++)
    {
        pstcSIDFilter[u32Count] = pstcConfig->sidFilterConfig.sidFilter[u32Count];
    }

    /* Extended Message ID Filters */
    /*  (7-3) Configure Extended ID Filter.  */
    pstcEXTIDFilter = (cy_stc_extid_filter_t *)(((uint32_t)pstcCanFD & 0xFFFF0000ul) +
                                            (uint32_t)CY_CANFD_MSGRAM_START      +
                                            (pstcCanFDChMTTCAN->unXIDFC.stcField.u14FLESA
<< 2u));
    for(u32Count = 0; u32Count < pstcConfig->extidFilterConfig.numberOfEXTIDFilters; u32Count++)
    {
        pstcEXTIDFilter[u32Count] = pstcConfig->extidFilterConfig.extidFilter[u32Count];
    }

    /* Configuration of Interrupt */
    /* Interrupt Enable */
    unIE.stcField.u1ARAE  = 0; /* Access to Reserved Address */
    unIE.stcField.u1PEDE  = 0; /* Protocol Error in Data Phase */
    unIE.stcField.u1PEAE  = 0; /* Protocol Error in Arbitration Phase */
    unIE.stcField.u1WDIE  = 0; /* Watchdog */
    unIE.stcField.u1BOE   = 0; /* Bus_Off Status */
    unIE.stcField.u1EWE   = 0; /* Warning Status */
```

```
        unIE.stcField.u1EPE    = 0; /* Error Passive */
        unIE.stcField.u1ELOE   = 0; /* Error Logging Overflow */
        unIE.stcField.u1BEUE   = 0; /* Bit Error Uncorrected */
        unIE.stcField.u1BECE   = 0; /* Bit Error Corrected */
        unIE.stcField.u1DRXE   = 1; /* Message stored to Dedicated Rx Buffer */
        unIE.stcField.u1TOOE   = 0; /* Timeout Occurred */
        unIE.stcField.u1MRAFE  = 0; /* Message RAM Access Failure */
        unIE.stcField.u1TSWE   = 0; /* Timestamp Wraparound */
        unIE.stcField.u1TEFLE  = 0; /* Tx Event FIFO Event Lost */
        unIE.stcField.u1TEFFE  = 0; /* Tx Event FIFO Full */
        unIE.stcField.u1TEFWE  = 0; /* Tx Event FIFO Watermark Reached */
        unIE.stcField.u1TEFNE  = 0; /* Tx Event FIFO New Entry */
        unIE.stcField.u1TFEE   = 0; /* Tx FIFO Empty */
        unIE.stcField.u1TCFE   = 0; /* Transmission Cancellation Finished */
        unIE.stcField.u1TCE    = 0; /* Transmission Completed */
        unIE.stcField.u1HPME   = 0; /* High Priority Message */
        unIE.stcField.u1RF1LE  = 0; /* Rx FIFO 1 Message Lost */
        unIE.stcField.u1RF1FE  = 0; /* Rx FIFO 1 Full */
        unIE.stcField.u1RF1WE  = 0; /* Rx FIFO 1 Watermark Reached */
        unIE.stcField.u1RF1NE  = 1; /* Rx FIFO 1 New Message */
        unIE.stcField.u1RF0LE  = 0; /* Rx FIFO 0 Message Lost */
        unIE.stcField.u1RF0FE  = 0; /* Rx FIFO 0 Full */
        unIE.stcField.u1RF0WE  = 0; /* Rx FIFO 0 Watermark Reached */
        unIE.stcField.u1RF0NE  = 1; /* Rx FIFO 0 New Message */    /*  (8-1) Configure Interrupt
    Enable.  */
        pstcCanFDChMTTCAN->unIE.u32Register = unIE.u32Register;

        /* Interrupt Line Select */
        unILS.stcField.u1ARAL   = 0; /* Access to Reserved Address */
        unILS.stcField.u1PEDL   = 0; /* Protocol Error in Data Phase */
        unILS.stcField.u1PEAL   = 0; /* Protocol Error in Arbitration Phase */
        unILS.stcField.u1WDIL   = 0; /* Watchdog */
        unILS.stcField.u1BOL    = 0; /* Bus_Off Status */
        unILS.stcField.u1EWL    = 0; /* Warning Status */
        unILS.stcField.u1EPL    = 0; /* Error Passive */
        unILS.stcField.u1ELOL   = 0; /* Error Logging Overflow */
        unILS.stcField.u1BEUL   = 0; /* Bit Error Uncorrected */
        unILS.stcField.u1BECL   = 0; /* Bit Error Corrected */
        unILS.stcField.u1DRXL   = 0; /* Message stored to Dedicated Rx Buffer */
        unILS.stcField.u1TOOL   = 0; /* Timeout Occurred */
        unILS.stcField.u1MRAFL  = 0; /* Message RAM Access Failure */
        unILS.stcField.u1TSWL   = 0; /* Timestamp Wraparound */
        unILS.stcField.u1TEFLL  = 0; /* Tx Event FIFO Event Lost */
        unILS.stcField.u1TEFFL  = 0; /* Tx Event FIFO Full */
        unILS.stcField.u1TEFWL  = 0; /* Tx Event FIFO Watermark Reached */
        unILS.stcField.u1TEFNL  = 0; /* Tx Event FIFO New Entry */
        unILS.stcField.u1TFEL   = 0; /* Tx FIFO Empty */
        unILS.stcField.u1TCFL   = 0; /* Transmission Cancellation Finished */
        unILS.stcField.u1TCL    = 0; /* Transmission Completed */
        unILS.stcField.u1HPML   = 0; /* High Priority Message */
        unILS.stcField.u1RF1LL  = 0; /* Rx FIFO 1 Message Lost */
        unILS.stcField.u1RF1FL  = 0; /* Rx FIFO 1 Full */
        unILS.stcField.u1RF1WL  = 0; /* Rx FIFO 1 Watermark Reached */
        unILS.stcField.u1RF1NL = 0; /* Rx FIFO 1 New Message */
        unILS.stcField.u1RF0LL = 0; /* Rx FIFO 0 Message Lost */
        unILS.stcField.u1RF0FL = 0; /* Rx FIFO 0 Full */
        unILS.stcField.u1RF0WL = 0; /* Rx FIFO 0 Watermark Reached */
        unILS.stcField.u1RF0NL = 0; /* Rx FIFO 0 New Message */    /*  (8-2) Configure Interrupt
    line Select.  */
        pstcCanFDChMTTCAN->unILS.u32Register = unILS.u32Register;

        /* Interrupt Line Enable */
        unILE.stcField.u1EINT0 = 1;  /* Enable Interrupt Line 0 */    /*  (8-3) Configure Interrupt
    line Enable.  */
        unILE.stcField.u1EINT1 = 0;   /* Disable Interrupt Line 1 */
        pstcCanFDChMTTCAN->unILE.u32Register = unILE.u32Register;

        /* CAN-FD operation start */
        /* Set CCCR.INIT to 0 and wait until it will be updated */
        unCCCR.stcField.u1INIT = 0;   /*  (9) Start CAN FD communication.  */
        pstcCanFDChMTTCAN->unCCCR.u32Register = unCCCR.u32Register;
        while(pstcCanFDChMTTCAN->unCCCR.stcField.u1INIT != 0)
        {
        }

        return CY_CANFD_SUCCESS;
    }
```

4.3 Message transmission

Figure 6 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) are performed in the 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. A message transmission is started by writing to the Tx Buffer Add Request (TXBAR) register.

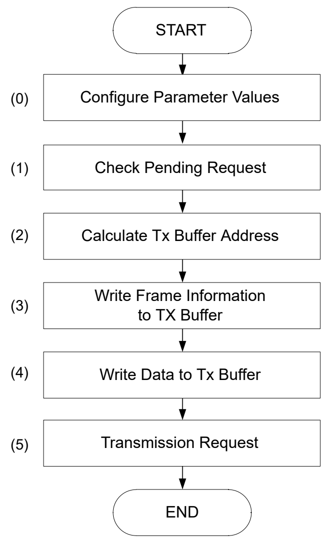
START
(0) Configure Parameter Values
(1) Check Pending Request
(2) Calculate Tx Buffer Address
(3) Write Frame Information to TX Buffer
(4) Write Data to Tx Buffer
(5) Transmission Request
END

Figure 6 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 Use case. CAN FD message transmission is configured using SDL.

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

Table 4 lists the parameters of the configuration part in SDL for message transmission.

Table 4 List of message transmission parameters

Parameters	Description	Value
.canFDFormat	Select CAN frame format. True: CAN FD, False: Classic CAN	true
.idConfig.extended	Select CAN identifier format. True: Extended identifier, False: Standard identifier	false
.idConfig.identifier	Set CAN identifier.	0x525
.dataConfig.dataLengthCode	Set CAN data length code	15
.dataConfig.data	Set transmission data	Arbitrary value

Code Listing 4 demonstrates an example program of CAN FD message transmission in the configuration part.

Code Listing 4 Example of message transmission in configuration part

```
int main(void)
{
   :
/* Prepare CANFD message to transmit*/
   cy_stc_canfd_msg_t stcMsg;

   stcMsg.canFDFormat = true;                /* CAN FD format */
   stcMsg.idConfig.extended = false;         /*  11-bit standard identifier  */
   stcMsg.idConfig.identifier = 0x525;       /*  CAN ID  */
   stcMsg.dataConfig.dataLengthCode = 15;    /*  CAN Data Length Code  */
   stcMsg.dataConfig.data[0]  = 0x70190523;  /*  Transmission data  */
   stcMsg.dataConfig.data[1]  = 0x70190819;  /*  Transmission data  */
   stcMsg.dataConfig.data[2]  = 0x33332222;  /*  Transmission data  */
   stcMsg.dataConfig.data[3]  = 0x33332222;  /*  Transmission data  */
   stcMsg.dataConfig.data[4]  = 0x55554444;  /*  Transmission data  */
   stcMsg.dataConfig.data[5]  = 0x77776666;  /*  Transmission data  */
   stcMsg.dataConfig.data[6]  = 0x99998888;  /*  Transmission data  */
   stcMsg.dataConfig.data[7]  = 0xBBBBAAAA;  /*  Transmission data  */
   stcMsg.dataConfig.data[8]  = 0xDDDDCCCC;  /*  Transmission data  */
   stcMsg.dataConfig.data[9]  = 0xFFFFEEEE;  /*  Transmission data  */
   stcMsg.dataConfig.data[10] = 0x78563412;
   stcMsg.dataConfig.data[11] = 0x00000000;
   stcMsg.dataConfig.data[12] = 0x11111111;
   stcMsg.dataConfig.data[13] = 0x22222222;
   stcMsg.dataConfig.data[14] = 0x33333333;
   stcMsg.dataConfig.data[15] = 0x44444444;

/*  CAN Transmission setting
CAN0 Channel 0
Message buffer 0
Transmission data
*/
Cy_CANFD_UpdateAndTransmitMsgBuffer(CY_CANFD_TYPE, 0, &stcMsg);
:
}
```

4.3.3 Example program of message transmission

Code Listing 5 demonstrates an example program of CAN FD message transmission in the driver part.

Code Listing 5 Example of message transmission in driver part

```
cy_en_canfd_status_t Cy_CANFD_UpdateAndTransmitMsgBuffer( cy_pstc_canfd_type_t pstcCanFD,
uint8_t u8MsgBuf, cy_stc_canfd_msg_t* pstcMsg )
{
    // Local variable declarations
    cy_stc_canfd_context_t* pstcCanFDContext;
    uint16_t u16DlcTemp;
    uint16_t u16Count;
    uint8_t u8DataLengthWord;

    cy_stc_canfd_tx_buffer_t* pstcCanFDTxBuffer;
    volatile stc_CANFD_CH_M_TTCAN_t* pstcCanFDChMTTCAN;

    /* Check for NULL pointers */
    /*  Check if configuration parameter values are valid  */
    if ( pstcCanFD  == NULL ||
         pstcMsg == NULL
       )
    {
        return CY_CANFD_BAD_PARAM;
    }

    if(u8MsgBuf > 31)
    {
        return CY_CANFD_BAD_PARAM;
    }

    /* Get pointer to internal data structure */
    pstcCanFDContext = Cy_CANFD_GetContext(pstcCanFD);

    /* Check for NULL */
    if (pstcCanFDContext == NULL)
    {
        return CY_CANFD_BAD_PARAM;
    }

    /* Get the pointer to M_TTCAN of the CAN FD channel */
    pstcCanFDChMTTCAN = &pstcCanFD->M_TTCAN;

    /* Check if CAN FD controller is in not in INIT state and Tx buffer is empty or not */
    if((pstcCanFDChMTTCAN->unCCCR.stcField.u1INIT != 0) ||
       ((pstcCanFDChMTTCAN->unTXBRP.u32Register & (1ul << u8MsgBuf)) != 0)   /* (1) Check
Pending Request */
      )
    {
        return CY_CANFD_BAD_PARAM;
    }

    /* Get Tx Buffer address */
    /* (2) Get the Tx Buffer Address with a calculation function.  (See Code Listing 5) */
    pstcCanFDTxBuffer = (cy_stc_canfd_tx_buffer_t*)Cy_CANFD_CalcTxBufAdrs(pstcCanFD, u8MsgBuf);
    if(pstcCanFDTxBuffer == NULL)
    {
        return CY_CANFD_BAD_PARAM;
    }

    pstcCanFDTxBuffer->t0_f.rtr = 0; /* Transmit data frame. */
    pstcCanFDTxBuffer->t0_f.xtd = (pstcMsg->idConfig.extended == true) ? 1 : 0;
    pstcCanFDTxBuffer->t0_f.id  = (pstcCanFDTxBuffer->t0_f.xtd == 0) ?
                                        (pstcMsg->idConfig.identifier << 18) : pstcMsg-
>idConfig.identifier;

    pstcCanFDTxBuffer->t1_f.efc = 0; /* Tx Event Fifo not used *//*(3) Write Frame Information
to TX Buffer */
    pstcCanFDTxBuffer->t1_f.mm  = 0; /* Not used */
    pstcCanFDTxBuffer->t1_f.dlc = pstcMsg->dataConfig.dataLengthCode;
    pstcCanFDTxBuffer->t1_f.fdf = (pstcMsg->canFDFormat == true) ? 1 : 0;
    pstcCanFDTxBuffer->t1_f.brs = (pstcMsg->canFDFormat == true) ? 1 : 0;

    /* Convert the DLC to data byte word */
    if (pstcMsg->dataConfig.dataLengthCode < 8 )
    {
        u16DlcTemp = 0;
    }
    else
    {
        u16DlcTemp = pstcMsg->dataConfig.dataLengthCode - 8;
    }
    u8DataLengthWord = dataBufferSizeInWord[u16DlcTemp];

    /* Data set */
    for ( u16Count = 0; u16Count < u8DataLengthWord; u16Count++ )  /*(4) Write Data to Tx
Buffer */
    {
        pstcCanFDTxBuffer->data_area_f[u16Count] = pstcMsg->dataConfig.data[u16Count];
    }

    /*  (5) Transmission Request  */
    pstcCanFDChMTTCAN->unTXBAR.u32Register = 1ul << u8MsgBuf;     // Transmit buffer add request

    return CY_CANFD_SUCCESS;
}
```

Code Listing 6 demonstrates an example program of Tx buffer address calculation in the driver part.

Code Listing 6 Example of Tx buffer address calculation in driver part

```
static uint32_t* Cy_CANFD_CalcTxBufAdrs(cy_pstc_canfd_type_t pstcCanFD, uint8_t u8MsgBuf)
{
    uint32_t* pu32Adrs;

    if ( u8MsgBuf > 31)
    {
        /* Set 0 to the return value if the index is invalid */
        pu32Adrs = NULL;
    }
    else
    {
        /* Set the message buffer address to the return value if the index is available */
        pu32Adrs = (uint32_t*)(((uint32_t)pstcCanFD & 0xFFFF0000ul) +
(uint32_t)CY_CANFD_MSGRAM_START);
        pu32Adrs += pstcCanFD->M_TTCAN.unTXBC.stcField.u14TBSA;
        /*  Calculate Tx Buffer Address  */
        pu32Adrs += u8MsgBuf * (2 + dataBufferSizeInWord[pstcCanFD-
>M_TTCAN.unTXESC.stcField.u3TBDS]);
    }
    return pu32Adrs;
}
```

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 7 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.

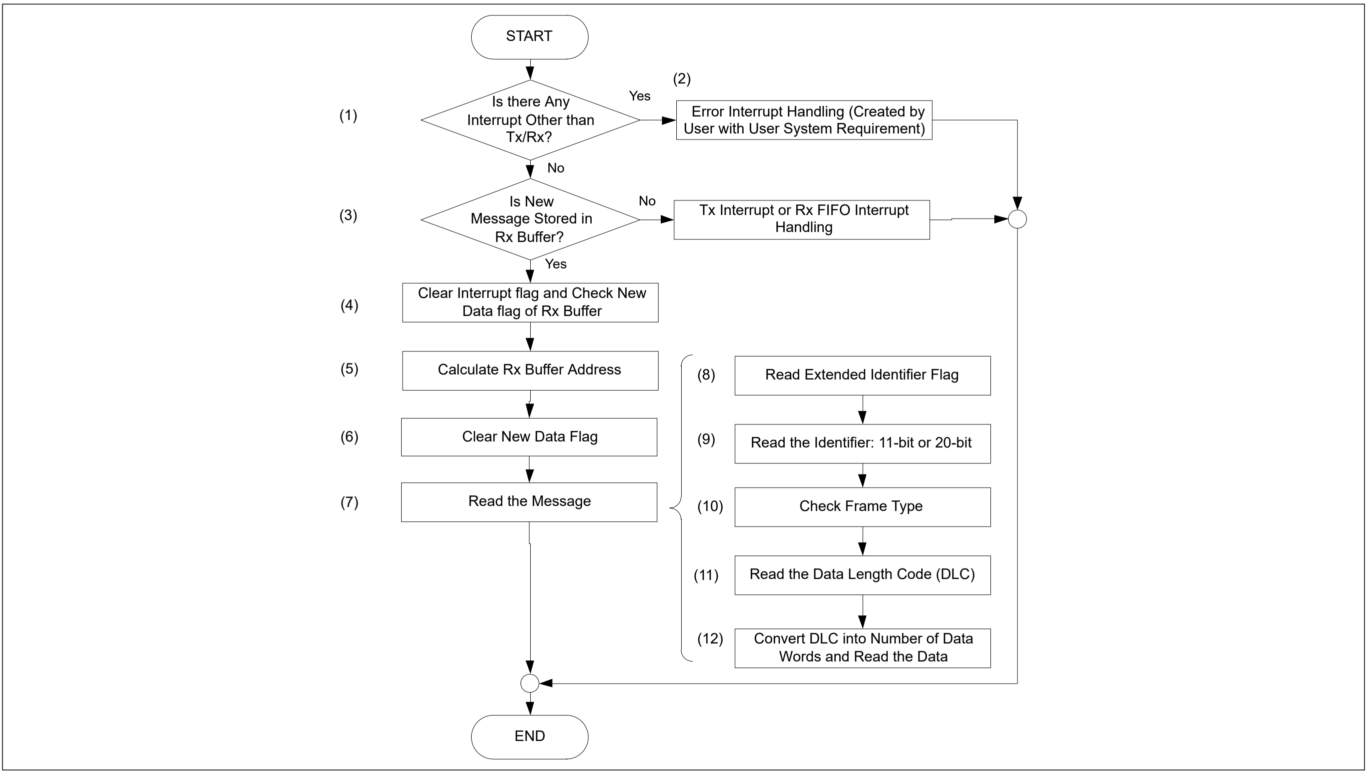
START
(1)
Is there Any Interrupt Other than Tx/Rx?
Yes
(2)
Error Interrupt Handling (Created by User with User System Requirement)
No
(3)
Is New Message Stored in Rx Buffer?
No
Tx Interrupt or Rx FIFO Interrupt Handling
Yes
(4)
Clear Interrupt flag and Check New Data flag of Rx Buffer
(5)
Calculate Rx Buffer Address
(6)
Clear New Data Flag
(7)
Read the Message
(8)
Read Extended Identifier Flag
(9)
Read the Identifier: 11-bit or 20-bit
(10)
Check Frame Type
(11)
Read the Data Length Code (DLC)
(12)
Convert DLC into Number of Data Words and Read the Data
END

Figure 7 **Example of message reception in dedicated Rx buffer flow**

4.4.1.1 Use case

This section describes an example of the message reception in the dedicated Rx buffer using the use case discussed in Use case.

4.4.1.2 Configuration

This section describes an example of the message reception in the dedicated Rx buffer using the configuration discussed in Configuration for CAN FD controller.

4.4.1.3 Example program of message reception in dedicated Rx buffer

Code Listing 7 demonstrates an example program of message reception in the dedicated Rx buffer. This program uses the dedicated Rx buffers 0–31 and the Rx Interrupt.

Code Listing 7 Example of message reception in dedicated Rx buffer

```
void Cy_CANFD_IrqHandler( cy_pstc_canfd_type_t pstcCanFD )
{
    uint32_t*                       pu32Adrs = 0;
    uint8_t                         u8MessageBufferNumber ;
    cy_stc_canfd_msg_t              stcCanFDmsg;
    volatile stc_CAN_CH_M_TTCAN_t*  pstcCanFDChMTTCAN;
    cy_stc_canfd_context_t*         pstcCanFDContext;
    uint8_t                         u8BufferSizeTemp = 0;
    uint32_t                        au32RxBuf[18];

    /* Get pointer to internal data structure */
    pstcCanFDContext = Cy_CANFD_GetContext(pstcCanFD);

    /* Get the pointer to M_TTCAN of the CAN FD channel */
    pstcCanFDChMTTCAN = &pstcCanFD->M_TTCAN;

    /* Other than Tx/Rx interrupt occurred */
    /*  (1) Check for interrupts other than Tx/Rx   */
    if (pstcCanFDChMTTCAN->unIR.u32Register & 0x3ff7E0EE)
    {
    /* (2) Error Interrupt handling (created by user with user system requirement)*/
        Cy_CANFD_ErrorHandling(pstcCanFD);
    }

    /* (3) Check if New Message stored in Dedicated Rx Buffer  */
    if(pstcCanFDChMTTCAN->unIR.stcField.u1DRX == 1) /* At least one message stored into an Rx 
Buffer */
    {
        /* Clear the Message stored to Dedicated Rx Buffer flag */
        /* (4) Clear Interrupt flag and check New Data flag of Dedicated Rx Buffers 0 -31 */
        pstcCanFDChMTTCAN->unIR.stcField.u1DRX = 1UL;

        if(pstcCanFDChMTTCAN->unNDAT1.u32Register != 0)      // Message buffers 0-31
        {
            for(u8MessageBufferNumber = 0; u8MessageBufferNumber < 32; u8MessageBufferNumber++)
            {
                if((pstcCanFDChMTTCAN->unNDAT1.u32Register & (1ul << u8MessageBufferNumber)) != 
0)
                {
                    // Calculate Rx Buffer address
                    /* (5) Get the Rx Buffer address for which New Data flag is set with a 
calculation function (See Code Listing 8) */
                    pu32Adrs = Cy_CANFD_CalcRxBufAdrs(pstcCanFD, u8MessageBufferNumber);

                    // Clear NDAT1 register
                    /* (6) Clear New Data flag */
                    pstcCanFDChMTTCAN->unNDAT1.u32Register = (1ul << u8MessageBufferNumber);

                    break;
                }
            }
        }
        else if(pstcCanFDChMTTCAN->unNDAT2.u32Register != 0)  // Message buffers 32-63
        {
            for(u8MessageBufferNumber = 0; u8MessageBufferNumber < 32; u8MessageBufferNumber++)
            {
                if((pstcCanFDChMTTCAN->unNDAT2.u32Register & (1ul << u8MessageBufferNumber)) !=
0)
                {
                    u8MessageBufferNumber += 32;

                    // Calculate Rx Buffer address
                    pu32Adrs = Cy_CANFD_CalcRxBufAdrs(pstcCanFD, u8MessageBufferNumber);

                    // Clear NDAT2 register
                    pstcCanFDChMTTCAN->unNDAT2.u32Register = (1ul << (u8MessageBufferNumber -
32));

                    break;
                }
            }
        }
        if(pu32Adrs)
        {
            /* (7) Read the message from Rx Buffer (See Code Listing 9) */
            Cy_CANFD_ExtractMsgFromRXBuffer((cy_stc_canfd_rx_buffer_t *) pu32Adrs,
&stcCanFDmsg);

            /* CAN-FD message received, check if there is a callback function */
            /* Call callback function if it was set previously. */
            if (pstcCanFDContext->canFDInterruptHandling.canFDRxInterruptFunction != NULL)
            {
               /* Message handling by application */
                pstcCanFDContext->canFDInterruptHandling.canFDRxInterruptFunction(false,
u8MessageBufferNumber, &stcCanFDmsg);
            }
        }
    }
}
```

Code Listing 8 demonstrates an example program of the Rx buffer address calculation.

Code Listing 8 Example of Rx buffer address calculation

```
static uint32_t* Cy_CANFD_CalcRxBufAdrs(cy_pstc_canfd_type_t pstcCanFD, uint8_t u8MsgBuf)
{
    uint32_t* pu32Adrs;

    if (u8MsgBuf > 63)
    {
        /* Set 0 to the return value if the index is invalid */
        pu32Adrs = NULL;
    }
    else
    {
        /*  Calculate Rx Buffer address   */
        /* Set the message buffer address to the return value if the index is available */
        pu32Adrs =  (uint32_t*)(((uint32_t)pstcCanFD & 0xFFFF0000ul) +
(uint32_t)CY_CANFD_MSGRAM_START);
        pu32Adrs += pstcCanFD->M_TTCAN.unRXBC.stcField.u14RBSA;
        pu32Adrs += u8MsgBuf * (2 + dataBufferSizeInWord[pstcCanFD-
>M_TTCAN.unRXESC.stcField.u3RBDS]);
    }
    return pu32Adrs;
}
```

Code Listing 9 demonstrates an example program of message extraction from Rx buffer.

Code Listing 9 Example of message extraction from Rx buffer

```
/*** Internal function to extract received message from Rx Buffer ***/
void Cy_CANFD_ExtractMsgFromRXBuffer(cy_stc_canfd_rx_buffer_t *pstcRxBufferAddr,
cy_stc_canfd_msg_t *pstcCanFDmsg)
{
    uint16_t     u16Count = 0;
    uint16_t     u16DlcTemp = 0;

    if(0 == pstcRxBufferAddr)
    {
        return;
    }
    /* Save received data */
/* XTD : Extended Identifier */
   /*  (8) Read extended identifier flag (XTD)  */
pstcCanFDmsg->idConfig.extended = pstcRxBufferAddr->r0_f.xtd;

/* ID : RxID */
   /*  (9) Read the identifier: 11-bit or 29-bit depending on the XTD flag  */
if (pstcCanFDmsg->idConfig.extended == false)
{
pstcCanFDmsg->idConfig.identifier = pstcRxBufferAddr->r0_f.id >> 18;
}
else
{
pstcCanFDmsg->idConfig.identifier = pstcRxBufferAddr->r0_f.id;
}

/* FDF : Extended Data Length */
   /*  (10) Check Frame type  */
pstcCanFDmsg->canFDFormat = pstcRxBufferAddr->r1_f.fdf;

/* DLC : Data Length Code */
   /*  (11) Read the Data Length Code (DLC)  */
pstcCanFDmsg->dataConfig.dataLengthCode = pstcRxBufferAddr->r1_f.dlc;

/* Copy 0-64 byte of data area */
if (pstcCanFDmsg->dataConfig.dataLengthCode < 8)
{
u16DlcTemp = 0;
}
else
{
   /*  (12) Convert DLC into number of data words and read the data  */
u16DlcTemp = pstcCanFDmsg->dataConfig.dataLengthCode - 8;
}

for (u16Count = 0; u16Count < iDlcInWord[u16DlcTemp]; u16Count++)
{
pstcCanFDmsg->dataConfig.data[u16Count] = pstcRxBufferAddr->data_area_f[u16Count];
}
```

```
    }
```

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 8 with eight FIFO elements.

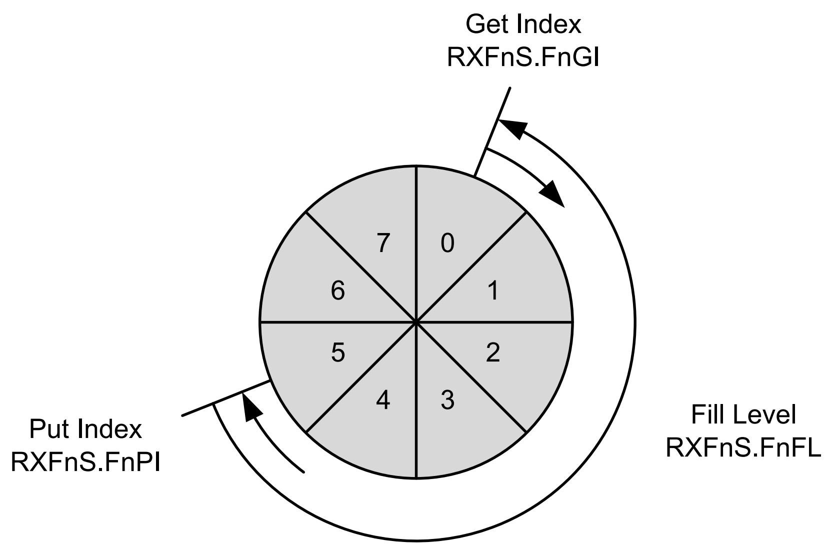

Figure 8 Example of Rx FIFO with eight elements

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, TRAVEO™ T2G 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, therefore, 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 9 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.

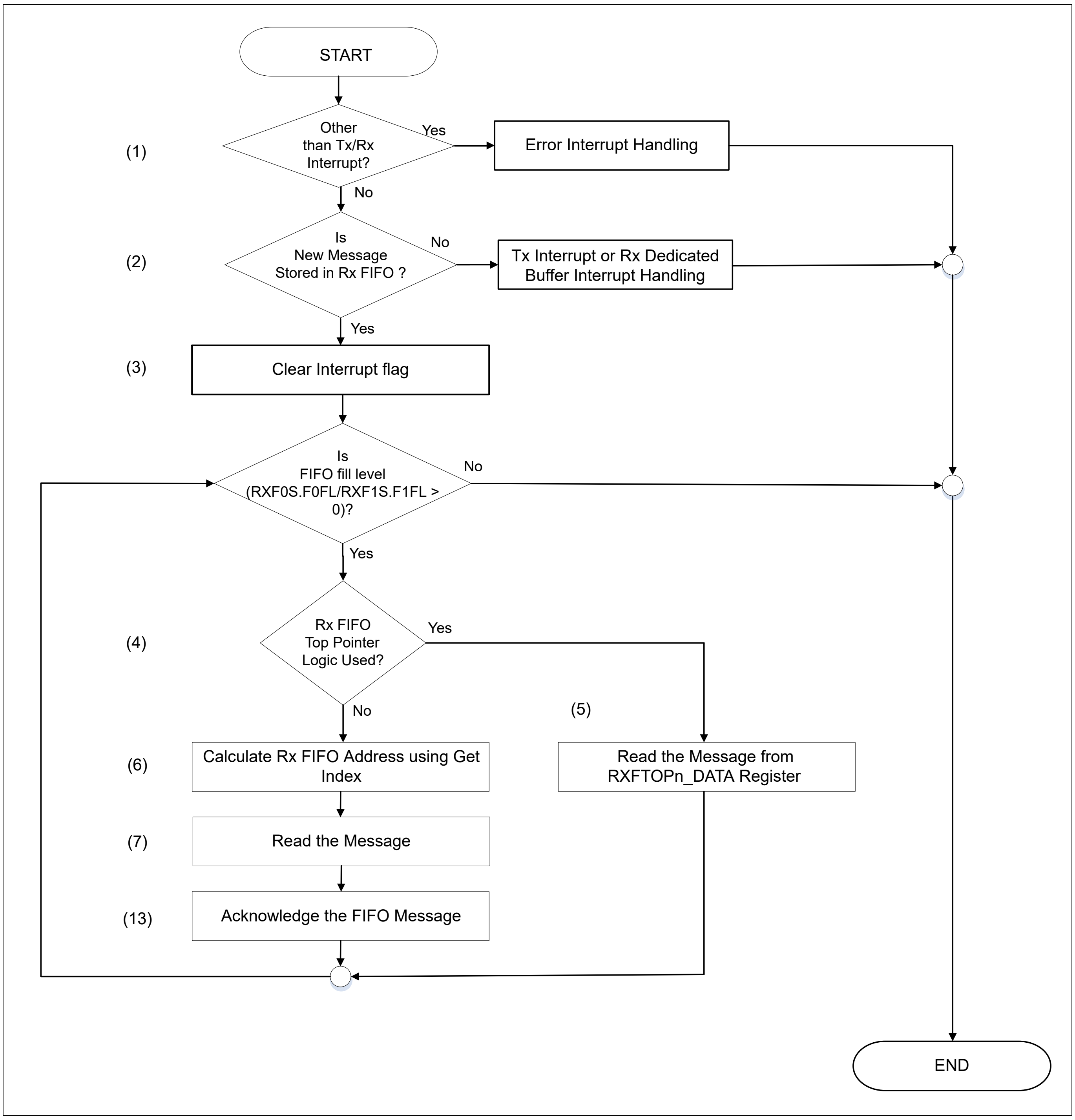
START
(1)
Other than Tx/Rx Interrupt?
Yes
Error Interrupt Handling
No
(2)
Is New Message Stored in Rx FIFO ?
No
Tx Interrupt or Rx Dedicated Buffer Interrupt Handling
Yes
(3)
Clear Interrupt flag
Is FIFO fill level (RXF0S.F0FL/RXF1S.F1FL > 0)?
No
Yes
(4)
Rx FIFO Top Pointer Logic Used?
Yes
No
(5)
(6)
Calculate Rx FIFO Address using Get Index
Read the Message from RXFTOPn_DATA Register
(7)
Read the Message
(13)
Acknowledge the FIFO Message
END

Figure 9 **Example of message reception in Rx FIFO flow**

4.4.2.1 Use case

This section describes an example of message reception in Rx FIFO using the use case discussed in Use case.

4.4.2.2 Configuration

This section describes an example of message reception in Rx FIFO using the configuration discussed in Configuration for CAN FD controller.

4.4.2.3 Example program of message reception in Rx FIFO

Code Listing 10 demonstrates an example program of message reception in Rx FIFO. This program uses the Rx FIFO 0 and the Rx Interrupt.

Code Listing 10 Example of message reception in Rx FIFO

```
void Cy_CANFD_IrqHandler( cy_pstc_canfd_type_t pstcCanFD )
{
    uint32_t*                      pu32Adrs = 0;
    uint8_t                        u8MessageBufferNumber ;
    cy_stc_canfd_msg_t             stcCanFDmsg;
    volatile stc_CAN_CH_M_TTCAN_t* pstcCanFDChMTTCAN;
    cy_stc_canfd_context_t*        pstcCanFDContext;
    uint8_t                        u8BufferSizeTemp = 0;
    uint32_t                       au32RxBuf[18];

    /* Get pointer to internal data structure */
    pstcCanFDContext = Cy_CANFD_GetContext(pstcCanFD);

    /* Get the pointer to M_TTCAN of the CAN FD channel */
    pstcCanFDChMTTCAN = &pstcCanFD->M_TTCAN;

    /* Other than Tx/Rx interrupt occurred */
    /* (1) Check for interrupts other than Tx/Rx */
    if (pstcCanFDChMTTCAN->unIR.u32Register & 0x3ff7E0EE)
    {
        Cy_CANFD_ErrorHandling(pstcCanFD);
    }
    /* (2) Check if New Message stored in Rx FIFO0 */
    else if(pstcCanFDChMTTCAN->unIR.stcField.u1RF0N == 1)   // New message stored into RxFIFO 0
    {
        /* (3) Clear Interrupt flag */
        pstcCanFDChMTTCAN->unIR.stcField.u1RF0N = 1;        // Clear the new message interrupt
flag
while(pstcCanFDChMTTCAN->unRXF0S.stcField.u7F0FL > 0)
{
            /* (4) Check if Rx FIFO 0 Top pointer logic is used */
            if(pstcCanFD->unRXFTOP_CTL.stcField.u1F0TPE == 1)   // RxFifo Top pointer logic is
used
            {
                u8BufferSizeTemp = 2 + dataBufferSizeInWord[pstcCanFD-
>M_TTCAN.unRXESC.stcField.u3F0DS];

                // Now read the RX FIFO Top Data register to copy the content of received
message
                for(uint8_t u8LoopVar = 0u; u8LoopVar < u8BufferSizeTemp; u8LoopVar++)
                {
                /* (5) Read the message content directly from RXFTOP0_DATA register */
                    au32RxBuf[u8LoopVar] = pstcCanFD->unRXFTOP0_DATA.u32Register;
                }

                /* CAN-FD message received, check if there is a callback function */
                /* Call callback function if it was set previously. */
                if (pstcCanFDContext->canFDInterruptHandling.canFDRxWithTopPtrInterruptFunction
!= NULL)
                {
                /* Message handling by application */
```

```
                    pstcCanFDContext-
>canFDInterruptHandling.canFDRxWithTopPtrInterruptFunction(CY_CANFD_RX_FIFO0, u8BufferSizeTemp,
&au32RxBuf[0]);
                }
            }
            else        // RxFifo Top pointer logic is not used
            {
                un_CAN_CH_RXF0S_t unRXF0S;
               /* (6) When Rx FIFO 0 Top pointer logic is not used, get the Rx FIFO address
holding the message at FIFO 0 get index position with a calculation function (See Code Listing
11).*/
                unRXF0S.u32Register = pstcCanFDChMTTCAN->unRXF0S.u32Register;

                pu32Adrs = Cy_CANFD_CalcRxFifoAdrs(pstcCanFD, CY_CANFD_RX_FIFO0,
unRXF0S.stcField.u6F0GI);

                if(pu32Adrs)
                {
                    // Extract the received message from Buffer
                    /* (7) Read the message at get index position (See Code Listing 9) */
                    Cy_CANFD_ExtractMsgFromRXBuffer((cy_stc_canfd_rx_buffer_t *) pu32Adrs,
&stcCanFDmsg);

                    // Acknowledge the FIFO message
                   /* (13) Acknowledge the FIFO 0 message at get index position */
                    pstcCanFDChMTTCAN->unRXF0A.stcField.u6F0AI = unRXF0S.stcField.u6F0GI;

                    /* CAN-FD message received, check if there is a callback function */
                    /* Call callback function if it was set previously. */
                    if (pstcCanFDContext->canFDInterruptHandling.canFDRxInterruptFunction !=
NULL)
                    {
                       /* Message handling by application */
                        pstcCanFDContext->canFDInterruptHandling.canFDRxInterruptFunction(true,
CY_CANFD_RX_FIFO0, &stcCanFDmsg);
                    }
                }
            }
        }
    }
}
```

Code Listing 11 demonstrates an example program of the Rx FIFO address calculation.

Code Listing 11 Example of Rx FIFO address calculation

```
static uint32_t* Cy_CANFD_CalcRxFifoAdrs(cy_pstc_canfd_type_t pstcCanFD, uint8_t u8FifoNumber,
uint32_t u32GetIndex)
{
    uint32_t* pu32Adrs;

    if(u8FifoNumber > 1)
    {
        /* Set 0 to the return value if the FIFO number is invalid */
        pu32Adrs = NULL;
    }
    else
    {
        /*   Calculate the Rx FIFO address   */
        /* Set the message buffer address to the return value if the index is available */
        pu32Adrs = (uint32_t*)(((uint32_t)pstcCanFD & 0xFFFF0000ul) +
(uint32_t)CY_CANFD_MSGRAM_START);
        pu32Adrs += (u8FifoNumber == 0) ? pstcCanFD->M_TTCAN.unRXF0C.stcField.u14F0SA :
pstcCanFD->M_TTCAN.unRXF1C.stcField.u14F1SA;
        pu32Adrs += u32GetIndex * (2 + dataBufferSizeInWord[(u8FifoNumber == 0) ? pstcCanFD-
>M_TTCAN.unRXESC.stcField.u3F0DS : pstcCanFD->M_TTCAN.unRXESC.stcField.u3F1DS]);
    }

    return pu32Adrs;
}
```

5 Glossary

Table 5 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

6 Related documents

The following are the TRAVEO™ T2G family series datasheets and technical reference manuals. Contact Technical Support to obtain these documents.

- Device datasheet
  - CYT2B6 datasheet 32-bit Arm® Cortex®-M4F microcontroller TRAVEO™ T2G family
  - CYT2B7 datasheet 32-bit Arm® Cortex®-M4F microcontroller TRAVEO™ T2G family
  - CYT2B9 datasheet 32-bit Arm® Cortex®-M4F microcontroller TRAVEO™ T2G family
  - CYT2BL datasheet 32-bit Arm® Cortex®-M4F microcontroller TRAVEO™ T2G family
  - CYT3BB/4BB datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family
  - CYT4BF datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family
  - CYT6BJ datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family (Doc No. 002-33466)
  - CYT3DL datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family
  - CYT4DN datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family
  - CYT4EN datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family (Doc No. 002-30842)
  - CYT2CL datasheet 32-bit Arm® Cortex®-M7 microcontroller TRAVEO™ T2G family
- Body controller entry family
  - TRAVEO™ T2G automotive body controller entry family architecture technical reference manual (TRM)
  - TRAVEO™ T2G automotive body controller entry registers technical reference manual (TRM) for CYT2B7
  - TRAVEO™ T2G automotive body controller entry registers technical reference manual (TRM) for CYT2B9
  - TRAVEO™ T2G automotive body controller entry registers technical reference manual (TRM) for CYT2BL (Doc No. 002-29852)
- Body controller high family
  - TRAVEO™ T2G automotive body controller high family architecture technical reference manual (TRM)
  - TRAVEO™ T2G automotive body controller high registers technical reference manual (TRM) for CYT3BB/4BB
  - TRAVEO™ T2G automotive body controller high registers technical reference manual (TRM) for CYT4BF
  - TRAVEO™ T2G automotive body controller high registers technical reference manual (TRM) for CYT6BJ (Doc No. 002-36068)
- Cluster 2D family
  - TRAVEO™ T2G automotive cluster 2D architecture technical reference manual (TRM)
  - TRAVEO™ T2G automotive cluster 2D registers technical reference manual (TRM) for CYT3DL
  - TRAVEO™ T2G automotive cluster 2D registers technical reference manual (TRM) for CYT4DN
  - TRAVEO™ T2G automotive cluster 2D registers technical reference manual (TRM) for CYT4EN (Doc No. 002-35181)
- Cluster entry family
  - TRAVEO™ T2G automotive cluster entry architecture technical reference manual (TRM)
  - TRAVEO™ T2G automotive cluster entry registers technical reference manual (TRM) for CYT2CL
- Application notes
  - AN219755 – Using a SAR ADC in TRAVEO™ T2G family
  - AN219842 – How to use interrupt in TRAVEO™ T2G
  - AN225401 – How to use Serial Communications Block (SCB) in TRAVEO™ T2G family
  - AN220224 – How to use Timer, Counter, and PWM (TCPWM) in TRAVEO™ T2G family

7 Other references

Infineon provides the Sample Driver Library (SDL) including startup as sample software to access various peripherals. SDL also serves as a reference, to customers, for drivers that are not covered by the official AUTOSAR products. The SDL cannot be used for production purposes as it does not qualify to any automotive standards. The code snippets in this application note are part of the SDL. Contact Technical Support to obtain the SDL.

Revision history

Document revision	Date	Description of changes
**	2018-09-28	New application note.
*A	2019-03-07	Updated Associated Part Family as “TRAVEO™ T2G family CYT2B/CYT4B Series”. Added target part numbers “CYT4B Series” related information in all instances across the document. [Section 1, 2.2] - Change to bps from bits/s. [Section 2.2] - Added the explanation of DATA FRAME. - Updated the figure 2 according to the specifications. [Section 4.2] - Updated the flow of Figure 5. - Added the CCCR.INIT setting to (1). - Updated the contents for (2), (3), (7). [Section 4.2.1] - Added new [Section 4.3] - Updated the flow of Figure 6. [Section 4.3.1] - Updated the example code.
*B	2019-09-12	Updated Associated Part Family as “TRAVEO™ T2G family CYT2B/CYT4B/CYT4D Series”. Added target part numbers “CYT4D Series” related information in all instances across the document. [Section 3] - Updated the link of device datasheet [Section 4] - Updated the link of Architecture TRM - Added the link of device datasheet [Section 6] - Changed to new format and added the CYT4D series documents

Revision history

Document revision	Date	Description of changes
*C	2020-03-10	Updated Associated Part Family as “TRAVEO™ T2G family CYT2/CYT3/ CYT4 Series”. Changed target part numbers from “CYT2B/CYT4B/CYT4D Series” to “CYT2/CYT4 Series” in all instances across the document. Added target part numbers “CYT3 Series” in all instances across the document. [Section 3] - Updated the Figure 4 [Section 4] - Updated the flow and code to align [Section 6] - Updated the Related Documents [Section 7] - Added the information of the Sample Driver Library
*D	2021-05-06	Updated to Infineon template.
*E	2023-11-10	Updated the document title. Template update; no content update.
*F	2024-08-29	Updated Related documents and added CYT6 series

Edition 2024-08-29

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
IFX-pvu1681443277092