MCP251XFD driver library

Guide

Language: **C**
Language version: **C99**
Endianness: **Little endian only**

Synchronous: **yes**
Asynchronous: **no**
OS: **need adaptation**

MCU compatibility:
no limit except endianness

Tests: **partial, ~20% coverage**

MISRA C: **to be determined**
CERT C: **to be determined**

PBIT: **yes**
CBIT: **possible**

MCP251XFD driver library

Version: 1.0.2   date: 13 June 2021

This library is compatible with components:
- MCP2517FD
- MCP2518FD

The MCP251XFD component is a CAN-bus controller supporting CAN2.0A, CAN2.0B and CAN-FD with SPI interface

Established by		Reviewed		Approved	
Name:	**FMA**	Name:	FMA	Name:	FMA
Date:	**13 June 2021**	Date:	13 June 2021	Date:	13 June 2021

Change history

Issue	Date	Nature / comment	Paragraph	Writer
1.0.2	13 June 2021	MessageCtrlFlags is a set of instead of an enum	15.5.4	FMA
1.0.1	21 March 2021	Correct MCP251XFD_EnterSleepMode() function's description	15.11	FMA
1.0.0	07 July 2020	**Initial release**	-	FMA

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

Content

Figure summary

Table summary

1. INTRODUCTION

1.1. Purpose

The purpose of this document is to explain how the driver library works and how to use it. It can work with either MCP2517FD controllers or MCP2518FD controllers or both.

The driver features are:
- Can be use with any MCU if its CPU use little endian
- Only take care of the controller, not the communication with it
- All functions and functionalities are implemented
- Configuration is very simplified
- Can communicate with virtually an infinite count of controllers
- Different configurations can be used with different controllers (no duplication of the driver needed)
- Direct communication with the controller, the driver has no buffer
- Can use the driver defines, enums, structs to create your own functions

1.2. Documents, References, and abbreviations

1.2.1. Applicable documents

IDENTIFICATION	TITLE	DATE
DS20005678D	MCP25XXFD CAN FD Controller Module Family Reference Manual	14 May 2019

1.2.2. Reference documents

IDENTIFICATION	TITLE	DATE
DS20005688B	MCP2517FD Datasheet Rev.B – External CAN FD Controller with SPI Interface	July 2019
DS20006027A	MCP2518FD Datasheet Rev.A – External CAN FD Controller with SPI Interface	April 2019

1.2.3. Abbreviations and Acronyms

This is the list of all the abbreviations and acronyms used in this document and their definitions. They are arranged in alphabetical order.

CAN	Controller Area Network
CAN-FD	Controller Area Network Flexible Data-Rate
CBIT	Continuous Built-In Test
CERT	Computer Emergency Response Team
CLK	Clock
CS	Chip Select
FIFO	First In First Out
MCU	Micro-Controller Unit
MISO	Master In Slave Out
MISRA	Motor Industry Software Reliability Association
MOSI	Master Out Slave In
OS	Operating System
PBIT	Power Up Built-In Test
PIO	Programmable Input/Output
RAM	Random Access Memory
SPI	Serial Peripheral Interface
TEF	Transmit Event FIFO
TXQ	Transmit Queue

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

2. PRESENTATION
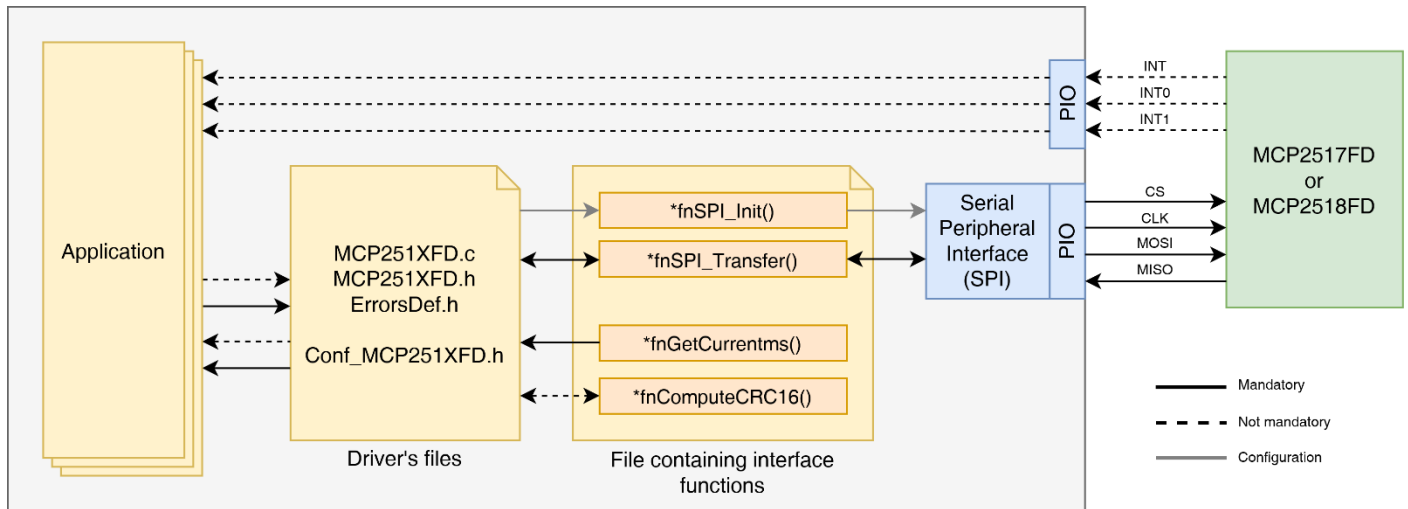
Figure 1 - Driver overview

This driver only takes care of configuration and check of the internal registers and the formatting of the communication with the device. That means it does not directly take care of the physical communication, there is functions interfaces to do that. By doing this, the driver can control a MCP2517FD trough a I2C to SPI converter without any change, the transformation will be done in the interface functions.

Each driver's functions need a device structure that indicate with which device he must threat and communicate. Each device can have its own configuration.

The driver can detect which one of the MCP2517FD or the MCP2518FD is connected.

To set up one or more devices in the project, you must:
- Configure the driver which will be the same for all devices but modify only its behavior in the project
- Create and define the configuration of as many device structures as there are devices to use
- Create and define controller and CAN controller configuration for each device. Multiple devices can share the same configuration
- Initialize the device with the configuration structure previously defined

3. DRIVER CONFIGURATION

The configuration is done by use of "Conf_MCP251XFD.h" file.
This file contains the some defines:
- `#define MCP251XFD_TRANS_BUF_SIZE`
- `#define CHECK_NULL_PARAM`
See example in §17.

3.1. MCP251XFD_TRANS_BUF_SIZE define

This defines the max size for the transfer buffer. Adjust for 1 max full frame that is needed of all controllers in use otherwise the message will be cut into parts that slow the transfer. In case of use write safe by all controllers set 9, which is the minimum allowed, because all data will be send by 4 bytes max.

All read and write possibilities will be presented later.

3.2. CHECK_NULL_PARAM define

This define enables check of pointing parameters are not NULL. It checks if function parameters point to something.

This define can be enabled only in debug. Normally in a static pure C programming, these parameters and function pointer are set and fix, so always checking them is not useful.

4. DEVICE CONFIGURATION

Each first parameter of a function is the device configuration. The purpose of this device configuration is to specify to the driver how and by which interface to communicate with the device selected. The device configuration type is MCP251XFD.

The MCP251XFD.DriverConfig specify how the driver will communicate with the device. Driver configurations, parameters can be OR'ed. The device has 3 protocols to communicate with:
- Normal protocol: just read and write data without verification
- Read and Write with CRC: which read and write data with length and CRC verification
- Safe Write: which write each data with CRC. One byte at a time for SFR and 4 bytes at a time for RAM write

MCP251XFD_DRIVER_USE_READ_WRITE_CRC will always replace MCP251XFD_DRIVER_NORMAL_USE when communicating with the device. MCP251XFD_DRIVER_USE_SAFE_WRITE will always replace MCP251XFD_DRIVER_NORMAL_USE and MCP251XFD_DRIVER_USE_READ_WRITE_CRC when communicating with the device in case of a write. For more information see datasheets.

All other driver configuration is explained in the eMCP251XFD_DriverConfig enumerator.

The MCP251XFD.fnSPI_Init, MCP251XFD.fnSPI_Transfer, MCP251XFD.fnGetCurrentms, and MCP251XFD.fnComputeCRC16 are explained at §14.1.4 and an example at §18.

The MCP251XFD.SPIClockSpeed is the desired frequency of the SPI clock in Hertz. This allows the driver to change the SPI speed and return to the specify clock by its own when necessary. The maximum clock speed is SYSCLK/2 (see §5.1 for SYSCLK).

The MCP251XFD.GPIOsOutState is the wanted state of GPIOs after initialization. It also indicates the last programmed state of the GPIO output thus this value is changed by the driver.

The MCP251XFD.UserDriverData is a generic pointer to what the user need. It can be used as context or bringing information for the interface functions for example. This variable is never touch by the driver.

4.1. Example

Example of driver configuration in a .c file:

```
MCP251XFD MCP251XFD_Ext1 =
{
  .UserDriverData  = NULL,
  //--- Driver configuration ---
  .DriverConfig    = MCP251XFD_DRIVER_USE_READ_WRITE_CRC
                   | MCP251XFD_DRIVER_USE_SAFE_WRITE
                   | MCP251XFD_DRIVER_ENABLE_ECC
                   | MCP251XFD_DRIVER_INIT_SET_RAM_AT_0
                   | MCP251XFD_DRIVER_CLEAR_BUFFER_BEFORE_READ,
  //--- IO configuration ---
  .GPIOsOutState   = MCP251XFD_GPIO0_LOW | MCP251XFD_GPIO1_HIGH,
  //--- Interface driver call functions ---
  .SPI_ChipSelect  = SPI_CS_EXT1, // Here the chip select of the EXT1 interface is 1
  .InterfaceDevice = SPI0,        // Here this point to the address memory of the peripheral SPI0
  .fnSPI_Init      = MCP251XFD_InterfaceInit_V71,
  .fnSPI_Transfer  = MCP251XFD_InterfaceTransfer_V71,
  //--- Time call function ---
  .fnGetCurrentms  = GetCurrentms_V71,
  //--- CRC16-CMS call function ---
  .fnComputeCRC16  = ComputeCRC16_V71,
  //--- Interface clocks ---
  .SPIClockSpeed   = 20000000, // 20MHz
};
```

Example of driver configuration in a .h file:

```
extern MCP251XFD MCP251XFD_Ext1;
#define CANEXT1   &MCP251XFD_Ext1
```

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

5. CONTROLLER AND CAN CONTROLLER CONFIGURATION

The controller and CAN controller configuration is only used at initialization by the *Init_MCP251XFD()* function. The controller and CAN controller configuration type is MCP251XFD_Config.

It configures the controller SYSCLK, configures the CAN controller, the GPIO/interrupt pins, and the interrupts.

5.1. Controller clocks configuration

First, you need to specify the CLKIN frequency. It is indicated in the schematic of the board that contains the MCP251XFD. This clock is either a crystal, a ceramic resonator, or an oscillator.

If both OSC1 and OSC2 is connected then it is a crystal or a ceramic oscillator, in this case put the frequency in Hertz to MCP251XFD_Config.XtalFreq and set MCP251XFD_Config.OscFreq to 0. If only OSC1 is connected and OSC2 is left unconnected then it is an oscillator, in this case put the frequency in Hertz to MCP251XFD_Config.OscFreq and set MCP251XFD_Config.XtalFreq to 0.

Why specify if it is a crystal or an oscillator? Because frequency range differ, and the driver check this.

Next specify the factor of CLKIN which is used for SYSCLK. SYSCLK is the MCP251XFD internal clock used for all its functions like maximum SPI clock, Nominal and Data bit rates. This clock is generated from CLKIN (the clock specified in MCP251XFD_Config.XtalFreq or MCP251XFD_Config.OscFreq).
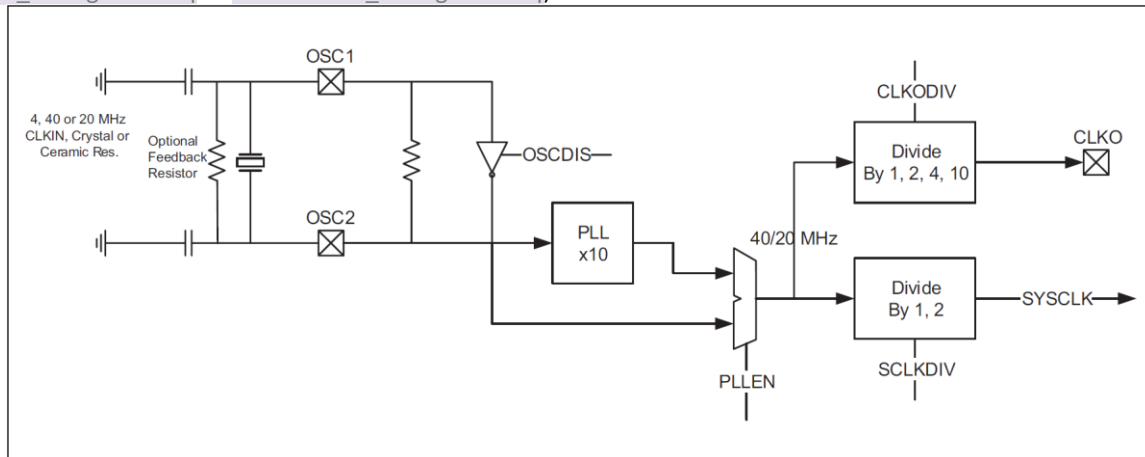
Figure 2 - MCP251XFD Oscillator block diagram (extracted from document DS20005678D page 13)

The minimum operating SYSCLK is 2MHz and the maximum is 40MHz. It is recommended to set a SYSCLK at 20MHz or 40MHz by the CAN specification:

CLKIN (XtalFreq/OscFreq)	Factor (MCP251XFD_Config.SysclkConfig)	SYSCLK
2MHz Oscillator	MCP251XFD_SYSCLK_IS_CLKIN_MUL_10	20MHz
4MHz	MCP251XFD_SYSCLK_IS_CLKIN_MUL_5	20MHz
4MHz	MCP251XFD_SYSCLK_IS_CLKIN_MUL_10	40MHz
20MHz	MCP251XFD_SYSCLK_IS_CLKIN	20MHz
40MHz	MCP251XFD_SYSCLK_IS_CLKIN_DIV_2	20MHz
40MHz	MCP251XFD_SYSCLK_IS_CLKIN	40MHz

Table 1 - Recommended SYSCLK configuration

The SYSCLK frequency is returned in the variable MCP251XFD_Config.SYSCLK_Result if it points to a uint32_t.

The MCP251XFD_Config.ClkoPinConfig is the configuration of the CLKO pin output. All possible value of configuration is indicated in the eMCP251XFD_CLKODIV enumerator.

As you see in Figure 2 - MCP251XFD Oscillator block diagram (extracted from document DS20005678D page 13), the CLKO pin is not linked to SYSCLK but CLKIN or CLKIN+PLL. So, the CLKO output frequency can be superior to SYSCLK. The value at the input of the CLKO divider cannot be superior to 40MHz (like 8MHz CLKIN and PLL enabled for example) and is checked by the driver.

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

If there is an error while configuring the SYSCLK, the driver will return an ERR__FREQUENCY_ERROR. If the MCP251XFD_Config.SYSCLK_Result point to an uint32_t variable, the SYSCLK out of range will be stored inside. With these protections, the device cannot be set wrongly and crash. There is no RESET pin, so these checks are mandatory. If the device crash because the frequency has not been correctly set, there is no choice other than power down and up the card to reset the device.

5.2. CAN controller configuration

The MCP251XFD_Config.NominalBitrate is the desired Nominal bit rate in bits per second.

The MCP251XFD_Config.DataBitrate is the desired Data bit rate in bits per second. If you do not use the CAN-FD feature, then set this variable to 0 or MCP251XFD_NO_CANFD.

Both MCP251XFD_Config.NominalBitrate and MCP251XFD_Config.DataBitrate are used to calculate the bit time configuration of the CAN controller with the minimum quantization error (NBRP and DBRP identical) during bit rate switching. The calculus is done at initialization and set to the controller. The only return of this calculus is the MCP251XFD_Config.BitTimeStats variable where it is indicate the actual Nominal and Data bitrates and oscillator tolerances (the calculus is described in the document DS20005678D, §3.4).

If you want to set the bit time configuration manually, fill the MCP251XFD_BitTimeConfig structure and use the MCP251XFD_SetBitTimeConfiguration() function after using *Init_MCP251XFD()*.

The MCP251XFD_Config.Bandwidth variable specify the delay between two consecutive transmissions. All possible value of configuration is explained in the eMCP251XFD_Bandwidth enumerator.

The MCP251XFD_Config.ControlFlags variable is the CAN control flags to configure the CAN controller. Configuration can be OR'ed. The complete configuration flags are explained in the eMCP251XFD_CANCtrlFlags enumerator.

5.3. GPIOs and Interrupts pins

The MCP251XFD_Config.GPIO0PinMode variable configure what the INT0/GPIO0/XSTBY pin will do. See the eMCP251XFD_GPIO0Mode enumerator for all possibilities.

The MCP251XFD_Config.GPIO1PinMode variable configure what the INT1/GPIO1 pin will do. See the eMCP251XFD_GPIO1Mode enumerator for all possibilities.

The MCP251XFD_Config.INTsOutMode variable specify the output mode of all interrupt pins (INT, INT0 and INT1). The MCP251XFD_Config.TXCANOutMode variable specify the output mode of the TXCAN pin. See the eMCP251XFD_OutMode enumerator for all possibilities.

5.4. Interrupts

The MCP251XFD_Config.SysInterruptFlags variable specify which interrupt event will be activated in the controller and the CAN controller. Interrupt event configuration can be OR'ed. All interrupt events are explained in the document DS20005678D, page 60. For more information about the configuration in the driver see eMCP251XFD_InterruptEvents enumerator.

5.5. Example

Example of controller and CAN controller configuration in a .c file:

```c
MCP251XFD_BitTimeStats MCP2517FD_Ext1_BTStats;
uint32_t SYSCLK_Ext1;

MCP251XFD_Config MCP2517FD_Ext1_Config =
{
  //--- Controller clocks ---
  .XtalFreq        = 0,        // CLKIN is not a crystal
  .OscFreq         = 40000000, // CLKIN is a 40MHz oscillator
  .SysclkConfig    = MCP251XFD_SYSCLK_IS_CLKIN,
  .ClkoPinConfig   = MCP251XFD_CLKO_SOF,
  .SYSCLK_Result   = &SYSCLK_Ext1,
  //--- CAN configuration ---
  .NominalBitrate  = 1000000, // Nominal Bitrate to 1Mbps
  .DataBitrate     = 2000000, // Data Bitrate to 2Mbps
  .BitTimeStats    = &MCP2517FD_Ext1_BTStats,
  .Bandwidth       = MCP251XFD_NO_DELAY,
  .ControlFlags    = MCP251XFD_CAN_RESTRICTED_MODE_ON_ERROR
                   | MCP251XFD_CAN_ESI_REFLECTS_ERROR_STATUS
                   | MCP251XFD_CAN_RESTRICTED_RETRANS_ATTEMPTS
                   | MCP251XFD_CANFD_BITRATE_SWITCHING_ENABLE
                   | MCP251XFD_CAN_PROTOCOL_EXCEPT_AS_FORM_ERROR
                   | MCP251XFD_CANFD_USE_ISO_CRC
                   | MCP251XFD_CANFD_DONT_USE_RRS_BIT_AS_SID11,
  //--- GPIOs and Interrupts pins ---
  .GPIO0PinMode    = MCP251XFD_PIN_AS_GPIO0_OUT,
  .GPIO1PinMode    = MCP251XFD_PIN_AS_INT1_RX,
  .INTsOutMode     = MCP251XFD_PINS_PUSHPULL_OUT,
  .TXCANOutMode    = MCP251XFD_PINS_PUSHPULL_OUT,
  //--- Interrupts ---
  .SysInterruptFlags = MCP251XFD_INT_ENABLE_ALL_EVENTS,
};
```

Example of controller and CAN controller configuration in a .h file:

```c
extern MCP251XFD_BitTimeStats MCP2517FD_Ext1_BTStats;
extern uint32_t SYSCLK_Ext1;
extern MCP251XFD_Config MCP2517FD_Ext1_Config;
```

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

6. FIFO AND FILTER CONFIGURATION

FIFO and Filter configuration can only be done when the device is in Configuration mode. If not, functions will return an ERR__NEED_CONFIG_MODE error.

6.1. FIFO configuration

It is recommended to prepare a full configuration of FIFO in a list of MCP251XFD_FIFO struct and call the MCP251XFD_ConfigureFIFOList() function. In this case, the driver will check the whole place taken in RAM to store all FIFO, if the amount of RAM taken by FIFO list configuration exceeds the maximum RAM of the device, then the function will return a ERR__OUT_OF_MEMORY error. The function will configure FIFO of the device to be identical to the list.

If you choose to configure one FIFO at a time, the driver cannot check the whole FIFO configuration, you must calculate manually the amount of RAM taken by your configuration.

The RAM usage always starts with the TEF, followed by the TXQ and after all the FIFO from 1 to 31. The total RAM size is 2048 bytes. See §14.3 for more information about functions and structures.

There is no way to disable a FIFO, except for the TEF and the TXQ, so it is mandatory to not leave a FIFO not configured between two configured FIFO. If not, this FIFO will take 16 bytes of unused RAM. If the amount of RAM used by your FIFO configuration take the limit and you leave some FIFO unconfigured which will go beyond the maximum of RAM, it is not a problem for the device as long as you do not try to work with these FIFO.

Example:
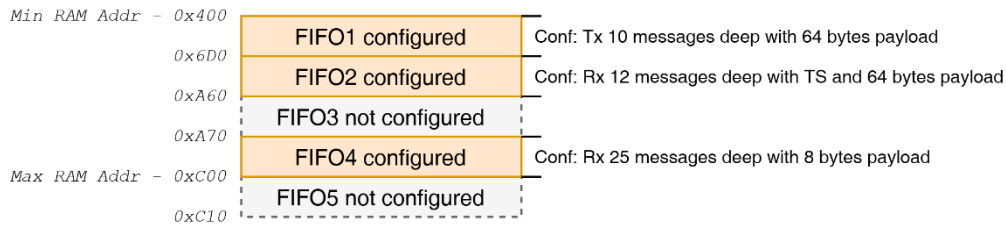
Here is the calculus to know how many RAM a FIFO will take in RAM.
A TEF:

$$T_{TEFSize} = \left(2 \times 4 + \begin{cases} 0 \text{ if } TimeStamp\ disable \\ 4 \text{ if } TimeStamp\ enable \end{cases}\right) \times MessageDeepSize \tag{1.1}$$

A TXQ:

$$T_{TXQSize} = (2 \times 4 + Payloal) \times MessageDeepSize \tag{1.2}$$

A Transmission FIFO:

$$T_{TransmitFIFOSize} = (2 \times 4 + Payloal) \times MessageDeepSize \tag{1.3}$$

A Receive FIFO:

$$T_{ReceiveFIFOSize} = \left(2 \times 4 + Payload + \begin{cases} 0 \text{ if } TimeStamp\ disable \\ 4 \text{ if } TimeStamp\ enable \end{cases}\right) \times MessageDeepSize \tag{1.4}$$

6.2. Filter configuration

It is recommended to prepare a full configuration of Filter in a list of MCP251XFD_Filter struct and call the MCP251XFD_ConfigureFilterList() function. The function will configure Filters of the device to be identical to the list.

You can also configure each filter manually as you want by using the MCP251XFD_ConfigureFilter() function for each Filter. At initialization, all Filters are disable.

These functions verify the Filter consistency, format the ID to correspond to the filter specification.

If you want a Filter to accept all messages, you can use the MCP251XFD_ACCEPT_ALL_MESSAGES for both MCP251XFD_Filter.AcceptanceID and MCP251XFD_Filter.AcceptanceMask.

See §14.4 for more information about functions and structures.

6.3. Example

Example of FIFO and Filter configuration in a .c file:

```c
MCP251XFD_RAMInfos Ext1_TEF_RAMInfos;
MCP251XFD_RAMInfos Ext1_TXQ_RAMInfos;
MCP251XFD_RAMInfos Ext1_FIFOs_RAMInfos[MCP2517FD_EXT1_FIFO_COUNT - 2];

MCP251XFD_FIFO MCP2517FD_Ext1_FIFOlist[MCP2517FD_EXT1_FIFO_COUNT] =
{
  { .Name = MCP251XFD_TEF, .Size = MCP251XFD_FIFO_10_MESSAGE_DEEP, .ControlFlags = MCP251XFD_FIFO_ADD_TIMESTAMP_ON_OBJ
    .InterruptFlags = MCP251XFD_FIFO_OVERFLOW_INT + MCP251XFD_FIFO_EVENT_FIFO_NOT_EMPTY_INT,
    .RAMInfos = &Ext1_TEF_RAMInfos, },
  { .Name = MCP251XFD_TXQ, .Size = MCP251XFD_FIFO_4_MESSAGE_DEEP, .Payload = MCP251XFD_PAYLOAD_64BYTE,
    .Attempts = MCP251XFD_THREE_ATTEMPTS, .Priority = MCP251XFD_MESSAGE_TX_PRIORITY16,
    .ControlFlags = MCP251XFD_FIFO_NO_RTR_RESPONSE,
    .InterruptFlags = MCP251XFD_FIFO_TX_ATTEMPTS_EXHAUSTED_INT + MCP251XFD_FIFO_TRANSMIT_FIFO_NOT_FULL_INT,
    .RAMInfos = &Ext1_TXQ_RAMInfos, },
  { .Name = MCP251XFD_FIFO1, .Size = MCP251XFD_FIFO_4_MESSAGE_DEEP, .Payload = MCP251XFD_PAYLOAD_64BYTE,
    .Direction = MCP251XFD_RECEIVE_FIFO, .ControlFlags = MCP251XFD_FIFO_ADD_TIMESTAMP_ON_RX,
    .InterruptFlags = MCP251XFD_FIFO_OVERFLOW_INT + MCP251XFD_FIFO_RECEIVE_FIFO_NOT_EMPTY_INT,
    .RAMInfos = &Ext1_FIFOs_RAMInfos[1], }, // SID: 0x000..0x1FF ; No EID
  { .Name = MCP251XFD_FIFO2, .Size = MCP251XFD_FIFO_2_MESSAGE_DEEP, .Payload = MCP251XFD_PAYLOAD_64BYTE,
    .Direction = MCP251XFD_RECEIVE_FIFO, .ControlFlags = MCP251XFD_FIFO_ADD_TIMESTAMP_ON_RX,
    .InterruptFlags = MCP251XFD_FIFO_OVERFLOW_INT + MCP251XFD_FIFO_RECEIVE_FIFO_NOT_EMPTY_INT,
    .RAMInfos = &Ext1_FIFOs_RAMInfos[2], }, // SID: 0x200..0x3FF ; No EID
  { .Name = MCP251XFD_FIFO3, .Size = MCP251XFD_FIFO_4_MESSAGE_DEEP, .Payload = MCP251XFD_PAYLOAD_64BYTE,
    .Direction = MCP251XFD_RECEIVE_FIFO, .ControlFlags = MCP251XFD_FIFO_ADD_TIMESTAMP_ON_RX,
    .InterruptFlags = MCP251XFD_FIFO_OVERFLOW_INT + MCP251XFD_FIFO_RECEIVE_FIFO_NOT_EMPTY_INT,
    .RAMInfos = &Ext1_FIFOs_RAMInfos[3], }, // SID: 0x400..0x5FF ; No EID
  { .Name = MCP251XFD_FIFO4, .Size = MCP251XFD_FIFO_2_MESSAGE_DEEP, .Payload = MCP251XFD_PAYLOAD_64BYTE,
    .Direction = MCP251XFD_RECEIVE_FIFO, .ControlFlags = MCP251XFD_FIFO_ADD_TIMESTAMP_ON_RX,
    .InterruptFlags = MCP251XFD_FIFO_OVERFLOW_INT + MCP251XFD_FIFO_RECEIVE_FIFO_NOT_EMPTY_INT,
    .RAMInfos = &Ext1_FIFOs_RAMInfos[4], }, // SID: 0x600..0x7FF ; No EID
  { .Name = MCP251XFD_FIFO5, .Size = MCP251XFD_FIFO_4_MESSAGE_DEEP, .Payload = MCP251XFD_PAYLOAD_64BYTE,
    .Direction = MCP251XFD_TRANSMIT_FIFO, .Attempts = MCP251XFD_THREE_ATTEMPTS,
    .Priority = MCP251XFD_MESSAGE_TX_PRIORITY16, .ControlFlags = MCP251XFD_FIFO_NO_RTR_RESPONSE,
    .InterruptFlags = MCP251XFD_FIFO_TX_ATTEMPTS_EXHAUSTED_INT + MCP251XFD_FIFO_TRANSMIT_FIFO_NOT_FULL_INT,
    .RAMInfos = &Ext1_FIFOs_RAMInfos[5], },
  { .Name = MCP251XFD_FIFO6, .Size = MCP251XFD_FIFO_2_MESSAGE_DEEP, .Payload = MCP251XFD_PAYLOAD_64BYTE,
    .Direction = MCP251XFD_TRANSMIT_FIFO, .Attempts = MCP251XFD_THREE_ATTEMPTS,
    .Priority = MCP251XFD_MESSAGE_TX_PRIORITY16, .ControlFlags = MCP251XFD_FIFO_NO_RTR_RESPONSE,
    .InterruptFlags = MCP251XFD_FIFO_TX_ATTEMPTS_EXHAUSTED_INT + MCP251XFD_FIFO_TRANSMIT_FIFO_NOT_FULL_INT,
    .RAMInfos = &Ext1_FIFOs_RAMInfos[6], },
  { .Name = MCP251XFD_FIFO7, .Size = MCP251XFD_FIFO_2_MESSAGE_DEEP, .Payload = MCP251XFD_PAYLOAD_64BYTE,
    .Direction = MCP251XFD_TRANSMIT_FIFO, .Attempts = MCP251XFD_THREE_ATTEMPTS,
    .Priority = MCP251XFD_MESSAGE_TX_PRIORITY16, .ControlFlags = MCP251XFD_FIFO_NO_RTR_RESPONSE,
    .InterruptFlags = MCP251XFD_FIFO_TX_ATTEMPTS_EXHAUSTED_INT + MCP251XFD_FIFO_TRANSMIT_FIFO_NOT_FULL_INT,
    .RAMInfos = &Ext1_FIFOs_RAMInfos[7], },
  { .Name = MCP251XFD_FIFO8, .Size = MCP251XFD_FIFO_2_MESSAGE_DEEP, .Payload = MCP251XFD_PAYLOAD_64BYTE,
    .Direction = MCP251XFD_TRANSMIT_FIFO, .Attempts = MCP251XFD_THREE_ATTEMPTS,
    .Priority = MCP251XFD_MESSAGE_TX_PRIORITY16, .ControlFlags = MCP251XFD_FIFO_NO_RTR_RESPONSE,
    .InterruptFlags = MCP251XFD_FIFO_TX_ATTEMPTS_EXHAUSTED_INT + MCP251XFD_FIFO_TRANSMIT_FIFO_NOT_FULL_INT,
    .RAMInfos = &Ext1_FIFOs_RAMInfos[8], },
};

MCP251XFD_Filter MCP2517FD_Ext1_FilterList[MCP2517FD_EXT1_FILTER_COUNT] =
{
  { .Filter = MCP251XFD_FILTER0, .EnableFilter = true, .Match = MCP251XFD_MATCH_ONLY_SID,
    .AcceptanceID = 0x000, .AcceptanceMask = 0x600, .PointTo = MCP251XFD_FIFO1, }, // 0x000..0x1FF
  { .Filter = MCP251XFD_FILTER1, .EnableFilter = true, .Match = MCP251XFD_MATCH_ONLY_SID,
    .AcceptanceID = 0x200, .AcceptanceMask = 0x600, .PointTo = MCP251XFD_FIFO2, }, // 0x200..0x3FF
  { .Filter = MCP251XFD_FILTER2, .EnableFilter = true, .Match = MCP251XFD_MATCH_ONLY_SID,
    .AcceptanceID = 0x400, .AcceptanceMask = 0x600, .PointTo = MCP251XFD_FIFO3, }, // 0x400..0x5FF
  { .Filter = MCP251XFD_FILTER3, .EnableFilter = true, .Match = MCP251XFD_MATCH_ONLY_SID,
    .AcceptanceID = 0x600, .AcceptanceMask = 0x600, .PointTo = MCP251XFD_FIFO4, }, // 0x600..0x7FF
};
```

Example of FIFO and Filter configuration in a .h file:

```c
#define MCP2517FD_EXT1_FIFO_COUNT    10
extern MCP251XFD_RAMInfos Ext1_TEF_RAMInfos;
extern MCP251XFD_RAMInfos Ext1_TXQ_RAMInfos;
extern MCP251XFD_FIFO MCP2517FD_Ext1_FIFOlist[MCP2517FD_EXT1_FIFO_COUNT];

#define MCP2517FD_EXT1_FILTER_COUNT  4
extern MCP251XFD_Filter MCP2517FD_Ext1_FilterList[MCP2517FD_EXT1_FILTER_COUNT];
```

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

7. DEVICE INITIALIZATION

Note: _After Power On, the device needs 3ms for the clock to stabilize. Be sure that the MCU takes more than 3ms to be configured or put a 3ms delay before device initialization._

After configuring the device, you need to apply its configuration. To do this, it is pretty simple, first you need to call the _Init_MCP251XFD()_ function with the MCP251XFD struct and the MCP251XFD_Config struct. If all went well, the function would return ERR_OK.

Next you need to call the _MCP251XFD_ConfigureFIFOList()_ function with the list of MCP251XFD_FIFO struct. If all went well, the function would return ERR_OK.

Next you need to call the _MCP251XFD_ConfigureFilterList()_ function with the list of MCP251XFD_Filter struct. If all went well, the function would return ERR_OK.

Currently, the device is still in configuration mode. You can call optional configurations like:
- The MCP251XFD_ConfigureTimeStamp() function if you need to use the Time Stamping of received frames or TEF. If all went well, the function would return ERR_OK.
- The MCP251XFD_ConfigureSleepMode() function if you need to put to sleep the device. If all went well, the function would return ERR_OK.

At this time, if all went fine, the device is fully configured and ready to be connected on the CAN bus.
To do this, execute the function:
- MCP251XFD_StartCAN20() to start the device in CAN2.0 only mode
- MCP251XFD_StartCANFD() to start the device in both CAN2.0 and CAN-FD mode.

If all went well, the function would return ERR_OK.

7.1. Example

Example of device initialization:

```
#define TIMESTAMP_TICK_us       ( 25 )                                    // TimeStamp tick is 25µs
#define TIMESTAMP_TICK(sysclk)  ( ((sysclk) / 1000000) * TIMESTAMP_TICK_us )


//=========================================================================
// Configure the MCP251XFD device on EXT1
//=========================================================================
eERRORRESULT ConfigureMCP251XFDDeviceOnEXT1(void)
{
  //--- Initialize Int pins or GPIOs ---
  MCP251XFD_Ext1_IntPinInit_V71(CANEXT1);       // Hardware dependent, not used by driver
  MCP251XFD_Ext1_Int0Gpio0PinInit_V71(CANEXT1); // Hardware dependent, not used by driver
  MCP251XFD_Ext1_Int1Gpio1PinInit_V71(CANEXT1); // Hardware dependent, not used by driver

  //--- Configure module on Ext1 ---
  eERRORRESULT ErrorExt1 = ERR__NO_DEVICE_DETECTED;
  ErrorExt1 = Init_MCP251XFD(CANEXT1, &MCP2517FD_Ext1_Config);
  if (ErrorExt1 != ERR_OK) return ErrorExt1;
  ErrorExt1 = MCP251XFD_ConfigureTimeStamp(CANEXT1, true, MCP251XFD_TS_CAN20_SOF_CANFD_SOF,
                                    TIMESTAMP_TICK(SYSCLK_Ext1), true);
  if (ErrorExt1 != ERR_OK) return ErrorExt1;
  ErrorExt1 = MCP251XFD_ConfigureFIFOList(CANEXT1, &MCP2517FD_Ext1_FIFOlist[0], MCP2517FD_EXT1_FIFO_COUNT);
  if (ErrorExt1 != ERR_OK) return ErrorExt1;
  ErrorExt1 = MCP251XFD_ConfigureFilterList(CANEXT1, MCP251XFD_D_NET_FILTER_DISABLE,
                                    &MCP2517FD_Ext1_FilterList[0], MCP2517FD_EXT1_FILTER_COUNT);
  if (ErrorExt1 != ERR_OK) return ErrorExt1;
  ErrorExt1 = MCP251XFD_StartCANFD(CANEXT1);
  return ErrorExt1;
}
```

This example is based on the example of configuration showed in §4.1, §5.5, and §6.3.

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

8. TRANSMIT A FRAME AND TRANSMIT FIFO MANAGEMENT

The simplest way to send a frame to a device is to call the MCP251XFD_TransmitMessageToFIFO() function for a FIFO (accept also TXQ) or the MCP251XFD_TransmitMessageToTXQ() function for the TXQ. But this function does not check the actual state of the FIFO. If you try to write a full FIFO, you will overwrite older FIFO/TXQ data.

So, before sending a message, you need to check the state of the FIFO by using the MCP251XFD_GetFIFOStatus() function and check if the is an interrupt flag that indicate there is a room for a frame.

If you have multiple transmit FIFO/TXQ, you can have all their status by calling:
- The MCP251XFD_GetTransmitInterruptStatusOfAllFIFO() function, which gives you a bit field image of the TXQ and all the 31 FIFO about their interrupt pending flag, and their attempt exhaust status flag.
- The MCP251XFD_GetTransmitPendingInterruptStatusOfAllFIFO() function, which gives you a bit field image of the TXQ and all the 31 FIFO about their interrupt pending flag only.

After you can check each bit in the field to know which FIFO has a room for a message, and then use the MCP251XFD_GetFIFOStatus() function to have more details.

If the MCU have access to interrupt pins of the device (INT and/or INT0), you can call:
- The MCP251XFD_GetCurrentTransmitFIFONameAndStatusInterrupt() function, which gives you the FIFO name and its status that generate the interrupt
- The MCP251XFD_GetCurrentTransmitFIFONameInterrupt() function, which gives you only the FIFO name that generate the interrupt

When you are sure that there is a room for a message, call the MCP251XFD_TransmitMessageToFIFO() (or MCP251XFD_TransmitMessageToTXQ()) function and you will send the message. If there are multiple room for messages, call the MCP251XFD_GetFIFOStatus() function at each time.

You do not need to clear the interrupts, the CAN controller will do it for you.

The MCP251XFD_ReceiveMessageFromFIFO() (or MCP251XFD_TransmitMessageToTXQ()) function needs a MCP251XFD_CANMessage struct where all data and information of the frame have to be sent. Do not forget to set a byte array for the payload data with the configured FIFO payload, the driver does not store this information and it will take too much time to get the information from the device.

8.1. Example

8.1.1. Send message to the only one Transmit FIFO with INT0

Example of how to send a message to a single transmit FIFO with interrupt INT0. Only FIFO2 is a transmit FIFO with interrupt on FIFO not full set, INT0 interrupt is set on GPIO0.

```
//=============================================================================
// Transmit a message to MCP251XFD device on EXT1
//=============================================================================
eERRORRESULT TransmitMessageFromEXT1(void)
{
  eERRORRESULT ErrorExt1 = ERR_OK;

  if (ioport_get_pin_level(EXT1_INT0_PIN) == 0)        // Check INT0 pin status of the MCP251XFD (Active low state)
  {
    MCP251XFD_CANMessage TansmitMessage;
    //***** Fill the message as you want *****
    //TansmitMessage.MessageID    = messageID;
    //TansmitMessage.MessageSEQ   = messageSEQ;
    //TansmitMessage.ControlFlags = controlFlags;
    //TansmitMessage.DLC          = dlc;
    //TansmitMessage.PayloadData  = &payloadData[0];
    ErrorExt1 = MCP251XFD_TransmitMessageToFIFO(CANEXT1, &TansmitMessage, MCP251XFD_FIFO2, true); // Send message and
flush
  }
  return ErrorExt1;
}
```

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

8.1.2. Send message to the only one Transmit FIFO without INT0

Example of how to send a message to a single transmit FIFO. Only FIFO2 is a transmit FIFO with interrupt on FIFO not full set, INT0 pin is not used.

```
//============================================================================
// Transmit a message to MCP251XFD device on EXT1
//============================================================================
eERRORRESULT TransmitMessageFromEXT1(void)
{
  eERRORRESULT ErrorExt1 = ERR_OK;
  eMCP251XFD_FIFOstatus FIFOstatus = 0;

  ErrorExt1 = MCP251XFD_GetFIFOStatus(CANEXT1, MCP251XFD_FIFO2, &FIFOstatus); // First get FIFO2 status
  if (ErrorExt1 != ERR_OK) return ErrorExt1;
  if ((FIFOstatus & MCP251XFD_TX_FIFO_NOT_FULL) > 0)                    // Second check FIFO not full
  {
    MCP251XFD_CANMessage TansmitMessage;
    //***** Fill the message as you want *****
    //TansmitMessage.MessageID    = messageID;
    //TansmitMessage.MessageSEQ   = messageSEQ;
    //TansmitMessage.ControlFlags = controlFlags;
    //TansmitMessage.DLC          = dlc;
    //TansmitMessage.PayloadData  = &payloadData[0];

    // Send message and flush
    ErrorExt1 = MCP251XFD_TransmitMessageToFIFO(CANEXT1, &TansmitMessage, MCP251XFD_FIFO2, true);
  }
  return ErrorExt1;
}
```

8.1.3. Send message to a Transmit FIFO with INT0

Example of how to send a message to a single transmit FIFO with interrupt INT0. Multiple FIFO are a transmit FIFO with interrupt on FIFO not full set, INT0 interrupt is set on GPIO0.

```
//============================================================================
// Transmit a message to MCP251XFD device on EXT1
//============================================================================
eERRORRESULT TransmitMessageFromEXT1(void)
{
  eERRORRESULT ErrorExt1 = ERR_OK;
  eMCP251XFD_FIFO FIFOname;
  eMCP251XFD_FIFOstatus FIFOstatus = 0;

  if (ioport_get_pin_level(EXT1_INT0_PIN) == 0)        // Check INT0 pin status of the MCP251XFD (Active low state)
  {
    ErrorExt1 = MCP251XFD_GetCurrentTransmitFIFONameAndStatusInterrupt(CANEXT1, &FIFOname, &FIFOstatus);
    if (ErrorExt1 != ERR_OK) return ErrorExt1;         // First get which FIFO set interrupt and its status
    // Second check FIFO not empty
    if (((FIFOstatus & MCP251XFD_TX_FIFO_NOT_FULL) > 0) && (FIFOname != MCP251XFD_NO_FIFO))
    {
      MCP251XFD_CANMessage TansmitMessage;
      //***** Fill the message as you want *****
      //TansmitMessage.MessageID    = messageID;
      //TansmitMessage.MessageSEQ   = messageSEQ;
      //TansmitMessage.ControlFlags = controlFlags;
      //TansmitMessage.DLC          = dlc;
      //TansmitMessage.PayloadData  = &payloadData[0];

      // Send message and flush
      ErrorExt1 = MCP251XFD_TransmitMessageToFIFO(CANEXT1, &TansmitMessage, FIFOname, true);
    }
  }
  return ErrorExt1;
}
```

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

8.1.4. Send message to a Transmit FIFO without INT0

Example of how to send a message to a single transmit FIFO. Multiple FIFO are a transmit FIFO with interrupt on FIFO not full set.

```c
//==============================================================================
// Transmit messages to MCP251XFD device on EXT1
//==============================================================================
eERRORRESULT TransmitMessageFromEXT1(void)
{
  eERRORRESULT ErrorExt1 = ERR_OK;
  eMCP251XFD_FIFOstatus FIFOstatus = 0;
  setMCP251XFD_InterruptOnFIFO InterruptOnFIFO = 0;
  ErrorExt1 = MCP251XFD_GetTransmitPendingInterruptStatusOfAllFIFO(CANEXT1, &InterruptOnFIFO); // Get all FIFO status
  if (ErrorExt1 != ERR_OK) return ErrorExt1;
  for (eMCP251XFD_FIFO zFIFO = 0; zFIFO < MCP251XFD_TX_FIFO_MAX; zFIFO++) // For each transmit FIFO, TXQ but not TEF
    if ((InterruptOnFIFO & (1 << zFIFO)) > 0)                            // If an Interrupt is flagged
    {
      ErrorExt1 = MCP251XFD_GetFIFOStatus(CANEXT1, zFIFO, &FIFOstatus);   // Get the status of the flagged FIFO
      if (ErrorExt1 != ERR_OK) return ErrorExt1;
      if ((FIFOstatus & MCP251XFD_TX_FIFO_NOT_FULL) > 0)                 // Check FIFO not empty
      {
        MCP251XFD_CANMessage TansmitMessage;
        //***** Fill the message as you want *****
        //TansmitMessage.MessageID   = messageID;
        //TansmitMessage.MessageSEQ  = messageSEQ;
        //TansmitMessage.ControlFlags = controlFlags;
        //TansmitMessage.DLC         = dlc;
        //TansmitMessage.PayloadData  = &payloadData[0];
        ErrorExt1 = MCP251XFD_TransmitMessageToFIFO(CANEXT1, &TansmitMessage, zFIFO, true); // Send message and flush
      }
    }
  return ErrorExt1;
}
```

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

9. RECEIVE A FRAME AND RECEIVE FIFO MANAGEMENT

The simplest way to get a receive frame from a device is to call the MCP251XFD_ReceiveMessageFromFIFO() function for a FIFO (accept also TEF) or the MCP251XFD_ReceiveMessageToTEF() function for the TEF. But this function does not check the actual state of the FIFO. If you try to read an empty FIFO, you will get garbage data.

So, before getting a message, you need to check the state of the FIFO/TEF by using the MCP251XFD_GetFIFOStatus() function and check if the is an interrupt flag that indicate there is a frame received.

If you have multiple receive FIFO (TEF needs to be address specifically), you can have all their status by calling:
- The MCP251XFD_GetReceiveInterruptStatusOfAllFIFO() function, which gives you a bit field image of all the 31 FIFO (TEF not included) about their interrupt pending flag, and their overflow status flag.
- The MCP251XFD_GetReceivePendingInterruptStatusOfAllFIFO() function, which gives you a bit field image of all the 31 FIFO about their interrupt pending flag only.

After you can check each bit in the field to know which FIFO has a message pending, and then use the MCP251XFD_GetFIFOStatus() function to have more details.

If the MCU have access to interrupt pins of the device (INT and/or INT1), you can call:
- The MCP251XFD_GetCurrentReceiveFIFONameAndStatusInterrupt() function, which gives you the FIFO/TEF name and its status that generate the interrupt
- The MCP251XFD_GetCurrentReceiveFIFONameInterrupt() function, which gives you only the FIFO/TEF name that generate the interrupt

When you are sure that a message is pending, call the MCP251XFD_ReceiveMessageFromFIFO() (or MCP251XFD_ReceiveMessageToTEF()) function and you will get the message. If there are multiple messages pending, call the MCP251XFD_GetFIFOStatus() function at each time.

You do not need to clear the interrupts, the CAN controller will do it for you.

The MCP251XFD_ReceiveMessageFromFIFO() (or MCP251XFD_ReceiveMessageToTEF()) function needs a MCP251XFD_CANMessage struct where all known data of the frame will be stored. Do not forget to set a byte array for the payload data with the configured FIFO payload, the driver doesn't store this information and it will take too much time to get the information from the device.

9.1. Example

9.1.1. Get message from the only one Receive FIFO with INT1

Example of how to get a message from a single receive FIFO with interrupt INT1. Only FIFO1 is a receive FIFO with interrupt on FIFO not empty set, INT1 interrupt is set on GPIO1.

```
//=============================================================================
// Receive a message from MCP251XFD device on EXT1
//=============================================================================
eERRORRESULT ReceiveMessageFromEXT1(void)
{
  eERRORRESULT ErrorExt1 = ERR_OK;

  if (ioport_get_pin_level(EXT1_INT1_PIN) == 0)      // Check INT1 pin status of the MCP251XFD (Active low state)
  {
    uint32_t MessageTimeStamp = 0;
    uint8_t RxPayloadData[64];                        // In this example, the FIFO1 have 64 bytes of payload
    MCP251XFD_CANMessage ReceivedMessage;
    ReceivedMessage.PayloadData = &RxPayloadData[0]; // Add receive payload data pointer to the message structure
                                                     // that will be received
    ErrorExt1 = MCP251XFD_ReceiveMessageFromFIFO(CANEXT1, &ReceivedMessage, MCP251XFD_PAYLOAD_64BYTE,
                                                 &MessageTimeStamp, MCP251XFD_FIFO1);
    if (ErrorExt1 == ERR_OK)
    {
      //***** Do what you want with the message *****

    }
  }
  return ErrorExt1;
}
```

Ref: MCP251XFD driver library guide
                                                                                    (Synchronous driver).docx

9.1.2. Get message from the only one Receive FIFO without INT1

Example of how to get a message from a single receive FIFO. Only FIFO1 is a receive FIFO with interrupt on FIFO not empty set, INT1 pin is not used.

```c
//==============================================================================
// Receive a message from MCP251XFD device on EXT1
//==============================================================================
eERRORRESULT ReceiveMessageFromEXT1(void)
{
  eERRORRESULT ErrorExt1 = ERR_OK;
  eMCP251XFD_FIFOstatus FIFOstatus = 0;

  ErrorExt1 = MCP251XFD_GetFIFOStatus(CANEXT1, MCP251XFD_FIFO1, &FIFOstatus); // First get FIFO1 status
  if (ErrorExt1 != ERR_OK) return ErrorExt1;
  if ((FIFOstatus & MCP251XFD_RX_FIFO_NOT_EMPTY) > 0) // Second check FIFO not empty
  {
    uint32_t MessageTimeStamp = 0;
    uint8_t RxPayloadData[64];                        // In this example, the FIFO1 have 64 bytes of payload
    MCP251XFD_CANMessage ReceivedMessage;
    ReceivedMessage.PayloadData = &RxPayloadData[0];  // Add receive payload data pointer to the message structure
                                                      // that will be received
    ErrorExt1 = MCP251XFD_ReceiveMessageFromFIFO(CANEXT1, &ReceivedMessage, MCP251XFD_PAYLOAD_64BYTE,
                                                 &MessageTimeStamp, MCP251XFD_FIFO1);

    if (ErrorExt1 == ERR_OK)
    {
      //***** Do what you want with the message *****

    }
  }
  return ErrorExt1;
}
```

9.1.3. Get message from a Receive FIFO with INT1

Example of how to get a message from a single receive FIFO with interrupt INT1. Multiple FIFO are a receive FIFO with interrupt on FIFO not empty set, INT1 interrupt is set on GPIO1.

```c
//==============================================================================
// Receive a message from MCP251XFD device on EXT1
//==============================================================================
eERRORRESULT ReceiveMessageFromEXT1(void)
{
  eERRORRESULT ErrorExt1 = ERR_OK;
  eMCP251XFD_FIFO FIFOname;
  eMCP251XFD_FIFOstatus FIFOstatus = 0;

  if (ioport_get_pin_level(EXT1_INT1_PIN) == 0)       // Check INT1 pin status of the MCP251XFD (Active low state)
  {
    ErrorExt1 = MCP251XFD_GetCurrentReceiveFIFONameAndStatusInterrupt(CANEXT1, &FIFOname, &FIFOstatus);
    if (ErrorExt1 != ERR_OK) return ErrorExt1;        // First get which FIFO set interrupt and its status
    // Second check FIFO not empty
    if (((FIFOstatus & MCP251XFD_RX_FIFO_NOT_EMPTY) > 0) && (FIFOname != MCP251XFD_NO_FIFO))
    {
      uint32_t MessageTimeStamp = 0;
      uint8_t RxPayloadData[8];                        // In this example, all the FIFO have 8 bytes of payload
      MCP251XFD_CANMessage ReceivedMessage;
      ReceivedMessage.PayloadData = &RxPayloadData[0]; // Add receive payload data pointer to the message structure
                                                       // that will be received
      ErrorExt1 = MCP251XFD_ReceiveMessageFromFIFO(CANEXT1, &ReceivedMessage, MCP251XFD_PAYLOAD_8BYTE,
                                                   &MessageTimeStamp, MCP251XFD_FIFO1);

      if (ErrorExt1 == ERR_OK)
      {
        //***** Do what you want with the message *****

      }
    }
  }
  return ErrorExt1;
}
```

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

9.1.4. Get message from a Receive FIFO without INT1

Example of how to get a message from a single receive FIFO. Multiple FIFO are a receive FIFO with interrupt on FIFO not empty set.

```c
//===========================================================================
// Receive messages from MCP251XFD device on EXT1
//===========================================================================
eERRORRESULT ReceiveMessageFromEXT1(void)
{
  eERRORRESULT ErrorExt1 = ERR_OK;
  eMCP251XFD_FIFOstatus FIFOstatus = 0;
  setMCP251XFD_InterruptOnFIFO InterruptOnFIFO = 0;
  ErrorExt1 = MCP251XFD_GetReceivePendingInterruptStatusOfAllFIFO(CANEXT1, &InterruptOnFIFO); // Get all FIFO status
  if (ErrorExt1 != ERR_OK) return ErrorExt1;
  for (eMCP251XFD_FIFO zFIFO = 1; zFIFO < MCP251XFD_FIFO_MAX; zFIFO++)  // For each receive FIFO but not TEF, TXQ
    if ((InterruptOnFIFO & (1 << zFIFO)) > 0)                          // If an Interrupt is flagged
    {
      ErrorExt1 = MCP251XFD_GetFIFOStatus(CANEXT1, zFIFO, &FIFOstatus); // Get the status of the flagged FIFO
      if (ErrorExt1 != ERR_OK) return ErrorExt1;
      if ((FIFOstatus & MCP251XFD_RX_FIFO_NOT_EMPTY) > 0) // Check FIFO not empty
      {
        uint32_t MessageTimeStamp = 0;
        uint8_t RxPayloadData[8];                        // In this example, all the FIFO have 8 bytes of payload
        MCP251XFD_CANMessage ReceivedMessage;
        ReceivedMessage.PayloadData = &RxPayloadData[0];  // Add receive payload data pointer to the message
                                                          // structure that will be received
        ErrorExt1 = MCP251XFD_ReceiveMessageFromFIFO(CANEXT1, &ReceivedMessage, MCP251XFD_PAYLOAD_8BYTE,
                                                     &MessageTimeStamp, zFIFO);
        if (ErrorExt1 == ERR_OK)
        {
          //***** Do what you want with the message *****

        }
      }
    }
  return ErrorExt1;
}
```

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

10. SLEEP

First of all, you need to configure to device Sleep Mode with the MCP251XFD_ConfigureSleepMode() function. You can configure at device initialization if you will use always the same Sleep Mode (see §7) or right before putting the device in Sleep Mode. It is at the configuration of the Sleep Mode that you indicate the Sleep Mode or Deep Sleep Mode.

To put a device in Sleep Mode (MCP2517FD and MCP2518FD) or Deep Sleep Mode (MCP2518FD only), just use the MCP251XFD_EnterSleepMode() function. If all went well, the function would return ERR_OK.

If you want to manually wake up the device, simply use the MCP251XFD_WakeUp() function. If all went well, the function would return ERR_OK.

After waking up, if the device was in Sleep Mode, then the device is in configuration mode.

After waking up, if the device was in Deep Sleep Mode, then the device needs a complete initialization (see §7) because a wake up from Deep Sleep Mode is similar to a Power-On-Reset. *After Power On, the device needs 3ms for the clock to stabilize.*

The MCP251XFD_WakeUp() function can tell from which Sleep Mode the device is wake up. You can use it to take the proper action. If you know the Sleep Mode, then put a NULL on this parameter.

If the device is wake up from the CAN bus, then there is two ways to know that:
- By using the MCP251XFD_IsDeviceInSleepMode() function regularly. This function works only if the device is not in Deep Sleep Mode, because a simple assert of SPI Chip Select wakes up from Deep Sleep Mode, so it is impossible to ask the device if it is actually in Deep Sleep Mode.
- By checking MCP251XFD_WAKEUP_INTERRUPT flag code and MCP251XFD_INT_BUS_WAKEUP_EVENT interrupt event regularly or when the INT pin goes low level.

The MCP251XFD_WAKEUP_INTERRUPT flag code is mainly available when the device is in Sleep Mode whereas MCP251XFD_INT_BUS_WAKEUP_EVENT interrupt event stays up after a Deep Sleep wake up or after a Sleep. If you want to know from which state the device was wake up by CAN bus, use the MCP251XFD_BusWakeUpFromState() function.

After a wake up from bus, it is mandatory to call the MCP251XFD_BusWakeUpFromState() function, otherwise the driver will think that the device is still in Sleep Mode and will refuse most of interactions with the device by thinking the device is in Sleep Mode.

Note: You can also use the MCP251XFD_Config.ClkoPinConfig configured as MCP251XFD_CLKO_SOF and linked to the MCU as an interrupt pin to detect the device wake up.

10.1. Examples

10.1.1. Put device in sleep mode

Example of putting device in sleep mode:

```
//===========================================================================
// Put in Sleep Mode device on EXT1
//===========================================================================
eERRORRESULT PutInSleepModeDeviceOnEXT1(void)
{
  eERRORRESULT ErrorExt1;
  ErrorExt1 = MCP251XFD_ConfigureSleepMode(CANEXT1, false, MCP251XFD_T11FILTER_300ns, true);
  if (ErrorExt1 == ERR_OK)
  {
    ErrorExt1 == MCP251XFD_EnterSleepMode(CANEXT1);
    // Here the device is in sleep mode
  }
  return ErrorExt1;
}


//===========================================================================
// Put in Deep Sleep Mode device on EXT1
//===========================================================================
eERRORRESULT PutInDeepSleepModeDeviceOnEXT1(void)
{
  eERRORRESULT ErrorExt1;
  ErrorExt1 = MCP251XFD_ConfigureSleepMode(CANEXT1, true, MCP251XFD_T11FILTER_300ns, true);
  if (ErrorExt1 == ERR_OK)
  {
    ErrorExt1 == MCP251XFD_EnterSleepMode(CANEXT1);
    // Here the device is in deep sleep mode
  }
```

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

```
  else
  {
    if (ErrorExt1 == ERR__NOT_SUPPORTED)
    {
      // Goes here if the device is a MCP2517FD which does not support DeepSleep
    }
  }
  return ErrorExt1;
}
```

10.1.2. Manually wake up device from sleep mode

Example of manually wake up device from sleep mode:

```
//=============================================================================
// Do actions after a wake up of the MCP251XFD device on EXT1
//=============================================================================
eERRORRESULT AfterWakeUpActionsOnEXT1(eMCP251XFD_PowerStates fromState)
{
  eERRORRESULT ErrorExt1 = ERR_OK;
  switch (fromState)
  {
    case MCP251XFD_DEVICE_SLEEP_STATE:           // In sleep state, the device is wake up but in configuration mode
      ErrorExt1 = MCP251XFD_StartCANFD(CANEXT1); // Start the CAN-FD mode
      break;
    case MCP251XFD_DEVICE_LOWPOWER_SLEEP_STATE: // The device is wake up from deep sleep is similar to a Power On Reset
      delay_ms(3); // Wait 3ms for the clock to stabilize. This is mandatory for the wake up from deep sleep mode
      ErrorExt1 = ConfigureMCP251XFDDeviceOnEXT1(); // The device need a complete reconfiguration
      break;
    default:
      break;
  }
  return ErrorExt1;
}


//=============================================================================
// Wake Up device on EXT1
//=============================================================================
eERRORRESULT WakeUpDeviceOnEXT1(void)
{
  eERRORRESULT ErrorExt1 = ERR_OK;
  eMCP251XFD_PowerStates PowerStateBeforeWakeUp;
  ErrorExt1 = MCP251XFD_WakeUp(CANEXT1, &PowerStateBeforeWakeUp); // Wake up device
  if (ErrorExt1 != ERR_OK) return ErrorExt1;
  // Device is wake up
  ErrorExt1 = AfterWakeUpActionsOnEXT1(PowerStateBeforeWakeUp);
  return ErrorExt1;
}
```

10.1.3. Automatic wake up device from sleep mode by CAN

Example of how to know the device is wake up from sleep mode by CAN:

```
//=============================================================================
// Do actions after a wake up of the MCP251XFD device on EXT1
//=============================================================================
eERRORRESULT AfterWakeUpActionsOnEXT1(eMCP251XFD_PowerStates fromState)
{
  eERRORRESULT ErrorExt1 = ERR_OK;
  switch (fromState)
  {
    case MCP251XFD_DEVICE_SLEEP_STATE:           // In sleep state, the device is wake up but in configuration mode
      ErrorExt1 = MCP251XFD_StartCANFD(CANEXT1); // Start the CAN-FD mode
      break;
    case MCP251XFD_DEVICE_LOWPOWER_SLEEP_STATE: // The device is wake up from deep sleep is similar to a Power On Reset
      delay_ms(3); // Wait 3ms for the clock to stabilize. This is mandatory for the wake up from deep sleep mode
      ErrorExt1 = ConfigureMCP251XFDDeviceOnEXT1(); // The device need a complete reconfiguration
      break;
    default:
      break;
  }
  return ErrorExt1;
}


//=============================================================================
// Check device interrupt on EXT1
```

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

```
//============================================================================
void CheckDeviceINTOnEXT1(void)
{
  eERRORRESULT ErrorExt1;
  setMCP251XFD_InterruptEvents Events;
  eMCP251XFD_PowerStates PowerStateBeforeWakeUp;
  eMCP251XFD_InterruptFlagCode InterruptCode = 0;

  ErrorExt1 = MCP251XFD_GetCurrentInterruptEvent(CANEXT1, &InterruptCode); // Get the current Interrupt event
  if (ErrorExt1 != ERR_OK) return ErrorExt1;
  switch (InterruptCode)
  {
    // Here handle others interrupts

    case MCP251XFD_WAKEUP_INTERRUPT:        // Wake-up interrupt
      ErrorExt1 = MCP251XFD_ClearInterruptEvents(CANEXT1, MCP251XFD_INT_BUS_WAKEUP_EVENT);
      if (ErrorExt1 != ERR_OK) return ErrorExt1;
      PowerStateBeforeWakeUp = MCP251XFD_BusWakeUpFromState(CANEXT1);
      ErrorExt1 = AfterWakeUpActionsOnEXT1(PowerStateBeforeWakeUp);
      if (ErrorExt1 != ERR_OK) return ErrorExt1;
      break;

    default:
      //--- Check others interrupts ---
      ErrorExt1 = MCP251XFD_GetInterruptEvents(CANEXT1, &Events);
      if (ErrorExt1 != ERR_OK) return ErrorExt1;
      if ((Events & MCP251XFD_INT_BUS_WAKEUP_EVENT) > 0) // Wake-up event (here is the one for the deep sleep state
                                                        // with wakeup from bus)
      { // Why here? Because there is a flood of events when wakeup in this specific case, and this one is the most
          important but not the one with the highest priority
        ErrorExt1 = MCP251XFD_ClearInterruptEvents(CANEXT1, MCP251XFD_INT_BUS_WAKEUP_EVENT);
        if (ErrorExt1 != ERR_OK) return ErrorExt1;
        PowerStateBeforeWakeUp = MCP251XFD_BusWakeUpFromState(CANEXT1);
        ErrorExt1 = AfterWakeUpActionsOnEXT1(PowerStateBeforeWakeUp);
        if (ErrorExt1 != ERR_OK) return ErrorExt1;
      }
      break;
  }
  return ErrorExt1;
}
```

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

11. INTERRUPT MANAGEMENT

The interrupt management can return multiple interrupts at the same time, depending on which interrupt you have enabled on the MCP251XFD_Config.SysInterruptFlags configuration. If there is an interrupt the INT pin will be in low level state.

If you use the INT pin, regularly check its state to know if there are pending interrupt flags. If you do not use the INT pin, you must check regularly the interrupt flags.

You have 2 main ways to check interrupts, both are complementary:
- The MCP251XFD_GetCurrentInterruptEvent() function, which gives you the current interrupt including from specific FIFO/TEF/TXQ but not some specific errors and is lost after a wake up from Deep Sleep
- The MCP251XFD_GetInterruptEvents() function, which gives you general and specific errors (all at once) but not what FIFO generate the interrupt but is available after a wake up from Deep Sleep

All depend on how you configure the device and what do you want to do. Of course, you can use both as shown in the example §11.1 but if you use INT0 and INT1 pins the MCP251XFD_GetCurrentInterruptEvent() function may not be useful. If you use the Deep Sleep Mode, the MCP251XFD_GetInterruptEvents() function is almost mandatory to use (see §10).

Some interrupt needs to be cleared manually, see eMCP251XFD_InterruptEvents or the example §11.1.
For more information about the interrupt, see DS20005678D, §10.0.

11.1. Example

Example of all possible interrupt that can be catch:

```c
//============================================================================
// Check device interrupt (INT) on EXT1
//============================================================================
void CheckDeviceINTOnEXT1(void)
{
  eERRORRESULT ErrorExt1;

#ifdef APP_USE_EXT1_INT_PIN
  if (ioport_get_pin_level(EXT1_INT_PIN) != 0) return; // Check INT pin status of the MCP251XFD (Active low state)
#endif
  eMCP251XFD_InterruptFlagCode InterruptCode = 0;
  ErrorExt1 = MCP251XFD_GetCurrentInterruptEvent(CANEXT1, &InterruptCode); // Get the current Interrupt event
  if (ErrorExt1 != ERR_OK) return ErrorExt1;
  switch (InterruptCode)
  {
    case MCP251XFD_ERROR_INTERRUPT:          // Error Interrupt
      LOGERROR("Ext1: CAN Bus Error");
      MCP251XFD_ClearInterruptEvents(CANEXT1, MCP251XFD_INT_BUS_ERROR_EVENT);
      break;

    case MCP251XFD_WAKEUP_INTERRUPT:         // Wake-up interrupt
      // See Sleep §10
      MCP251XFD_ClearInterruptEvents(CANEXT1, MCP251XFD_INT_BUS_WAKEUP_EVENT);
      break;

    case MCP251XFD_RECEIVE_FIFO_OVF:         // Receive FIFO Overflow Interrupt
      // Only available on INT pin not on INT1 pin
      // Use MCP251XFD_GetReceiveOverflowInterruptStatusOfAllFIFO() function to get all FIFOs with overflow
      // and clear them manually with MCP251XFD_ClearFIFOOverflowEvent() function (for FIFO and TEF)
      // or MCP251XFD_ClearTEFOverflowEvent for the TEF
      break;

    case MCP251XFD_TRANSMIT_ATTEMPT:         // Transmit Attempt Interrupt
      // Only available on INT pin not on INT1 pin
      // Use MCP251XFD_GetTransmitAttemptInterruptStatusOfAllFIFO() function to get all FIFOs with attempt exhaust
      // and clear them manually with MCP251XFD_ClearFIFOAttemptsEvent() function (for FIFO and TXQ)
      // or MCP251XFD_ClearTXQAttemptsEvent for the TXQ
      break;

    case MCP251XFD_ADDRESS_ERROR_INTERRUPT:
      // Address Error Interrupt (illegal FIFO address presented to system)
      MCP251XFD_ClearInterruptEvents(CANEXT1, MCP251XFD_INT_SYSTEM_ERROR_EVENT);
      break;

    case MCP251XFD_RXTX_MAB_OVF_UVF:
      // RX MAB Overflow (RX: message received before previous message was saved to memory)
      MCP251XFD_ClearInterruptEvents(CANEXT1, MCP251XFD_INT_SYSTEM_ERROR_EVENT);
      break;
```

```
      case MCP251XFD_TBC_OVF_INTERRUPT:       // TBC Overflow
        // The time base for the timestamp overflow (32-bits)
        MCP251XFD_ClearInterruptEvents(CANEXT1, MCP251XFD_INT_TIME_BASE_COUNTER_EVENT);
        break;

      case MCP251XFD_OPMODE_CHANGE_OCCURED:    // Operation Mode Change Occurred
        //eMCP251XFD_OperationMode OpMode;
        //MCP251XFD_GetActualOperationMode(CANEXT1, &OpMode);
        MCP251XFD_ClearInterruptEvents(CANEXT1, MCP251XFD_INT_OPERATION_MODE_CHANGE_EVENT);
        break;

      case MCP251XFD_INVALID_MESSAGE_OCCURED: // Invalid Message Occurred
        // Invalid Message Occurred
        MCP251XFD_ClearInterruptEvents(CANEXT1, MCP251XFD_INT_RX_INVALID_MESSAGE_EVENT);
        break;

      case MCP251XFD_TRANSMIT_EVENT_FIFO:     // Transmit Event FIFO Interrupt (TEF)
        // Events on TEF are only available on INT pin not on INT1 pin
        // Use MCP251XFD_GetTEFStatus() function to get its status
        // and MCP251XFD_ReceiveMessageFromFIFO() to get the frame
        break;

      case MCP251XFD_TXQ_INTERRUPT:           // TXQ Interrupt (TFIF<0> set)
      case MCP251XFD_FIFO1_INTERRUPT:         // FIFO 1 Interrupt (TFIF<1> or RFIF<1> set)
        /* ... */
      case MCP251XFD_FIFO31_INTERRUPT:        // FIFO 31 Interrupt (TFIF<31> or RFIF<31> set)
        // If you only have the INT pin an no INT0 or INT1, TXQ and FIFO1 through FIFO31 interrupts can be catch here
        // See §8 and §9 to know what to do here
        break;

      case MCP251XFD_NO_INTERRUPT:            // No interrupt, do nothing
        break;

      default:
        //--- Check others interrupts ---
        setMCP251XFD_InterruptEvents Events;
        MCP251XFD_GetInterruptEvents(CANEXT1, &Events);

        if ((Events & MCP251XFD_INT_BUS_WAKEUP_EVENT) > 0) // Wake-up event (deep sleep state with wakeup from bus)
        {
          // See Sleep §10
          MCP251XFD_ClearInterruptEvents(CANEXT1, MCP251XFD_INT_BUS_WAKEUP_EVENT);
        }

        if ((Events & MCP251XFD_INT_SPI_CRC_EVENT) > 0) // SPI CRC event
        {
          setMCP251XFD_CRCEvents CRCEvent;
          MCP251XFD_GetCRCEvents(CANEXT1, &CRCEvent, NULL);
          if (CRCEvent == MCP251XFD_CRC_CRCERR_EVENT)
              LOGERROR("Ext1: CRC mismatch occurred");
          else LOGERROR("Ext1: Number of Bytes mismatch during 'SPI with CRC' command occurred");
          MCP251XFD_ClearCRCEvents(CANEXT1);
        }

        if ((Events & MCP251XFD_INT_RAM_ECC_EVENT) > 0) // ECC event
        {
          uint16_t AddrError = 0;
          setMCP251XFD_ECCEvents ECCEvent;
          MCP251XFD_GetECCEvents(CANEXT1, &ECCEvent, &AddrError);
          if (ECCEvent == MCP251XFD_ECC_SEC_EVENT)
              /* ECC Single Error was corrected */;
          else /* ECC Double Error was detected*/;
          MCP251XFD_ClearECCEvents(CANEXT1);
        }

        if ((Events & MCP251XFD_INT_BUS_ERROR_EVENT) > 0) // Bus error event
        {
          //MCP251XFD_GetBusDiagnostic();
          //MCP251XFD_ClearBusDiagnostic();
          MCP251XFD_ClearInterruptEvents(CANEXT1, MCP251XFD_INT_BUS_ERROR_EVENT);
        }
        break;
  }
}
```

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

12.   GPIO USAGE

Normally the GPIOs pins are already configured when calling the *Init_MCP251XFD()* function with the MCP251XFD struct and the MCP251XFD_Config struct, but you can change the configuration thereafter by calling the MCP251XFD_ConfigurePins() function.

To change the direction of one or more pins, you need to call the MCP251XFD_SetGPIOPinsDirection() function. This function have a 'pinsChangeMask' parameter to select which pin have to be modified. For both parameters, the bit 0 is the GPIO0 and the bit 1 is the GPIO1.

pinsDirection	pinsChangeMask	GPIO0	GPIO1
-	0b00	No change	No change
0b00	0b01	OUTPUT	No change
0b01	0b01	INPUT	No change
0b10	0b01	OUTPUT	No change
0b11	0b01	INPUT	No change
0b00	0b10	No change	OUTPUT
0b01	0b10	No change	OUTPUT
0b10	0b10	No change	INPUT
0b11	0b10	No change	INPUT
0b00	0b11	OUTPUT	OUTPUT
0b01	0b11	INPUT	OUTPUT
0b10	0b11	OUTPUT	INPUT
0b11	0b11	INPUT	INPUT

The defines indicate in 15.8.2 can be used and OR'ed if necessary.

The MCP251XFD_GetGPIOPinsInputLevel() function return the actual status of all GPIO where a corresponding 'pinsState' status bit '0' is low level and '1' is high level.

To change the output level of one or more pins, you need to call the MCP251XFD_SetGPIOPinsOutputLevel() function. This function have a 'pinsChangeMask' parameter to select which pin have to be modified. For both parameters, the bit 0 is the GPIO0 and the bit 1 is the GPIO1.

pinsLevel	pinsChangeMask	GPIO0	GPIO1
-	0b00	No change	No change
0b00	0b01	LOW	No change
0b01	0b01	HIGH	No change
0b10	0b01	LOW	No change
0b11	0b01	HIGH	No change
0b00	0b10	No change	LOW
0b01	0b10	No change	LOW
0b10	0b10	No change	HIGH
0b11	0b10	No change	HIGH
0b00	0b11	LOW	LOW
0b01	0b11	HIGH	LOW
0b10	0b11	LOW	HIGH
0b11	0b11	HIGH	HIGH

The defines indicate in 15.8.2 can be used and OR'ed if necessary.

13. BITRATES, SPEED, AND TIMING CALCULATION

The purpose of this paragraph is to give some formulas and data to determine the SPI bus load and the CAN bus load in some cases.

13.1. CAN Bitrates

Here are all possible configurations with exact bitrate with a SYSCLK of 20MHz and 40MHz. Here are listed only the normalized bitrates by CAN specification.

13.1.1. Data, characteristics, and specifications

As indicate in the datasheets (Table 7-4), this is the CAN bit rate range:

AC Specifications		Electrical Characteristics: Extended (E): TAMB = –40°C to +125°C; High (H): TAMB = –40°C to +150°C VDD = 2.7V to 5.5V				
Sym	Characteristic	Min	Typ	Max	Units	Conditions/Comments
BRNOM	Nominal Bit Rate	0.125	0.5	1	Mbps	
BRDATA	Data Bit Rate	0.5	2	8	Mbps	BRDATA ≥ BRNOM

Table 2 - CAN bit rate range

These are tested bit rates by Microchip. Device allows the configuration of more bit rates, including slower bit rates than the minimum stated. Lower or higher bitrates than specified are more stable with the MCP2518FD device than the MCP2517FD device. In such case, the device can enter in Restricted Operation Mode and/or have some errors counted in the bus diagnostic registers (see §15.16 for functions to get it).

13.1.2. CAN 2.0 possible CAN Bitrates

Following the Table 2, the normalized possible Nominal Bitrates with exact bitrate are:

20MHz	40MHz
125kbps	125kbps
160kbps	160kbps
200kbps	200kbps
250kbps	250kbps
312.5kbps	312.5kbps
-	320kbps
400kbps	400kbps
500kbps	500kbps
625kbps	625kbps
800kbps	800kbps
1000kbps	1000kbps

Table 3 - CAN2.0 exact Nominal Bitrates

13.1.1. CAN-FD possible CAN Bitrates

Following the Table 2, the normalized possible Nominal and Data Bitrates with exact bitrate are:

20MHz		40MHz	
Nominal Bitrate	Data Bitrate	Nominal Bitrate	Data Bitrate
125 kbps	500kbps, 625kbps, 800kbps, 1Mbps, 1.25Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps	125 kbps	500kbps, 625kbps, 800kbps, 1Mbps, 1.25Mbps, 1.6Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps, 8Mbps
160 kbps	500kbps, 625kbps, 800kbps, 1Mbps, 1.25Mbps, 2Mbps, 2Mbps, 4Mbps, 5Mbps	160 kbps	500kbps, 625kbps, 800kbps, 1Mbps, 1.25Mbps, 1.6Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps, 8Mbps
200 kbps	500kbps, 625kbps, 800kbps, 1Mbps, 1.25Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps	200 kbps	500kbps, 625kbps, 800kbps, 1Mbps, 1.25Mbps, 1.6Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps, 8Mbps

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

250 kbps	500kbps, 625kbps, 800kbps, 1Mbps, 1.25Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps	250 kbps	500kbps, 625kbps, 800kbps, 1Mbps, 1.25Mbps, 1.6Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps, 8Mbps
312,5 kbps	500kbps, 625kbps, 800kbps, 1Mbps, 1.25Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps	312,5 kbps	500kbps, 625kbps, 800kbps, 1Mbps, 1.25Mbps, 1.6Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps, 8Mbps
	-	320 kbps	500kbps, 625kbps, 800kbps, 1Mbps, 1.25Mbps, 1.6Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps, 8Mbps
400 kbps	500kbps, 625kbps, 800kbps, 1Mbps, 1.25Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps	400 kbps	500kbps, 625kbps, 800kbps, 1Mbps, 1.25Mbps, 1.6Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps, 8Mbps
500 kbps	500kbps, 625kbps, 800kbps, 1Mbps, 1.25Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps	500 kbps	500kbps, 625kbps, 800kbps, 1Mbps, 1.25Mbps, 1.6Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps, 8Mbps
625 kbps	625kbps, 800kbps, 1Mbps, 1.25Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps	625 kbps	625kbps, 800kbps, 1Mbps, 1.25Mbps, 1.6Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps, 8Mbps
800 kbps	800kbps, 1Mbps, 1.25Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps	800 kbps	800kbps, 1Mbps, 1.25Mbps, 1.6Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps, 8Mbps
1000 kbps	1Mbps, 1.25Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps	1000 kbps	1Mbps, 1.25Mbps, 1.6Mbps, 2Mbps, 2.5Mbps, 4Mbps, 5Mbps, 8Mbps

Table 4 - CAN-FD exact Nominal and Data Bitrates

13.2. CAN Nominal and Data speed frames count

CAN 2.0 frame speed use only the Nominal bit time. CAN-FD frame speed use both Nominal and Data bit time.

A CAN frame begin with 1 bit of Start-Of-Frame, followed with an Arbitration Field. Next is the Control Field. After that the Data Field which is variable. At the end is the CRC Field followed with End-Of-Frame. A minimum of 3 bit is necessary between frames.

CAN protocol use a Non-Return-Zero. This force a bit change every 5 consecutive bits of the same polarity. This bit stuffing begins with the Start-Of-Frame. It ends on the last bit of the CRC Field for the CAN2.0 frames and the CAN-FD NON-ISO frames. It ends on the last bit of the Data Field for the CAN-FD ISO frames.

13.2.1. CAN2.0A - Base Data Frames

A CAN2.0A frame is constituted by:
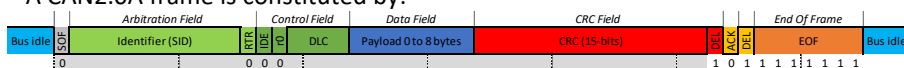
The minimum bit count to send a frame is: $MinCAN20ABitCount = 44 + 8 \times DLC$. But with bit stuffing the minimum is:

$$MinCAN20ABitCount = 44 + 8 \times DLC + 1 \qquad (1.5)$$

This is because there is a bit stuffing in the Control Field that is impossible to skip.

The maximum possible frame with 0-byte DLC at 1Mbit/s Nominal Bit Rate is: $\frac{1000000}{45+3} = 20833 \ frame \ per \ second$.

The maximum possible frame with 8-byte DLC at 1Mbit/s Nominal Bit Rate is: $\frac{1000000}{109+3} = 8928 \ frame \ per \ second$.

Here are some real data for the minimum bit count:

DLC	Min theorical	Min real	Frame example
0	45	45	SID: 0x084; RTR: 0; IDE: 0; r0: 0; DLC: 0
1	53	53	SID: 0x085; RTR: 0; IDE: 0; r0: 0; DLC: 1; Data0: 0x08
2	61	61	SID: 0x089; RTR: 0; IDE: 0; r0: 0; DLC: 2; Data0: 0x11, Data1: 0x08
3	69	69	SID: 0x084; RTR: 0; IDE: 0; r0: 0; DLC: 3; Data0: 0x09, Data1: 0x09, Data2: 0x08
4	77	76	SID: 0x085; RTR: 0; IDE: 0; r0: 0; DLC: 4; Data0: 0x21, Data1: 0x09, Data2: 0x09, Data3: 0x08
5	85	?	
6	93	?	
7	101	?	
8	109	?	

The maximum theorical bit count to send a frame is:

$$MaxCAN20ABitCount = (44 + 8 \times DLC) + \frac{34 + 8 \times DLC - 1}{4} - 3 \qquad (1.6)$$

Here are some real data for the maximum bit count:

DLC	Max theorical	Max real	Frame example
0	49	50	SID: 0x000; RTR: 0; IDE: 0; r0: 0; DLC: 0
1	59	59	SID: 0x078; RTR: 0; IDE: 0; r0: 0; DLC: 1; Data0: 0x01
2	69	69	SID: 0x041; RTR: 0; IDE: 0; r0: 0; DLC: 2; Data0: 0x04, Data1: 0x00
3	79	80	SID: 0x41F; RTR: 0; IDE: 0; r0: 0; DLC: 3; Data0: 0xDF, Data1: 0x0F, Data2: 0x08
4	89	90	SID: 0x03E; RTR: 0; IDE: 0; r0: 0; DLC: 4; Data0: 0x10, Data1: 0x78, Data2: 0x03, Data3: 0xFE
5	99	?	
6	109	?	
7	119	?	
8	129	?	

13.2.2. CAN2.0B - Extended Data Frames

A CAN2.0B frame is constituted by:
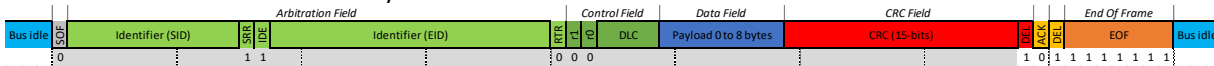
The minimum bit count to send a frame is: $MinCAN20BBitCount = 64 + 8 \times DLC$. But with bit stuffing the minimum is:

$$MinCAN20BBitCount = 64 + 8 \times DLC + 1 \qquad (1.7)$$

The maximum possible frame with 0-byte DLC at 1Mbit/s Nominal Bit Rate is: $\frac{1000000}{65+3} = 14705 \; frame \; per \; second$.

The maximum possible frame with 8-byte DLC at 1Mbit/s Nominal Bit Rate is: $\frac{1000000}{129+3} = 7575 \; frame \; per \; second$.

This is because there is a bit stuffing in the Control Field that is impossible to skip.

Here are some real data for the minimum bit count:

DLC	Min theorical	Min real	Frame example
0	65	65	SID: 0x084; EID: 0x02109; RTR: 0; IDE: 1; r1: 0; r0: 0; DLC: 0
1	73	73	SID: 0x086; EID: 0x02108; RTR: 0; IDE: 1; r1: 0; r0: 0; DLC: 1; Data0: 0x08
2	81	?	
3	89	?	
4	97	?	
5	105	?	
6	113	?	
7	121	?	
8	129	?	

The maximum theorical bit count to send a frame is:

$$MaxCAN20BBitCount = (64 + 8 \times DLC) + \frac{54 + 8 \times DLC - 1}{4} - 3 \qquad (1.8)$$

Here are some real data for the maximum bit count:

DLC	Max theorical	Max real	Frame example
0	74	75	SID: 0x3E1; EID: 0x21FF8; RTR: 0; IDE: 1; r1: 0; r0: 0; DLC: 0
1	84	85	SID: 0x7C3; EID: 0x03FF0; RTR: 0; IDE: 1; r1: 0; r0: 0; DLC: 1; Data: 0xF0
2	94	?	
3	104	?	
4	114	?	
5	124	?	
6	134	?	
7	144	?	
8	154	?	

13.2.3. CAN-FD ISO - Base Data Frames (Up to 16 bytes)

A CAN-FD ISO base frame is constituted by:
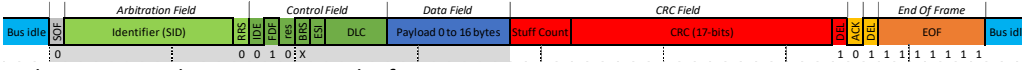
The minimum bit count to send a frame is:

$$MinBaseCANFDISOBitCount = 59 + 8 \times DLC + 1 \tag{1.9}$$

The maximum possible frame with 0-byte DLC at 1Mbit/s Nominal Bit Rate and 8Mbit/s Data Bit Rate is:

$$\frac{1s}{\frac{26+3}{1000000} + \frac{33}{8000000}} = 30188 \ frame \ per \ second.$$

The maximum possible frame with 16-byte DLC at 1Mbit/s Nominal Bit Rate and 8Mbit/s Data Bit Rate is:

$$\frac{1s}{\frac{26+3}{1000000} + \frac{33+16\times8}{8000000}} = 20356 \ frame \ per \ second.$$

The maximum theorical bit count to send a frame is:

$$MaxBaseCANFDISOBitCount = (59 + 8 \times DLC) + \frac{22+8\times DLC-1}{4} - 3 \tag{1.10}$$

13.2.4. CAN-FD ISO - Base Data Frames (17 to 64 bytes)

A CAN-FD ISO base frame is constituted by:
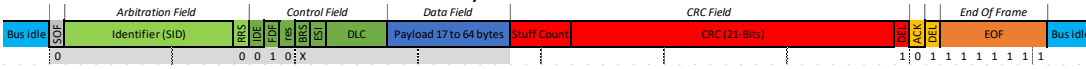
The minimum bit count to send a frame is:

$$MinBaseCANFDISOBitCount = 64 + 8 \times DLC + 1 \tag{1.11}$$

The maximum possible frame with 20-byte DLC at 1Mbit/s Nominal Bit Rate and 8Mbit/s Data Bit Rate is:

$$\frac{1s}{\frac{26+3}{1000000} + \frac{38+20\times8}{8000000}} = 18604 \ frame \ per \ second.$$

The maximum possible frame with 64-byte DLC at 1Mbit/s Nominal Bit Rate and 8Mbit/s Data Bit Rate is:

$$\frac{1s}{\frac{26+3}{1000000} + \frac{38+64\times8}{8000000}} = 10230 \ frame \ per \ second.$$

The maximum theorical bit count to send a frame is:

$$MaxBaseCANFDISOBitCount = (64 + 8 \times DLC) + \frac{22+8\times DLC-1}{4} - 3 \tag{1.12}$$

13.2.5. CAN-FD ISO - Extended Data Frames (Up to 16 bytes)

A CAN-FD ISO extended frame is constituted by:

The minimum bit count to send a frame is:

$$MinExtendedCANFDISOBitCount = 78 + 8 \times DLC + 1 \tag{1.13}$$

The maximum possible frame with 0-byte DLC at 1Mbit/s Nominal Bit Rate and 8Mbit/s Data Bit Rate is:

$$\frac{1s}{\frac{45+3}{1000000} + \frac{33}{8000000}} = 19184 \ frame \ per \ second.$$

The maximum possible frame with 16-byte DLC at 1Mbit/s Nominal Bit Rate and 8Mbit/s Data Bit Rate is:

$$\frac{1s}{\frac{45+3}{1000000} + \frac{33+16\times8}{8000000}} = 14678 \ frame \ per \ second.$$

The maximum theorical bit count to send a frame is:

$$MaxExtendedCANFDISOBitCount = (78 + 8 \times DLC) + \frac{41+8\times DLC-1}{4} - 3 \tag{1.14}$$

13.2.6. CAN-FD ISO - Extended Data Frames (17 to 64 bytes)

A CAN-FD ISO extended frame is constituted by:
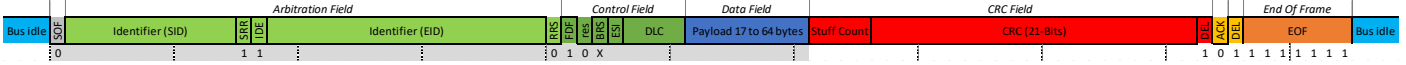
The minimum bit count to send a frame is:

$$MinExtendedCAN\text{FD}ISOBitCount = 83 + 8 \times DLC + 1 \tag{1.15}$$

The maximum possible frame with 20-byte DLC at 1Mbit/s Nominal Bit Rate and 8Mbit/s Data Bit Rate is:

$$\frac{1s}{\frac{45+3}{1000000} + \frac{38+20 \times 8}{8000000}} = 13745 \; frame \; per \; second.$$

The maximum possible frame with 64-byte DLC at 1Mbit/s Nominal Bit Rate and 8Mbit/s Data Bit Rate is:

$$\frac{1s}{\frac{45+3}{1000000} + \frac{38+64 \times 8}{8000000}} = 8565 \; frame \; per \; second.$$

The maximum theorical bit count to send a frame is:

$$MaxExtendedCAN\text{FD}ISOBitCount = (83 + 8 \times DLC) + \frac{41+8 \times DLC-1}{4} - 3 \tag{1.16}$$

13.2.7. CAN-FD NON-ISO - Base Data Frames (Up to 16 bytes)

A CAN-FD NON-ISO base frame is constituted by:
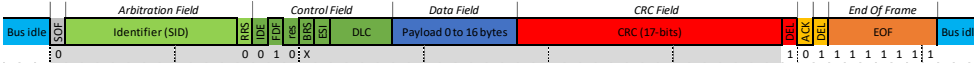
The minimum bit count to send a frame is:

$$MinBaseCAN\text{FD}NONISOBitCount = 49 + 8 \times DLC + 1 \tag{1.17}$$

The maximum possible frame with 0-byte DLC at 1Mbit/s Nominal Bit Rate and 8Mbit/s Data Bit Rate is:

$$\frac{1s}{\frac{26+3}{1000000} + \frac{23}{8000000}} = 31372 \; frame \; per \; second.$$

The maximum possible frame with 16-byte DLC at 1Mbit/s Nominal Bit Rate and 8Mbit/s Data Bit Rate is:

$$\frac{1s}{\frac{26+3}{1000000} + \frac{23+16 \times 8}{8000000}} = 20887 \; frame \; per \; second.$$

The maximum theorical bit count to send a frame is:

$$MaxBaseCAN\text{FD}NONISOBitCount = (49 + 8 \times DLC) + \frac{39+8 \times DLC-1}{4} - 3 \tag{1.18}$$

13.2.8. CAN-FD NON-ISO - Base Data Frames (17 to 64 bytes)

A CAN-FD NON-ISO base frame is constituted by:
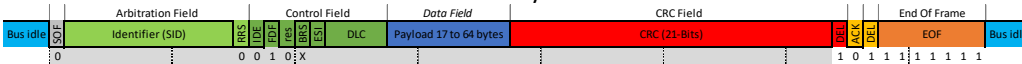
The minimum bit count to send a frame is:

$$MinBaseCAN\text{FD}NONISOBitCount = 53 + 8 \times DLC + 1 \tag{1.19}$$

The maximum possible frame with 0-byte DLC at 1Mbit/s Nominal Bit Rate and 8Mbit/s Data Bit Rate is:

$$\frac{1s}{\frac{26+3}{1000000} + \frac{27+20 \times 8}{8000000}} = 19093 \; frame \; per \; second.$$

The maximum possible frame with 16-byte DLC at 1Mbit/s Nominal Bit Rate and 8Mbit/s Data Bit Rate is:

$$\frac{1s}{\frac{26+3}{1000000} + \frac{27+64 \times 8}{8000000}} = 10376 \; frame \; per \; second.$$

The maximum theorical bit count to send a frame is:

$$MaxBaseCAN\text{FD}NONISOBitCount = (53 + 8 \times DLC) + \frac{43+8 \times DLC-1}{4} - 3 \tag{1.20}$$

13.2.9. CAN-FD NON-ISO - Extended Data Frames (Up to 16 bytes)

A CAN-FD NON-ISO extended frame is constituted by:
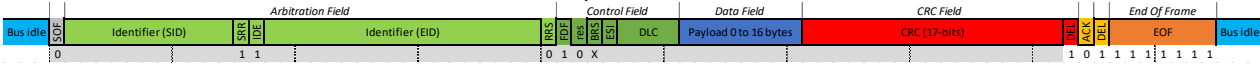
The minimum bit count to send a frame is:

$$MinExtendedCANFDNONISOBitCount = 72 + 8 \times DLC + 1 \tag{1.21}$$

The maximum possible frame with 0-byte DLC at 1Mbit/s Nominal Bit Rate and 8Mbit/s Data Bit Rate is:

$$\frac{1s}{\frac{45+3}{1000000} + \frac{27}{8000000}} = 19464 \; frame \; per \; second.$$

The maximum possible frame with 16-byte DLC at 1Mbit/s Nominal Bit Rate and 8Mbit/s Data Bit Rate is:

$$\frac{1s}{\frac{45+3}{1000000} + \frac{27+16 \times 8}{8000000}} = 14842 \; frame \; per \; second.$$

The maximum theorical bit count to send a frame is:

$$MaxExtendedCANFDNONISOBitCount = (72 + 8 \times DLC) + \frac{62 + 8 \times DLC - 1}{4} - 3 \tag{1.22}$$

13.2.10.   CAN-FD NON-ISO - Extended Data Frames (17 to 64 bytes)

A CAN-FD NON-ISO extended frame is constituted by:
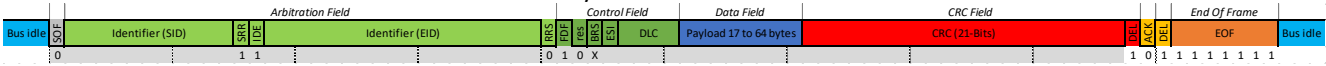
The minimum bit count to send a frame is:

$$MinExtendedCANFDNONISOBitCount = 76 + 8 \times DLC + 1 \tag{1.23}$$

The maximum possible frame with 0-byte DLC at 1Mbit/s Nominal Bit Rate and 8Mbit/s Data Bit Rate is:

$$\frac{1s}{\frac{45+3}{1000000} + \frac{31+20 \times 8}{8000000}} = 13913 \; frame \; per \; second.$$

The maximum possible frame with 16-byte DLC at 1Mbit/s Nominal Bit Rate and 8Mbit/s Data Bit Rate is:
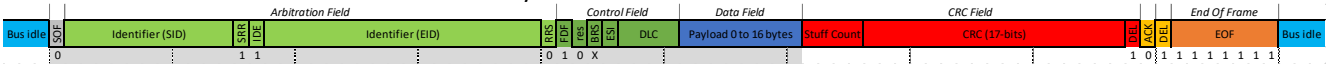
$$\frac{1s}{\frac{45+3}{1000000} + \frac{31+64 \times 8}{8000000}} = 8630 \; frame \; per \; second.$$

The maximum theorical bit count to send a frame is:

$$MaxExtendedCANFDNONISOBitCount = (76 + 8 \times DLC) + \frac{66 + 8 \times DLC - 1}{4} - 3 \tag{1.24}$$

Can be widely distributed

Guide / MCP251XFD driver library

© Copyright 2020 Fabien MAILLY – All rights reserved

Page 32 of 112

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

13.3.   Communication timings with the controller

Here will be indicate all communication timings through the SPI interface when sending a message, configure the device or simply change a GPIO state. All timings indicate in this section are for a direct communication with the component through an SPI interface and therefore no component in the middle.

Why this section? Because in a synchronous system, the MCU will be blocked for the time the SPI communication works and a little more. With these metrics, you will be able to determinate how much time the communication with the device will take and therefore how much time remains for the rest of your program. In an asynchronous system, it will only indicate how much time the SPI interface will be busy.

13.3.1. Data, characteristics, and specifications

As indicate in the datasheets (Table 7-6), this is the SPI AC characteristics:

AC Specifications			Electrical Characteristics: Extended (E): TAMB = −40°C to +125°C High (H): TAMB = −40°C to +150°C; VDD = 2.7V to 5.5V				
Param.	Sym	Characteristic	Min	Typ	Max	Units	Conditions/Comments
	F_{SCK}	SCK Input Frequency	—	—	20	MHz	F_{SCK} must be less than or equal to $F_{SYSCLK}/2$
	T_{SCK}	SCK Period, TSCK=1/FSCK	50	—	—	ns	F_{SCK} must be less than or equal to $F_{SYSCLK}/2$
1	T_{SCKH}	SCK High Time	20	—	—	ns	
2	T_{SCKL}	SCK Low Time	20	—	—	ns	
3	T_{SCKR}	SCK Rise Time	—	—	100	ns	Design guidance only
4	T_{SCKF}	SCK Fall Time	—	—	100	ns	Design guidance only
5	T_{CS2SCK}	nCS ↓ to SCK ↑	$T_{SCK}/2$	—	—	ns	
6	T_{SCK2CS}	SCK ↑ to nCS ↑	T_{SCK}	—	—	ns	
7	$T_{SDI2SCK}$	SDI Setup: SDI ↕ to SCK ↑	5	—	—	ns	
8	$T_{SCK2SDI}$	SDI Hold: SCK ↑ to SDI ↕	5	—	—	ns	
9	$T_{SCK2SDO}$	SDO Valid: SCK ↓ to SDO ↕	—	—	20	ns	C_{LOAD} = 50 pF
10	$T_{CS2SDOZ}$	SDO High Z: nCS ↑ to SDO Z	—	—	2 T_{SCK}	ns	C_{LOAD} = 50 pF
11	T_{CSD}	nCS ↑ to nCS ↓	T_{SCK}	—	—	ns	Design guidance only

Table 5 - SPI timings

This information is linked to Figure 3 - SPI I/O Timing.

Below is the SPI I/O timing which is extract from datasheet (Figure 7-1):
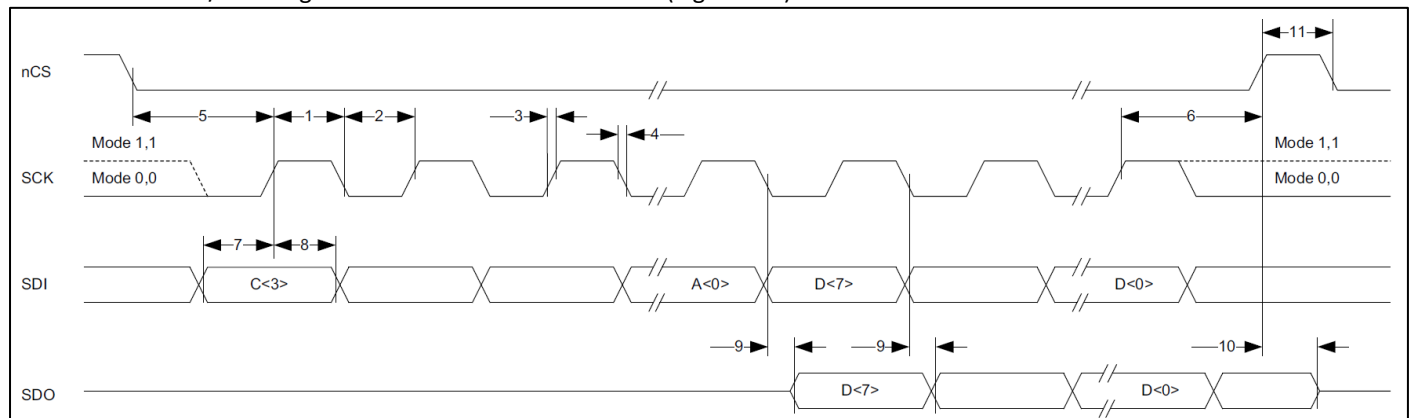
Figure 3 - SPI I/O Timing

Below is the number of bytes that are sent according to the command:

	Driver configuration	Byte count
Normal read	eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_NORMAL_USE	2 + Data
Normal write	eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_NORMAL_USE	2 + Data
Read with CRC	eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_USE_READ_WRITE_CRC	5 + Data
Write with CRC	eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_USE_READ_WRITE_CRC	5 + Data
Safe write	eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_USE_SAFE_WRITE	4 + Data

Table 6 - Byte count according to driver configuration

Ref: MCP251XFD driver library guide (Synchronous driver).docx

According to the Errata and Data Sheet Clarification (DS80000792B page 1 with Figure 1 and Table 1), the SPI communication can block the CAN FD controller module when the communication between bytes takes too long time in case of SPI Read or SPI Read CRC. The maximum time between bytes and time at end of transmission depends on the Nominal and Data bit time:
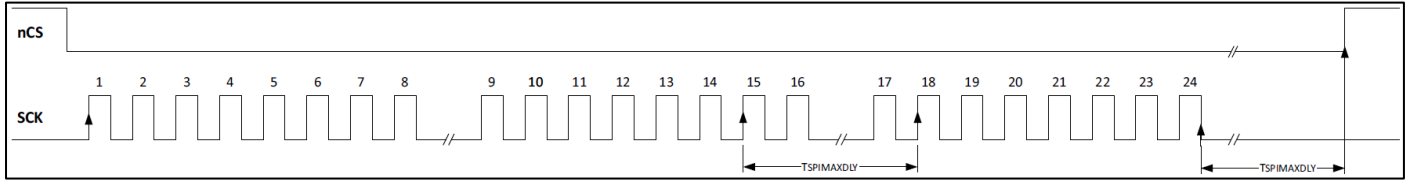
Table 7 - Maximum delay between SPI bytes

Scenario	Frame Format	$T_{SPIMAXDLY}$
1	CAN Base Frame	5 NBT (Nominal bit time)
2	CAN FD Control Field	3 NBT (Nominal bit time) + 5 DBT (Data bit time)
3	CAN FD Data Phase	32 DBT (Data bit time)

Table 8 - Worst-case scenarios

13.3.2. General formula to access SFR or RAM register

With a normal driver communication (eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_NORMAL_USE), the communication takes 2 control bytes. The address is internally incremented. The time to read x bytes from an SFR register or write x bytes to an SFR register takes:

- Time for x byte: $\qquad T_{ReadXBytes} = T_{WriteXBytes} = T_{CS2SCK} + (2 + x) \times 8T_{SCK} + (1 + x) \times T_{BetweenBytes} + T_{SCK2CS}$ (2.1)
- Minimum time: $\qquad T_{ReadXBytesMin} = T_{WriteXBytesMin} = \frac{T_{SCK}}{2} + (2 + x) \times 8T_{SCK} + T_{SCK}$ (2.2)
- Maximum time: $\qquad T_{ReadXBytesMax} = T_{CS2SCK} + (2 + x) \times 8T_{SCK} + (1 + x) \times (T_{SPIMAXDLY} - 3T_{SCK}) + T_{SPIMAXDLY}$ (2.3)

With a CRC communication (eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_USE_READ_WRITE_CRC), the communication takes 5 control bytes. The address is internally incremented. The time to read x bytes from an SFR register or write x bytes to an SFR register takes:

- Time for x byte: $\qquad T_{CRCReadXBytes} = T_{CRCWriteXBytes} = T_{CS2SCK} + (5 + x) \times 8T_{SCK} + (4 + x) \times T_{BetweenBytes} + T_{SCK2CS}$ (2.4)
- Minimum time: $\qquad T_{CRCReadXBytesMin} = T_{CRCWriteXBytesMin} = \frac{T_{SCK}}{2} + (5 + x) \times 8T_{SCK} + T_{SCK}$ (2.5)
- Maximum time: $\qquad T_{CRCReadXBytesMax} = T_{CS2SCK} + (5 + x) \times 8T_{SCK} + (4 + x) \times (T_{SPIMAXDLY} - 3T_{SCK}) + T_{SPIMAXDLY}$ (2.6)

With a Safe Write communication (eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_USE_SAFE_WRITE), the communication takes 4 control bytes + the data byte. The time to write 1 byte to an SFR register or 4 bytes to a RAM register takes:

- Time for x byte:
$$T_{SafeWriteXBytes} = \frac{x}{\begin{cases} 1 \ if \ SFR \ register \\ 4 \ if \ RAM \ register \end{cases}} \times \left(T_{CS2SCK} + \left(4 + \begin{cases} 1 \ if \ SFR \ register \\ 4 \ if \ RAM \ register \end{cases}\right) \times 8T_{SCK} \right.$$
$$\left. + \left(3 + \begin{cases} 1 \ if \ SFR \ register \\ 4 \ if \ RAM \ register \end{cases}\right) \times T_{BetweenBytes} + T_{SCK2CS} \right) \qquad (2.7)$$

- Minimum time:
$$T_{SafeWriteXBytesMin} = \frac{x}{\begin{cases} 1 \ if \ SFR \ register \\ 4 \ if \ RAM \ register \end{cases}} \times \left(\frac{T_{SCK}}{2} + \left(4 + \begin{cases} 1 \ if \ SFR \ register \\ 4 \ if \ RAM \ register \end{cases}\right) \times 8T_{SCK} + T_{SCK} \right) \qquad (2.8)$$

- Maximum time:
$$T_{SafeWriteXBytesMax} = \frac{x}{\begin{cases} 1 \ if \ SFR \ register \\ 4 \ if \ RAM \ register \end{cases}} \times \left(T_{CS2SCK} + \left(4 + \begin{cases} 1 \ if \ SFR \ register \\ 4 \ if \ RAM \ register \end{cases}\right) \times 8T_{SCK} \right.$$
$$\left. + \left(3 + \begin{cases} 1 \ if \ SFR \ register \\ 4 \ if \ RAM \ register \end{cases}\right) \times (T_{SPIMAXDLY} - 3T_{SCK}) + T_{SPIMAXDLY} \right) \qquad (2.9)$$

13.3.3. Time to toggle GPIO

The toggle of GPIO need to address only one byte of SFR register. In this case only count formula (2.1), (2.4) or (2.7) depending to the driver configuration.

The maximum possible speed to toggle GPIO is with eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_NORMAL_USE and 20MHz SPI clock. For this, use of the formula (2.2):

$$T_{MinToggleGPIOTime} = \frac{T_{SCK}}{2} + (2 + 1) \times 8T_{SCK} + T_{SCK} = \frac{\frac{1}{20MHz}}{2} + 25 \times \frac{1}{20MHz} = 25ns + 25 \times 50ns = 1275ns = 1.275\mu s$$

Ref: MCP251XFD driver library guide (Synchronous driver).docx

13.3.4. Time to get one CAN2.0 frame

For this, there is some checks to do before retrieving the frame like the message address in RAM who takes a read of 4-byte SFR. And after, get the frame who takes 8 bytes plus 8 bytes for the payload plus 4 bytes for the time stamp if stored by the FIFO to retrieve from RAM. And to finish, update the tail of the FIFO who takes 1-byte SFR to write. These actions are mandatory for the driver for each message. If the application does not use the Rx interrupt pin (INT1), it takes 1-byte SFR more.

If there are more than one FIFO, it takes more data to read to get the list where a message is available.

13.3.4.1. With a Normal driver configuration

With a normal driver communication (eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_NORMAL_USE), the time to get one message with TimeStamp and 8-bytes of payload takes:
- Time for 1 frame + TS: $T_{Read2.0Frame+TS} = T_{Read4Bytes} + T_{Read20Bytes} + T_{Write1Bytes}$ (3.1)
- Minimum time: $T_{Read2.0Frame+TSMin} = T_{Read4BytesMin} + T_{Read20BytesMin} + T_{Write1BytesMin}$ (3.2)

The minimum possible time to get a CAN 2.0 frame with 8-bytes of payload + TimeStamp and a 20MHz SPI clock. For this, use the formula (3.2):

$T_{MinRead2.0Frame8BytesWithTimeStampTime}$

$$= \left(\frac{T_{SCK}}{2} + (2+4) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (2+20) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (2+1) \times 8T_{SCK} + T_{SCK}\right)$$

$$= \left(\frac{\frac{1}{20MHz}}{2} + 49 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 177 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 25 \times \frac{1}{20MHz}\right)$$

$$= (25ns + 49 \times 50ns) + (25ns + 177 \times 50ns) + (25ns + 25 \times 50ns) = 2475ns + 8875ns + 1275ns$$

$$= 12625ns = 12.625\mu s$$

This means 79207 frames per second.

The maximum possible CAN2.0A frames on a CAN bus is 8928 frames which means at full capability, the MCU will be blocked at maximum approximatively 11.2% of its execution time.

The maximum possible CAN2.0B frames on a CAN bus is 7575 frames which means at full capability, the MCU will be blocked at maximum approximatively 9.6% of its execution time.

13.3.4.2. With a CRC communication driver configuration

With a CRC communication (eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_USE_READ_WRITE_CRC), the time to get one message with TimeStamp and 8-bytes of payload takes:
- Time for 1 frame + TS: $T_{CRCRead2.0Frame+TS} = T_{CRCRead4Bytes} + T_{CRCRead20Bytes} + T_{CRCWrite1Bytes}$ (3.3)
- Minimum time: $T_{CRCRead2.0Frame+TSMin} = T_{CRCRead4BytesMin} + T_{CRCRead20BytesMin} + T_{CRCWrite1BytesMin}$ (3.4)

The minimum possible time to get a CAN 2.0 frame with 8-bytes of payload + TimeStamp and a 20MHz SPI clock. For this, use the formula (3.4):

$T_{MinRead2.0Frame8BytesWithTimeStampTime}$

$$= \left(\frac{T_{SCK}}{2} + (5+4) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (5+20) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (5+1) \times 8T_{SCK} + T_{SCK}\right)$$

$$= \left(\frac{\frac{1}{20MHz}}{2} + 73 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 201 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 49 \times \frac{1}{20MHz}\right)$$

$$= (25ns + 73 \times 50ns) + (25ns + 201 \times 50ns) + (25ns + 49 \times 50ns) = 3675ns + 10075ns + 2475ns$$

$$= 16225ns = 16.225\mu s$$

This means 61633 frames per second.

The maximum possible CAN2.0A frames on a CAN bus is 8928 frames which means at full capability, the MCU will be blocked at maximum approximatively 14.4% of its execution time.

The maximum possible CAN2.0B frames on a CAN bus is 7575 frames which means at full capability, the MCU will be blocked at maximum approximatively 12.3% of its execution time.

13.3.5. Time to send one CAN2.0 frame

For this, there is some checks to do before sending the frame like the message address in RAM who takes a read of 4-byte SFR. And after, send of the frame who takes 8 bytes plus 8 bytes for the payload to send to RAM. And to finish, update the head of the FIFO who takes 1-byte SFR to write. These actions are mandatory for the driver for each message. If the application does not use the Tx interrupt pin (INT0), it takes 1-byte SFR more.

If there are more than one FIFO, it takes more data to read to get the list where a message room is available.

13.3.5.1. With a Normal driver configuration

With a normal driver communication (eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_NORMAL_USE), the time to send one message and 8-bytes of payload takes:

- Time for 1 frame:

$$T_{Write2.0Frame} = T_{Read4Bytes} + T_{Write16Bytes} + T_{Write1Bytes} \quad (3.5)$$

- Minimum time:

$$T_{Write2.0FrameMin} = T_{Read4BytesMin} + T_{Write16BytesMin} + T_{Write1BytesMin} \quad (3.6)$$

The minimum possible time to send a CAN 2.0 frame with 8-bytes of payload and a 20MHz SPI clock. For this, use the formula (3.6):

$$T_{MinWrite2.0Frame8BytesTime}$$

$$= \left(\frac{T_{SCK}}{2} + (2+4) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (2+16) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (2+1) \times 8T_{SCK} + T_{SCK}\right)$$

$$= \left(\frac{\frac{1}{20MHz}}{2} + 49 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 145 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 25 \times \frac{1}{20MHz}\right)$$

$$= (25ns + 49 \times 50ns) + (25ns + 145 \times 50ns) + (25ns + 25 \times 50ns) = 2475ns + 7275ns + 1275ns$$

$$= 11025ns = 11.025\mu s$$

This means 90702 frames per second.

The maximum possible CAN2.0A frames on a CAN bus is 8928 frames which means at full capability, the MCU will be blocked at maximum approximatively 9.8% of its execution time.

The maximum possible CAN2.0B frames on a CAN bus is 7575 frames which means at full capability, the MCU will be blocked at maximum approximatively 8.4% of its execution time.

13.3.5.2. With a CRC communication driver configuration

With a CRC communication (eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_USE_READ_WRITE_CRC), the time to send one message and 8-bytes of payload takes:

- Time for 1 frame:

$$T_{CRCWrite2.0Frame} = T_{CRCRead4Bytes} + T_{CRCWrite16Bytes} + T_{CRCWrite1Bytes} \quad (3.7)$$

- Minimum time:

$$T_{CRCWrite2.0FrameMin} = T_{CRCRead4BytesMin} + T_{CRCWrite16BytesMin} + T_{CRCWrite1BytesMin} \quad (3.8)$$

The minimum possible time to send a CAN 2.0 frame with 8-bytes of payload and a 20MHz SPI clock. For this, use the formula (3.8):

$$T_{MinWrite2.0Frame8BytesTime}$$

$$= \left(\frac{T_{SCK}}{2} + (5+4) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (5+16) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (5+1) \times 8T_{SCK} + T_{SCK}\right)$$

$$= \left(\frac{\frac{1}{20MHz}}{2} + 73 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 169 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 49 \times \frac{1}{20MHz}\right)$$

$$= (25ns + 73 \times 50ns) + (25ns + 169 \times 50ns) + (25ns + 49 \times 50ns) = 3675ns + 8475ns + 2475ns$$

$$= 14625ns = 14.625\mu s$$

This means 68376 frames per second.

The maximum possible CAN2.0A frames on a CAN bus is 8928 frames which means at full capability, the MCU will be blocked at maximum approximatively 13.1% of its execution time.

The maximum possible CAN2.0B frames on a CAN bus is 7575 frames which means at full capability, the MCU will be blocked at maximum approximatively 11.1% of its execution time.

Can be widely distributed
Guide / **MCP251XFD driver library**
Page 36 of 112

13.3.5.3. With a Safe Write communication driver configuration

With a CRC communication (eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_USE_SAFE_WRITE), the time to send one message and 8-bytes of payload takes:
- Time for 1 frame: $T_{SafeWrite2.0Frame} = T_{CRCRead4Bytes} + T_{SafeWrite16Bytes} + T_{SafeWrite1Bytes}$ (3.9)
- Minimum time: $T_{SafeWrite2.0FrameMin} = T_{CRCRead4BytesMin} + T_{SafeWrite16BytesMin} + T_{SafeWrite1BytesMin}$ (3.10)

The minimum possible time to send a CAN 2.0 frame with 8-bytes of payload and a 20MHz SPI clock. For this, use the formula (3.10):

$$T_{MinWrite2.0Frame8BytesTime}$$

$$= \left(\frac{T_{SCK}}{2} + (5+4) \times 8T_{SCK} + T_{SCK}\right) + \frac{16}{4} \times \left(\frac{T_{SCK}}{2} + (4+4) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (4+1) \times 8T_{SCK} + T_{SCK}\right)$$

$$= \left(\frac{\frac{1}{20MHz}}{2} + 73 \times \frac{1}{20MHz}\right) + 4 \times \left(\frac{\frac{1}{20MHz}}{2} + 65 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 41 \times \frac{1}{20MHz}\right)$$

$$= (25ns + 73 \times 50ns) + 4 \times (25ns + 65 \times 50ns) + (25ns + 41 \times 50ns) = 3675ns + 13100ns + 2075ns$$

$$= 18850ns = 18.85\mu s$$

This means 53050 frames per second.

The maximum possible CAN2.0A frames on a CAN bus is 8928 frames which means at full capability, the MCU will be blocked at maximum approximatively 16.8% of its execution time.

The maximum possible CAN2.0B frames on a CAN bus is 7575 frames which means at full capability, the MCU will be blocked at maximum approximatively 14.3% of its execution time.

13.3.6. Time to get a full CAN-FD frame

For this, there is some checks to do before retrieving the frame like the message address in RAM who takes a read of 4-byte SFR. And after, get the frame who takes 8 bytes plus 64 bytes for the payload plus 4 bytes for the time stamp if stored by the FIFO to retrieve from RAM. And to finish, update the tail of the FIFO who takes 1-byte SFR to write. These actions are mandatory for the driver for each message. If the application does not use the Rx interrupt pin (INT1), it takes 1-byte SFR more.

If there are more than one FIFO, it takes more data to read to get the list where a message is available.

13.3.6.1. With a Normal driver configuration

With a normal driver communication (eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_NORMAL_USE), the time to get one message with TimeStamp and 64-bytes of payload takes:

- Time for 1 frame + TS: $T_{ReadFDFrame+TS} = T_{Read4Bytes} + T_{Read76Bytes} + T_{Write1Bytes}$ (3.11)
- Minimum time: $T_{ReadFDFrame+TSMin} = T_{Read4BytesMin} + T_{Read76BytesMin} + T_{Write1BytesMin}$ (3.12)

The minimum possible time to get a CAN-FD frame with 64-bytes of payload + TimeStamp and a 20MHz SPI clock. For this, use the formula (3.12):

$T_{MinRead2FDFrame64BytesWithTimeStampTime}$

$$= \left(\frac{T_{SCK}}{2} + (2+4) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (2+76) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (2+1) \times 8T_{SCK} + T_{SCK}\right)$$

$$= \left(\frac{\frac{1}{20MHz}}{2} + 49 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 625 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 25 \times \frac{1}{20MHz}\right)$$

$$= (25ns + 49 \times 50ns) + (25ns + 625 \times 50ns) + (25ns + 25 \times 50ns) = 2475ns + 31225ns + 1275ns$$

$$= 34975ns = 34.975\mu s$$

This means 28591 frames per second.

The maximum possible frames on a Base CAN-FD bus is 10230 frames which means at full capability, the MCU will be blocked at maximum approximatively 35.8% of its execution time.

The maximum possible frames on an Extended CAN-FD bus is 8565 frames which means at full capability, the MCU will be blocked at maximum approximatively 30.0% of its execution time.

13.3.6.2. With a CRC communication driver configuration

With a CRC communication (eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_USE_READ_WRITE_CRC), the time to get one message with TimeStamp and 64-bytes of payload takes:

- Time for 1 frame + TS: $T_{CRCReadFDFrame+TS} = T_{CRCRead4Bytes} + T_{CRCRead76Bytes} + T_{CRCWrite1Bytes}$ (3.13)
- Minimum time: $T_{CRCReadFDFrame+TSMin} = T_{CRCRead4BytesMin} + T_{CRCRead76BytesMin} + T_{CRCWrite1BytesMin}$ (3.14)

The minimum possible time to get a CAN-FD frame with 64-bytes of payload + TimeStamp and a 20MHz SPI clock. For this, use the formula (3.14):

$T_{MinRead2.0Frame64BytesWithTimeStampTime}$

$$= \left(\frac{T_{SCK}}{2} + (5+4) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (5+76) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (5+1) \times 8T_{SCK} + T_{SCK}\right)$$

$$= \left(\frac{\frac{1}{20MHz}}{2} + 73 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 649 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 49 \times \frac{1}{20MHz}\right)$$

$$= (25ns + 73 \times 50ns) + (25ns + 649 \times 50ns) + (25ns + 49 \times 50ns) = 3675ns + 32475ns + 2475ns$$

$$= 38625ns = 38.625\mu s$$

This means 25889 frames per second.

The maximum possible frames on a Base CAN-FD bus is 10230 frames which means at full capability, the MCU will be blocked at maximum approximatively 39.5% of its execution time.

The maximum possible frames on an Extended CAN-FD bus is 8565 frames which means at full capability, the MCU will be blocked at maximum approximatively 33.1% of its execution time.

13.3.7. Time to send a full CAN-FD frame

For this, there is some checks to do before sending the frame like the message address in RAM who takes a read of 4-byte SFR. And after, send of the frame who takes 8 bytes plus 64 bytes for the payload to send to RAM. And to finish, update the head of the FIFO who takes 1-byte SFR to write. These actions are mandatory for the driver for each message. If the application does not use the Tx interrupt pin (INT0), it takes 1-byte SFR more.

If there are more than one FIFO, it takes more data to read to get the list where a message room is available.

13.3.7.1. With a Normal driver configuration

With a normal driver communication (eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_NORMAL_USE), the time to send one message and 64-bytes of payload takes:

- Time for 1 frame: $T_{WriteFDFrame} = T_{Read4Bytes} + T_{Write72Bytes} + T_{Write1Bytes}$ (3.15)
- Minimum time: $T_{WriteFDFrameMin} = T_{Read4BytesMin} + T_{Write72BytesMin} + T_{Write1BytesMin}$ (3.16)

The minimum possible time to send a CAN-FD frame with 64-bytes of payload and a 20MHz SPI clock. For this, use the formula (3.16):

$T_{MinWriteFDFrame64BytesTime}$

$$= \left(\frac{T_{SCK}}{2} + (2+4) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (2+72) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (2+1) \times 8T_{SCK} + T_{SCK}\right)$$

$$= \left(\frac{\frac{1}{20MHz}}{2} + 49 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 593 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 25 \times \frac{1}{20MHz}\right)$$

$$= (25ns + 49 \times 50ns) + (25ns + 593 \times 50ns) + (25ns + 25 \times 50ns) = 2475ns + 29625ns + 1275ns$$

$$= 33375ns = 33.375\mu s$$

This means 29962 frames per second.

The maximum possible frames on a Base CAN-FD bus is 10230 frames which means at full capability, the MCU will be blocked at maximum approximatively 34.1% of its execution time.

The maximum possible frames on an Extended CAN-FD bus is 8565 frames which means at full capability, the MCU will be blocked at maximum approximatively 28.6% of its execution time.

13.3.7.2. With a CRC communication driver configuration

With a CRC communication (eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_USE_READ_WRITE_CRC), the time to send one message and 64-bytes of payload takes:

- Time for 1 frame: $T_{CRCWriteFDFrame} = T_{CRCRead4Bytes} + T_{CRCWrite72Bytes} + T_{CRCWrite1Bytes}$ (3.17)
- Minimum time: $T_{CRCWriteFDFrameMin} = T_{CRCRead4BytesMin} + T_{CRCWrite72BytesMin} + T_{CRCWrite1BytesMin}$ (3.18)

The minimum possible time to send a CAN-FD frame with 64-bytes of payload and a 20MHz SPI clock. For this, use the formula (3.18):

$T_{MinWriteFDFrame64BytesTime}$

$$= \left(\frac{T_{SCK}}{2} + (5+4) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (5+72) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (5+1) \times 8T_{SCK} + T_{SCK}\right)$$

$$= \left(\frac{\frac{1}{20MHz}}{2} + 73 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 617 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 49 \times \frac{1}{20MHz}\right)$$

$$= (25ns + 73 \times 50ns) + (25ns + 617 \times 50ns) + (25ns + 49 \times 50ns) = 3675ns + 30875ns + 2475ns$$

$$= 37025ns = 37.025\mu s$$

This means 27008 frames per second.

The maximum possible frames on a Base CAN-FD bus is 10230 frames which means at full capability, the MCU will be blocked at maximum approximatively 37.9% of its execution time.

The maximum possible frames on an Extended CAN-FD bus is 8565 frames which means at full capability, the MCU will be blocked at maximum approximatively 31.7% of its execution time.

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

13.3.7.3. With a Safe Write communication driver configuration

With a CRC communication (eMCP251XFD_DriverConfig::MCP251XFD_DRIVER_USE_SAFE_WRITE), the time to send one message and 64-bytes of payload takes:
- Time for 1 frame: $T_{SafeWriteFDFrame} = T_{CRCRead4Bytes} + T_{SafeWrite72Bytes} + T_{SafeWrite1Bytes}$ (3.19)
- Minimum time: $T_{SafeWriteFDFrameMin} = T_{CRCRead4BytesMin} + T_{SafeWrite72BytesMin} + T_{SafeWrite1BytesMin}$ (3.20)

The minimum possible time to send a CAN-FD frame with 64-bytes of payload and a 20MHz SPI clock. For this, use the formula (3.20):

$$T_{MinWriteFDFrame64BytesTime}$$

$$= \left(\frac{T_{SCK}}{2} + (5+4) \times 8T_{SCK} + T_{SCK}\right) + \frac{72}{4} \times \left(\frac{T_{SCK}}{2} + (4+4) \times 8T_{SCK} + T_{SCK}\right) + \left(\frac{T_{SCK}}{2} + (4+1) \times 8T_{SCK} + T_{SCK}\right)$$

$$= \left(\frac{\frac{1}{20MHz}}{2} + 73 \times \frac{1}{20MHz}\right) + 18 \times \left(\frac{\frac{1}{20MHz}}{2} + 65 \times \frac{1}{20MHz}\right) + \left(\frac{\frac{1}{20MHz}}{2} + 41 \times \frac{1}{20MHz}\right)$$

$$= (25ns + 73 \times 50ns) + 18 \times (25ns + 65 \times 50ns) + (25ns + 41 \times 50ns) = 3675ns + 58950ns + 2075ns$$

$$= 64700ns = 64.7\mu s$$

This means 15455 frames per second.

The maximum possible frames on a Base CAN-FD bus is 10230 frames which means at full capability, the MCU will be blocked at maximum approximatively 66.2% of its execution time.

The maximum possible frames on an Extended CAN-FD bus is 8565 frames which means at full capability, the MCU will be blocked at maximum approximatively 55.4% of its execution time.

This means that if you want to use more than one device which sends continuously FD frames on separates buses, you need an Asynchronous system with DMA and as many SPI buses as device.

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

14. CONFIGURATION STRUCTURES

14.1. Device object structure

The MCP251XFD device object structure contains all information that is mandatory to work with a device. It is always the first parameter of each functions of the driver.

Source code:

```
typedef struct MCP251XFD MCP251XFD;
typedef uint8_t DriverInternal;

struct MCP251XFD
{
  void *UserDriverData;

  //--- Driver configuration ---
  setMCP251XFD_DriverConfig DriverConfig;
  TMCP251XFDDriverInternal InternalConfig;

  //--- IO configuration ---
  uint8_t GPIOsOutState;

  //--- Interface driver call functions ---
  uint8_t SPI_ChipSelect;
  void *InterfaceDevice;
  uint32_t SPIClockSpeed;
  MCP251XFD_SPIInit_Func      fnSPI_Init;
  MCP251XFD_SPITransfer_Func fnSPI_Transfer;

  //--- Time call function ---
  GetCurrentms_Func fnGetCurrentms;

  //--- CRC16-CMS call function ---
  ComputeCRC16_Func fnComputeCRC16;
};
```

14.1.1. Data fields

void *UserDriverData**

Optional, can be used to store driver data related to the project and the device or NULL. This field is not used or modified by the driver.

setMCP251XFD_DriverConfig **DriverConfig**

This is the driver's configuration. Configuration can be OR'ed. See type enum for the possible configurations.

Type

    enum   *setMCP251XFD_DriverConfig*

Initial value / default

    MCP251XFD_DRIVER_NORMAL_USE

TMCP251XFDDriverInternal **InternalConfig**

This is the internal driver configuration. The user should not change the value, only the driver can change values. The value is changed following driver usage.

Type

    typedef   *uint8_t* TMCP251XFDDriverInternal

Initial value / default

    Regardless of the value set when filling the struct, the value will be modify at device initialization when using the function *Init_MCP251XFD()*

uint8_t GPIOsOutState

This is the current GPIO output state. By checking this value, you can know the latest status set.

Initial value / default

The value indicate when filling the structure set the output state of the pin GPIO0 and GPIO1 when put in output at initialization when using *Init_MCP251XFD()*.

uint8_t SPI_ChipSelect

This is the Chip Select index that will be set at the call of a transfer.

void *InterfaceDevice

This is the pointer that will be in the first parameter of all interface call functions.

MCP251XFD_SPIInit_Func fnSPI_Init

This function will be called at driver initialization to configure the SPI interface driver.

Type

typedef   *eERRORRESULT (*MCP251XFD_SPIInit_Func)(void *, uint8_t, const uint32_t)*

Initial value / default

This function must point to a function else a ERR__PARAMETER_ERROR is returned by the function *Init_MCP251XFD()*.

MCP251XFD_SPITransfer_Func fnSPI_Transfer

This function will be called at driver read/write data from/to the interface driver. It cannot point to NULL.

Type

typedef   *eERRORRESULT (*MCP251XFD_SPITransfer_Func)(void *, uint8_t, uint8_t *, uint8_t *, size_t)*

Initial value / default

This function must point to a function else an ERR__PARAMETER_ERROR is returned when using a function that require to communicate with the device.

GetCurrentms_Func fnGetCurrentms

This function will be called when the driver needs to get current millisecond. Some functions need a timeout, without, they can be stuck forever.

Type

typedef   *uint32_t (*GetCurrentms_Func)(void)*

Initial value / default

This function has to point to a function else an ERR__PARAMETER_ERROR is returned by the functions *Init_MCP251XFD()*, *MCP251XFD_WaitOperationModeChange()*, *MCP251XFD_ResetFIFO()* or *MCP251XFD_ResetDevice()*.

ComputeCRC16_Func fnComputeCRC16

This function will be called when a CRC16-CMS computation is needed (MCP251XFD_DRIVER_USE_READ_WRITE_CRC or MCP251XFD_DRIVER_USE_SAFE_WRITE flags set in the field MCP251XFD::DriverConfig). In normal mode, this can point to NULL.

Type

typedef   *uint16_t (*ComputeCRC16_Func)(const uint8_t*, size_t)*

Initial value / default

This function has to point to a function in case of using else an ERR__PARAMETER_ERROR is returned by the functions *Init_MCP251XFD()*, *MCP251XFD_WaitOperationModeChange()*, *MCP251XFD_ResetFIFO()* or *MCP251XFD_ResetDevice()*.

uint32_t SPIClockSpeed

This is the SPI nominal clock speed in Hertz. This value cannot be higher than device SYSCLK divide by 2.

14.1.2. Enumerators

enum eMCP251XFD_DriverConfig
typedef eMCP251XFD_DriverConfig setMCP251XFD_DriverConfig

Driver configurations, parameters can be OR'ed. MCP251XFD_DRIVER_USE_READ_WRITE_CRC will always replace MCP251XFD_DRIVER_NORMAL_USE when communicating with the device. MCP251XFD_DRIVER_USE_SAFE_WRITE will always replace MCP251XFD_DRIVER_NORMAL_USE and MCP251XFD_DRIVER_USE_READ_WRITE_CRC when communicating with the device in case of a write.

Enumerator

MCP251XFD_DRIVER_NORMAL_USE	0x00	Use the driver with no special verifications, just settings verifications (usually the fastest mode)
MCP251XFD_DRIVER_SAFE_RESET	0x01	Set Configuration mode first and next send a Reset command with a SPI clock at 1MHz max (MCP251XFD_OSCFREQ_MIN div by 2)
MCP251XFD_DRIVER_ENABLE_ECC	0x02	Enable the ECC just before the RAM initialization and activate ECCCON_SECIE and ECCCON_DEDIE interrupt flags
MCP251XFD_DRIVER_INIT_CHECK_RAM	0x04	Check RAM at initialization by writing some data and checking them on all the RAM range (slower at initialization, take a long time)
MCP251XFD_DRIVER_INIT_SET_RAM_AT_0	0x08	Set all bytes of the RAM to 0x00 (slower at initialization)
MCP251XFD_DRIVER_CLEAR_BUFFER_BEFORE_READ	0x10	This send 0x00 byte while reading SPI interface, mainly for cybersecurity purpose (little bit slower)
MCP251XFD_DRIVER_USE_READ_WRITE_CRC	0x20	Use CRC with all commands and data going to and from the controller (add 3 more bytes to each transaction, 2 for CRC + 1 for length)
MCP251XFD_DRIVER_USE_SAFE_WRITE	0x40	Each SFR write or memory write is sent one at a time (slower but send only the 2 bytes for CRC)

14.1.3. TMCP251XFDDriverInternal type and InternalConfig variable

Warning: This variable should never be changed by the application.

This variable is used by the driver to estimate the state of the device and some information that is impossible to retrieve directly from device. This save some unnecessary transfer communications.

For information, the TMCP251XFDDriverInternal type is defined from a uint8_t and is constituted as this Cl..LDSS where:

C	'0' → CAN2.0	
	'1' → CAN-FD	(set by using the define MCP251XFD_CANFD_ENABLED)
l	'0' → Do not use RRS as SID11	
	'1' → Use RRS as SID11	(set by using the enum eMCP251XFD_CANCtrlFlags::CANFD_USE_RRS_BIT_AS_SID11)
.	Not used	
L	'0' → LowPower mode disable	
	'1' → LowPower mode enable	(set by using the define MCP251XFD_SFR_OSC8_LPMEN)
D	'0' → MCP2517FD	(enum eMCP251XFD_Devices::MCP2517FD)
	'1' → MCP2518FD	(enum eMCP251XFD_Devices::MCP2518FD)
S	'00' → Sleep not configured	(enum eMCP251XFD_PowerStates::MCP251XFD_DEVICE_SLEEP_NOT_CONFIGURED)
	'01' → Normal power	(enum eMCP251XFD_PowerStates::MCP251XFD_DEVICE_NORMAL_POWER_STATE)
	'10' → Sleep state	(enum eMCP251XFD_PowerStates::MCP251XFD_DEVICE_SLEEP_STATE)
	'11' → LowPower Sleep state	(enum eMCP251XFD_PowerStates::MCP251XFD_DEVICE_LOWPOWER_SLEEP_STATE)

14.1.4. Driver interface handle functions

```
eERRORRESULT (*MCP251XFD_SPIInit_Func)(
      void *pIntDev,
      uint8_t chipSelect
      const uint32_t sckFreq)
```

Function for interface driver initialization of the MCP251XFD. This function will be called at driver initialization to configure the interface driver.

Parameters

Input	*pIntDev	Is the MCP251XFD.InterfaceDevice of the device that call the interface initialization
Input	chipSelect	Is the Chip Select index to use for the SPI initialization
Input	sckFreq	Is the SCK frequency in Hz to set at the interface initialization

Return

Returns an *eERRORRESULT* value enumerator dependent of how the return error is implemented by the user. It is recommended, during the implement of the pointer interface function, to return only ERR__SPI_PARAMETER_ERROR, ERR__SPI_COMM_ERROR, ERR__SPI_CONFIG_ERROR, or ERR__SPI_TIMEOUT when there is an error and ERR_OK when all went fine.

```
eERRORRESULT (*MCP251XFD_SPITransfer_Func)(
      void *pIntDev,
      uint8_t chipSelect,
      uint8_t *txData,
      uint8_t *rxData,
      size_t size)
```

Function for interface transfer of the MCP251XFD. This function will be called at driver read/write data from/to the interface driver.

Parameters

Input	*pIntDev	Is the MCP251XFD.InterfaceDevice of the device that call the data transfer
Input	chipSelect	Is the Chip Select index to use for the SPI transfer
Input	*txData	Is the data to send through the interface
Output	*rxData	Is where the data received through the interface will be stored. This parameter can be nulled by the driver if no received data is expected
Input	size	Is the size of the data to send and receive through the interface

Return

Returns an *eERRORRESULT* value enumerator dependent of how the return error is implemented by the user. It is recommended, during the implement of the pointer interface function, to return only ERR__SPI_PARAMETER_ERROR, ERR__SPI_COMM_ERROR, ERR__SPI_CONFIG_ERROR, or ERR__SPI_TIMEOUT when there is an error and ERR_OK when all went fine.

```
uint32_t (*GetCurrentms_Func)(void)
```

Function that give the current millisecond of the system to the driver. This function will be called when the driver needs to get current millisecond.

Return

Returns the current millisecond of the system

```
uint16_t (*ComputeCRC16_Func)(
      const uint8_t* data,
      size_t size)
```

Function that compute CRC16-CMS for the driver. This function will be called when a CRC16-CMS computation is needed (*MCP251XFD_DRIVER_USE_READ_WRITE_CRC* or *MCP251XFD_DRIVER_USE_SAFE_WRITE* flags set in the field MCP251XFD::DriverConfig). In normal mode, this can point to NULL.

Parameters

Input	*data	Is the byte steam of data to compute
Input	size	Is the size of the byte stream

Return

Returns the result of the CRC16-CMS computation

The MCP251XFD configuration structure contains all information to configure most of the controller and CAN controller of the device at initialization. This structure is used once, when using *Init_MCP251XFD()* function.

Source code:

```
typedef struct MCP251XFD_Config
{
  //--- Controller clocks ---
  uint32_t XtalFreq;
  uint32_t OscFreq;
  eMCP251XFD_CLKINtoSYSCLK SysclkConfig;
  eMCP251XFD_CLKODIV ClkoPinConfig;
  uint32_t *SYSCLK_Result;

  //--- CAN configuration ---
  uint32_t NominalBitrate;
  uint32_t DataBitrate;
  MCP251XFD_BitTimeStats *BitTimeStats;
  eMCP251XFD_Bandwidth Bandwidth;
  setMCP251XFD_CANCtrlFlags ControlFlags;

  //--- GPIOs and Interrupts pins ---
  eMCP251XFD_GPIO0Mode GPIO0PinMode;
  eMCP251XFD_GPIO1Mode GPIO1PinMode;
  eMCP251XFD_OutMode INTsOutMode;
  eMCP251XFD_OutMode TXCANOutMode;

  //--- Interrupts ---
  setMCP251XFD_InterruptEvents SysInterruptFlags;
} MCP251XFD_Config;
```

14.2.1. Data fields

uint32_t *XtalFreq*

Component CLKIN Xtal/Resonator frequency (min 4MHz, max 40MHz). Set it to 0 if oscillator is used.

uint32_t *OscFreq*

Component CLKIN oscillator frequency (min 2MHz, max 40MHz). Set it to 0 if Xtal/Resonator is used.

eMCP251XFD_CLKINtoSYSCLK *SysclkConfig*

Factor of frequency for the SYSCLK. SYSCLK = CLKIN x SysclkConfig where CLKIN is XtalFreq or OscFreq.

Type

    enum   *eMCP251XFD_SCLKDIV*

eMCP251XFD_CLKODIV *ClkoPinConfig*

Configure the CLKO pin (SCLK div by 1, 2, 4, 10 or Start Of Frame).

Type

    enum   *eMCP251XFD_CLKODIV*

Initial value / default

    MCP251XFD_CLKO_DivBy10

uint32_t **SYSCLK_Result*

This is the SYSCLK of the component after configuration (can be NULL if the internal SYSCLK of the component do not have to be known). If not NULL, this pointed variable is filled by the *Init_MCP251XFD()* following XtalFreq or OscFreq and Use10xPLL.

uint32_t *NominalBitrate*

Speed of the Frame description and arbitration.

uint32_t *DataBitrate*

Speed of all the data bytes of the frame (if CAN2.0 only mode, set to value MCP251XFD_NO_CANFD).

MCP251XFD_BitTimeStats ****BitTimeStats***

Point to a Bit Time stat structure (set to NULL if no statistics are necessary). If not NULL, this pointed variable is filled with the result of the MCP251XFD_Config::NormalBitrate and MCP251XFD_Config::DataBitrate computation and result calculus.

Type

struct   *MCP251XFD_BitTimeStats*

eMCP251XFD_Bandwidth ***Bandwidth***

Transmit Bandwidth Sharing, this is the delay between two consecutive transmissions (in arbitration bit times) for the CiCON.TXBWS register.

Type

enum   *eMCP251XFD_Bandwidth*

Initial value / default

MCP251XFD_NO_DELAY

setMCP251XFD_CANCtrlFlags ***ControlFlags***

Set of CAN control flags to configure the CAN controller. Configuration can be OR'ed.

Type

Set of enum   *setMCP251XFD_CANCtrlFlags*

Initial value / default

MCP251XFD_CAN_RESTRICTED_MODE_ON_ERROR | *MCP251XFD_CAN_ESI_REFLECTS_ERROR_STATUS* | *MCP251XFD_CAN_UNLIMITED_RETRANS_ATTEMPTS* | *MCP251XFD_CANFD_BITRATE_SWITCHING_ENABLE* | *MCP251XFD_CAN_PROTOCOL_EXEPT_ENTER_INTEGRA* | *MCP251XFD_CANFD_USE_NONISO_CRC* | *MCP251XFD_CANFD_DONT_USE_RRS_BIT_AS_SID11*

eMCP251XFD_GPIO0Mode ***GPIO0PinMode***

Startup INT0/GPIO0/XSTBY pins mode (INT0 => Interrupt for TX).

Type

enum   *eMCP251XFD_GPIO0Mode*

eMCP251XFD_GPIO1Mode ***GPIO1PinMode***

Startup INT1/GPIO1 pins mode (INT1 => Interrupt for RX).

Type

enum   *eMCP251XFD_GPIO1Mode*

eMCP251XFD_OutMode ***INTsOutMode***

Define the output type of all interrupt pins (INT, INT0 and INT1).

Type

enum   *eMCP251XFD_OutMode*

eMCP251XFD_OutMode ***TXCANOutMode***

Define the output type of the TXCAN pin.

Type

enum   *eMCP251XFD_OutMode*

setMCP251XFD_InterruptEvents ***SysInterruptFlags***

Set of system interrupt flags to enable. Configuration can be OR'ed.

Type

enum   *setMCP251XFD_InterruptEvents*

Ref: MCP251XFD driver library guide (Synchronous driver).docx

14.2.2. Enumerators

enum **eMCP251XFD_CLKINtoSYSCLK**

Xtal/Oscillator (CLKIN) multiplier/divisor to SYSCLK.

Enumerator

MCP251XFD_SYSCLK_IS_CLKIN	SYSCLK is CLKIN (no PLL, SCLK divide by 1).
	For CLKIN at 20MHz or 40MHz
MCP251XFD_SYSCLK_IS_CLKIN_DIV_2	SYSCLK is CLKIN divide by 2 (no PLL, SCLK divide by 2).
	For CLKIN at 20MHz or 40MHz
MCP251XFD_SYSCLK_IS_CLKIN_MUL_5	SYSCLK is CLKIN multiply by 5 (PLL enable, SCLK divide by 2).
	For CLKIN at 2MHz or 4MHz
MCP251XFD_SYSCLK_IS_CLKIN_MUL_10	SYSCLK is CLKIN multiply by 10 (PLL enable, SCLK divide by 1).
	For CLKIN at 2MHz or 4MHz

enum **eMCP251XFD_CLKODIV**

Clock Output Divisor for the OSC.CLKODIV register and IOCON.SOF configuration.

Enumerator

MCP251XFD_CLKO_DivBy1	**0b000**	Clock Output Divisor by 1
MCP251XFD_CLKO_DivBy2	**0b001**	Clock Output Divisor by 2
MCP251XFD_CLKO_DivBy4	**0b010**	Clock Output Divisor by 4
MCP251XFD_CLKO_DivBy10	**0b011**	Clock Output Divisor by 10 (default)
MCP251XFD_CLKO_SOF	**0b111**	CLKO pin output Start Of Frame (Not configured in the OSC.CLKODIV register)

enum **eMCP251XFD_Bandwidth**

CAN Controller Transmit Bandwidth Sharing bits for the CiCON.TXBWS register. Delay between two consecutive transmissions (in arbitration bit times).

Enumerator

MCP251XFD_NO_DELAY	**0b0000**	No delay (default)
MCP251XFD_DELAY_2BIT_TIMES	**0b0001**	Delay 2 arbitration bit times
MCP251XFD_DELAY_4BIT_TIMES	**0b0010**	Delay 4 arbitration bit times
MCP251XFD_DELAY_8BIT_TIMES	**0b0011**	Delay 8 arbitration bit times
MCP251XFD_DELAY_16BIT_TIMES	**0b0100**	Delay 16 arbitration bit times
MCP251XFD_DELAY_32BIT_TIMES	**0b0101**	Delay 32 arbitration bit times
MCP251XFD_DELAY_64BIT_TIMES	**0b0110**	Delay 64 arbitration bit times
MCP251XFD_DELAY_128BIT_TIMES	**0b0111**	Delay 128 arbitration bit times
MCP251XFD_DELAY_256BIT_TIMES	**0b1000**	Delay 256 arbitration bit times
MCP251XFD_DELAY_512BIT_TIMES	**0b1001**	Delay 512 arbitration bit times
MCP251XFD_DELAY_1024BIT_TIMES	**0b1010**	Delay 1024 arbitration bit times
MCP251XFD_DELAY_2048BIT_TIMES	**0b1011**	Delay 2048 arbitration bit times
MCP251XFD_DELAY_4096BIT_TIMES	**0b1100**	Delay 4096 arbitration bit times

enum **eMCP251XFD_GPIO0Mode**

INT0/GPIO0/XSTBY configuration for the IOCON register.

Enumerator

MCP251XFD_PIN_AS_INT0_TX	**0b00**	INT0/GPIO0/XSTBY pin as TX Interrupt output (active low)
MCP251XFD_PIN_AS_GPIO0_IN	**0b01**	INT0/GPIO0/XSTBY pin as GPIO input
MCP251XFD_PIN_AS_GPIO0_OUT	**0b10**	INT0/GPIO0/XSTBY pin as GPIO output
MCP251XFD_PIN_AS_XSTBY	**0b11**	INT0/GPIO0/XSTBY pin as Transceiver Standby output

enum **eMCP251XFD_GPIO1Mode**

INT1/GPIO1 configuration for the IOCON register.

Enumerator

MCP251XFD_PIN_AS_INT1_RX	**0b00**	INT1/GPIO1 pin as RX Interrupt output (active low)
MCP251XFD_PIN_AS_GPIO1_IN	**0b01**	INT1/GPIO1 pin as GPIO input
MCP251XFD_PIN_AS_GPIO1_OUT	**0b10**	INT1/GPIO1 pin as GPIO output

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

enum eMCP251XFD_OutMode

Output configuration for the IOCON.INTOD and the IOCON.TXCANOD register.

Enumerator

MCP251XFD_PINS_PUSHPULL_OUT	0b0	Pin with Push/Pull output
MCP251XFD_PINS_OPENDRAIN_OUT	0b1	Pin with Open Drain output

enum eMCP251XFD_CANCtrlFlags
typedef eMCP251XFD_CANCtrlFlags **setMCP251XFD_CANCtrlFlags**

CAN control configuration flags. Configuration can be OR'ed.

Enumerator

MCP251XFD_CAN_RESTRICTED_MODE_ON_ERROR	0x00	Transition to Restricted Operation Mode on system error
MCP251XFD_CAN_LISTEN_ONLY_MODE_ON_ERROR	0x01	Transition to Listen Only Mode on system error
MCP251XFD_CAN_ESI_REFLECTS_ERROR_STATUS	0x00	ESI reflects error status of CAN controller
MCP251XFD_CAN_GATEWAY_MODE_ESI_RECESSIVE	0x02	Transmit ESI in Gateway Mode, ESI is transmitted recessive when ESI of message is high or CAN controller error passive
MCP251XFD_CAN_UNLIMITED_RETRANS_ATTEMPTS	0x00	Unlimited number of retransmission attempts, MCP251XFD_FIFO.Attempts (CiFIFOCONm.TXAT) will be ignored
MCP251XFD_CAN_RESTRICTED_RETRANS_ATTEMPTS	0x04	Restricted retransmission attempts, MCP251XFD_FIFO.Attempts (CiFIFOCONm.TXAT) is used
MCP251XFD_CANFD_BITRATE_SWITCHING_ENABLE	0x00	Bit Rate Switching is Enabled, Bit Rate Switching depends on BRS in the Transmit Message Object
MCP251XFD_CANFD_BITRATE_SWITCHING_DISABLE	0x08	Bit Rate Switching is Disabled, regardless of BRS in the Transmit Message Object
MCP251XFD_CAN_PROTOCOL_EXEPT_ENTER_INTEGRA	0x00	If a Protocol Exception is detected, the CAN FD Controller Module will enter Bus Integrating state. A recessive "res bit" following a recessive FDF bit is called a Protocol Exception
MCP251XFD_CAN_PROTOCOL_EXEPT_AS_FORM_ERROR	0x10	Protocol Exception is treated as a Form Error. A recessive "res bit" following a recessive FDF bit is called a Protocol Exception
MCP251XFD_CANFD_USE_NONISO_CRC	0x00	Do NOT include Stuff Bit Count in CRC Field and use CRC Initialization Vector with all zeros
MCP251XFD_CANFD_USE_ISO_CRC	0x20	Include Stuff Bit Count in CRC Field and use Non-Zero CRC Initialization Vector according to ISO 11898-1:2015
MCP251XFD_CANFD_DONT_USE_RRS_BIT_AS_SID11	0x00	Do not use RRS; SID<10:0> according to ISO 11898-1:2015
MCP251XFD_CANFD_USE_RRS_BIT_AS_SID11	0x40	RRS is used as SID11 in CAN FD base format messages: SID<11:0> = {SID<10:0>, SID11}

enum eMCP251XFD_InterruptEvents
typedef eMCP251XFD_InterruptEvents **setMCP251XFD_InterruptEvents**

Interrupt Events, can be OR'ed.

Enumerator

MCP251XFD_INT_NO_EVENT	0x0000	No interrupt events
MCP251XFD_INT_TX_EVENT	0x0001	Transmit events. Equivalent to INT0
MCP251XFD_INT_RX_EVENT	0x0002	Receive events. Equivalent to INT1
MCP251XFD_INT_TEF_EVENT	0x0010	TEF events. Clearable in specific FIFO
MCP251XFD_INT_TX_ATTEMPTS_EVENT	0x0400	Transmit attempts events. Clearable in specific FIFO
MCP251XFD_INT_RX_OVERFLOW_EVENT	0x0800	Receive overflow events. Clearable in specific FIFO
MCP251XFD_INT_TIME_BASE_COUNTER_EVENT	0x0004	Time base counter events. Clearable in CiINT
MCP251XFD_INT_OPERATION_MODE_CHANGE_EVENT	0x0008	Operation mode change events. Clearable in CiINT

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

MCP251XFD_INT_RAM_ECC_EVENT	0x0100	ECC RAM events. Clearable in ECCSTA
MCP251XFD_INT_SPI_CRC_EVENT	0x0200	SPI CRC events. Clearable in CRC
MCP251XFD_INT_SYSTEM_ERROR_EVENT	0x1000	System error events. Clearable in CiINT
MCP251XFD_INT_BUS_ERROR_EVENT	0x2000	Bus error events. Clearable in CiINT
MCP251XFD_INT_BUS_WAKEUP_EVENT	0x4000	Bus wakeup events, only when sleeping. Clearable in CiINT
MCP251XFD_INT_RX_INVALID_MESSAGE_EVENT	0x8000	Invalid receipt message events. Clearable in CiINT
MCP251XFD_INT_ENABLE_ALL_EVENTS	0xFF1F	Enable all events

14.3. FIFO configuration structure

The MCP251XFD FIFO configuration structure contains all information to configure each FIFO of the device at initialization. This structure is used when using *MCP251XFD_ConfigureFIFO()* and in case of configuration of one FIFO, an array of the struct can be used when using *MCP251XFD_ConfigureFIFOList()* function.

Source code:

```
typedef struct MCP251XFD_FIFO
{
  eMCP251XFD_FIFO Name;

  //--- FIFO Size ---
  eMCP251XFD_MessageDeep Size;
  eMCP251XFD_PayloadSize Payload;

  //--- Configuration ---
  eMCP251XFD_SelTXRX Direction;
  eMCP251XFD_Attempts Attempts;
  eMCP251XFD_Priority Priority;
  eMCP251XFD_FIFOCtrlFlags ControlFlags;
  eMCP251XFD_FIFOIntFlags InterruptFlags;

  //--- FIFO RAM Infos ---
  MCP251XFD_RAMInfos *RAMInfos;
} MCP251XFD_FIFO;
```

14.3.1. Data fields

eMCP251XFD_FIFO *Name*

FIFO name (MCP251XFD_TXQ, MCP251XFD_FIFO1..31 or MCP251XFD_TEF).

Type

enum   *eMCP251XFD_FIFO*

eMCP251XFD_MessageDeep *Size*

FIFO Message size deep (1 to 32).

Type

enum   *eMCP251XFD_MessageDeep*

Initial value / default

MCP251XFD_FIFO_1_MESSAGE_DEEP

eMCP251XFD_PayloadSize *Payload*

Message Payload Size (8, 12, 16, 20, 24, 32, 48 or 64 bytes).

Type

enum   *eMCP251XFD_PayloadSize*

Initial value / default

MCP251XFD_PAYLOAD_8BYTE

eMCP251XFD_SelTXRX *Direction*

TX/RX FIFO Selection.

Type

enum   *eMCP251XFD_SelTXRX*

Initial value / default

MCP251XFD_RECEIVE_FIFO

eMCP251XFD_Attempts *Attempts*

Retransmission Attempts. This feature is enabled when CiCON.RTXAT is set.

Type

enum   *eMCP251XFD_Attempts*

Initial value / default

MCP251XFD_UNLIMITED_ATTEMPTS

eMCP251XFD_Priority *Priority*

Message Transmit Priority ('0x00' = Lowest Message Priority and '0x1F' = Highest Message Priority).

Type

enum   *eMCP251XFD_Priority*

Initial value / default

MCP251XFD_MESSAGE_TX_PRIORITY1

eMCP251XFD_FIFOCtrlFlags *ControlFlags*

FIFO control flags to configure the FIFO.

Type

enum   *eMCP251XFD_FIFOCtrlFlags*

Initial value / default

MCP251XFD_FIFO_NO_CONTROL_FLAGS

eMCP251XFD_FIFOIntFlags *InterruptFlags*

FIFO interrupt flags to configure interrupts of the FIFO.

Type

enum   *eMCP251XFD_FIFOIntFlags*

Initial value / default

MCP251XFD_FIFO_NO_INTERRUPT_FLAGS

MCP251XFD_RAMInfos **RAMInfos*

Point to a RAM Information's structure of the FIFO (set to NULL if no RAM use is necessary). If not NULL, this pointed variable is filled with the result of the RAM used and addresses of the FIFO.

Type

struct   *MCP251XFD_RAMInfos*

14.3.2. Enumerators

enum **eMCP251XFD_FIFO**

Available FIFO list.

Enumerator

MCP251XFD_TEF	-1	TEF - Transmit Event FIFO
MCP251XFD_TXQ	0	TXQ - Transmit Queue
MCP251XFD_FIFO1	1	FIFO 1
…	…	…
MCP251XFD_FIFO31	31	FIFO 31
MCP251XFD_NO_FIFO	32	No FIFO code for FIFO status functions

enum **eMCP251XFD_MessageDeep**

FIFO Size for the CiTEFCON.FSIZE, CiTXQCON.FSIZE and CiFIFOCONm.FSIZE.

Enumerator

MCP251XFD_FIFO_1_MESSAGE_DEEP	-1	FIFO is 1 Message deep
…	…	…
MCP251XFD_FIFO_32_MESSAGE_DEEP	31	FIFO is 32 Message deep

enum eMCP251XFD_Attempts

Retransmission Attempts for the CiTXQCON.TXAT and CiFIFOCONm.TXAT.

Enumerator

MCP251XFD_DISABLE_ATTEMPS	**0b00**	Disable retransmission attempts
MCP251XFD_THREE_ATTEMPTS	**0b01**	Three retransmission attempts
MCP251XFD_UNLIMITED_ATTEMPTS	**0b10**	Unlimited number of retransmission attempts

enum eMCP251XFD_PayloadSize

Payload Size for the CiTXQCON.PLSIZE and CiFIFOCONm.PLSIZE.

Enumerator

MCP251XFD_PAYLOAD_8BYTE	**0b000**	Payload 8 data bytes
MCP251XFD_PAYLOAD_12BYTE	**0b001**	Payload 12 data bytes
MCP251XFD_PAYLOAD_16BYTE	**0b010**	Payload 16 data bytes
MCP251XFD_PAYLOAD_20BYTE	**0b011**	Payload 20 data bytes
MCP251XFD_PAYLOAD_24BYTE	**0b100**	Payload 24 data bytes
MCP251XFD_PAYLOAD_32BYTE	**0b101**	Payload 32 data bytes
MCP251XFD_PAYLOAD_48BYTE	**0b110**	Payload 48 data bytes
MCP251XFD_PAYLOAD_64BYTE	**0b111**	Payload 64 data bytes

enum eMCP251XFD_SelTXRX

FIFO Direction for the CiFIFOCONm.TXEN.

Enumerator

MCP251XFD_RECEIVE_FIFO	**0b0**	Receive FIFO
MCP251XFD_TRANSMIT_FIFO	**0b1**	Transmit FIFO

enum eMCP251XFD_Priority

Message Transmit Priority for the CiTXQCON.TXPRI.

Enumerator

MCP251XFD_MESSAGE_TX_PRIORITY1	**0x00**	Message transmit priority 1 (Lowest)
…	**…**	…
MCP251XFD_MESSAGE_TX_PRIORITY32	**0x1F**	Message transmit priority 32 (Highest)

enum eMCP251XFD_FIFOCtrlFlags

MCP251XFD FIFO configuration flags.

Enumerator

MCP251XFD_FIFO_NO_CONTROL_FLAGS	**0x00**	Set no control flags
MCP251XFD_FIFO_NO_RTR_RESPONSE	**0x00**	When a remote transmit is received, Transmit Request (TXREQ) of the FIFO will be unaffected
MCP251XFD_FIFO_AUTO_RTR_RESPONSE	**0x40**	When a remote transmit is received, Transmit Request (TXREQ) of the FIFO will be set
MCP251XFD_FIFO_NO_TIMESTAMP_ON_RX	**0x00**	Do not capture time stamp
MCP251XFD_FIFO_ADD_TIMESTAMP_ON_RX	**0x20**	Capture time stamp in received message object in RAM
MCP251XFD_FIFO_ADD_TIMESTAMP_ON_OBJ	**0x20**	Capture time stamp in objects in TEF

enum eMCP251XFD_FIFOIntFlags

MCP251XFD FIFO interruption flags.

Enumerator

MCP251XFD_FIFO_NO_INTERRUPT_FLAGS	**0x00**	Set no interrupt flags
MCP251XFD_FIFO_TX_ATTEMPTS_EXHAUSTED_INT	**0x10**	Transmit Attempts Exhausted Interrupt Enable
MCP251XFD_FIFO_OVERFLOW_INT	**0x08**	Overflow Interrupt Enable (Not available on TXQ (FIFO0))
MCP251XFD_FIFO_TRANSMIT_FIFO_EMPTY_INT	**0x04**	Transmit FIFO Empty Interrupt Enable
MCP251XFD_FIFO_TRANSMIT_FIFO_HALF_EMPTY_INT	**0x02**	Transmit FIFO Half Empty Interrupt Enable (Not available on TXQ (FIFO0))
MCP251XFD_FIFO_TRANSMIT_FIFO_NOT_FULL_INT	**0x01**	Transmit FIFO Not Full Interrupt Enable
MCP251XFD_FIFO_RECEIVE_FIFO_FULL_INT	**0x04**	Receive FIFO Full Interrupt Enable
MCP251XFD_FIFO_RECEIVE_FIFO_HALF_FULL_INT	**0x02**	Receive FIFO Half Full Interrupt Enable
MCP251XFD_FIFO_RECEIVE_FIFO_NOT_EMPTY_INT	**0x01**	Receive FIFO Not Empty Interrupt Enable
MCP251XFD_FIFO_EVENT_FIFO_FULL_INT	**0x04**	Transmit Event FIFO Full Interrupt Enable
MCP251XFD_FIFO_EVENT_FIFO_HALF_FULL_INT	**0x02**	Transmit Event FIFO Half Full Interrupt Enable
MCP251XFD_FIFO_EVENT_FIFO_NOT_EMPTY_INT	**0x01**	Transmit Event FIFO Not Empty Interrupt Enable

14.4. Filter configuration structure

The MCP251XFD filter configuration structure contains all information to configure each filter of the device at initialization. This structure is used when using MCP251XFD_ConfigureFilter() and in case of configuration of one filter, an array of the struct can be used when using MCP251XFD_ConfigureFilterList() function.

Source code:

```c
typedef struct MCP251XFD_Filter
{
  //--- Configuration ---
  eMCP251XFD_Filter Filter;
  bool EnableFilter;
  eMCP251XFD_FilterMatch Match;
  eMCP251XFD_FIFO PointTo;

  //--- Message Filter ---
  uint32_t AcceptanceID;
  uint32_t AcceptanceMask;
  bool ExtendedID;
} MCP251XFD_Filter;
```

14.4.1. Data fields

eMCP251XFD_Filter *Filter*

Filter to configure.

Type

enum   eMCP251XFD_Filter

bool *EnableFilter*

If 'true' it enables the filter else it disables the filter.

eMCP251XFD_FilterMatch *Match*

Filter match type of the frame (SID and/or EID).

Type

enum   eMCP251XFD_FilterMatch

eMCP251XFD_FIFO *PointTo*

Message matching filter is stored in pointed FIFO name (FIFO1 to 31).

Type

enum   eMCP251XFD_FIFO

uint32_t *AcceptanceID*

Message Filter Acceptance SID+(SID11 in FD mode)+EID.

uint32_t *AcceptanceMask*

Message Filter Mask SID+(SID11 in FD mode)+EID. Corresponding bits to AcceptanceID: '1': bit to filter ; '0' bit that do not care.

bool *ExtendedID*

If 'true' it uses the extended ID else, it does not use the extended ID.

14.4.2. Enumerators

enum eMCP251XFD_Filter

Available Filter list.

Enumerator

MCP251XFD_FILTER0	**0**	Filter 0
…	**…**	…
MCP251XFD_FILTER31	**31**	Filter 31

enum eMCP251XFD_FilterMatch

Filter match type.

Enumerator

MCP251XFD_MATCH_ONLY_SID	**0**	Match only messages with standard identifier (+SID11 in FD mode if configured)
MCP251XFD_MATCH_ONLY_EID	**1**	Match only messages with extended identifier
MCP251XFD_MATCH_SID_EID	**2**	Match both standard and extended message frames

14.4.3. Defines

#define MCP251XFD_ACCEPT_ALL_MESSAGES

Indicate that the filter will accept all messages.

Value

0x00000000u

14.5. Bit Time Statistics structure

The MCP251XFD bit time statistics structure contains all information about the calculated bit time configuration regarding Nominal Bitrate and Data Bitrate of the device at initialization. This structure is filled by the function *MCP251XFD_CalculateBitrateStatistics()* when using *MCP251XFD_CalculateBitTimeConfiguration()* function. The calculous of bit time configuration is done when using *Init_MCP251XFD()* function.

Source code:

```
typedef struct MCP251XFD_BitTimeStats
{
  uint32_t NominalBitrate;
  uint32_t DataBitrate;
  uint32_t MaxBusLength;
  uint32_t NSamplePoint;
  uint32_t DSamplePoint;
  uint32_t OscTolC1;
  uint32_t OscTolC2;
  uint32_t OscTolC3;
  uint32_t OscTolC4;
  uint32_t OscTolC5;
  uint32_t OscTolerance;
} MCP251XFD_BitTimeStats;
```

14.5.1. Data fields

uint32_t *NominalBitrate*

This is the actual nominal bitrate with the current configuration after calculous of bit time configuration with *MCP251XFD_CalculateBitTimeConfiguration()* function.

uint32_t *DataBitrate*

This is the actual data bitrate with the current configuration after calculous of bit time configuration with *MCP251XFD_CalculateBitTimeConfiguration()* function.

uint32_t *MaxBusLength*

This is the maximum bus length according to parameters of the bit time configuration. This is calculated in the *MCP251XFD_CalculateBitrateStatistics()* function.

uint32_t *NSamplePoint*

Nominal Sample Point of the bit time configuration. This is calculated in the *MCP251XFD_CalculateBitrateStatistics()* function. Should be as close as possible to 80%.

uint32_t *DSamplePoint*

Data Sample Point of the bit time configuration. This is calculated in the *MCP251XFD_CalculateBitrateStatistics()* function. Should be as close as possible to 80%.

uint32_t *OscTolC1*

Condition 1 for the maximum tolerance of the oscillator (Equation 3-12 of MCP25XXFD Family Reference Manual) of the bit time configuration. This is calculated in the *MCP251XFD_CalculateBitrateStatistics()* function.

uint32_t *OscTolC2*

Condition 2 for the maximum tolerance of the oscillator (Equation 3-13 of MCP25XXFD Family Reference Manual) of the bit time configuration. This is calculated in the *MCP251XFD_CalculateBitrateStatistics()* function.

uint32_t *OscTolC3*

Condition 3 for the maximum tolerance of the oscillator (Equation 3-14 of MCP25XXFD Family Reference Manual) of the bit time configuration. This is calculated in the *MCP251XFD_CalculateBitrateStatistics()* function.

uint32_t *OscTolC4*

Condition 4 for the maximum tolerance of the oscillator (Equation 3-15 of MCP25XXFD Family Reference Manual) of the bit time configuration. This is calculated in the *MCP251XFD_CalculateBitrateStatistics()* function.

uint32_t *OscTolC5*

Condition 5 for the maximum tolerance of the oscillator (Equation 3-16 of MCP25XXFD Family Reference Manual) of the bit time configuration. This is calculated in the *MCP251XFD_CalculateBitrateStatistics()* function.

uint32_t *OscTolerance*

Oscillator Tolerance, minimum of conditions 1-5 (Equation 3-11 of MCP25XXFD Family Reference Manual) of the bit time configuration. This is calculated in the *MCP251XFD_CalculateBitrateStatistics()* function.

14.6.   RAM FIFO information structure

The MCP251XFD RAM FIFO information structure contains all information about the RAM composition of a FIFO of the device when configured. This structure is filled by *MCP251XFD_ConfigureTEF()*, *MCP251XFD_ConfigureTXQ()* and *MCP251XFD_ConfigureFIFO()* functions. With TXQ and FIFO functions, the start address is set to 0 because it is impossible to know the actual start in RAM without knowing all the others configured FIFO. With the TEF it is easy, this FIFO is always the first.

If you want the complete RAM structure, create an array of *MCP251XFD_FIFO* with *MCP251XFD_FIFO*.RAMInfos of each FIFO configuration linked to a *MCP251XFD_RAMInfos* structure and use the *MCP251XFD_ConfigureFIFOList()* instead.

Source code:

```
typedef struct MCP251XFD_RAMInfos
{
  uint16_t ByteInFIFO;
  uint16_t RAMStartAddress;
  uint8_t  ByteInObject;
} MCP251XFD_RAMInfos;
```

14.6.1. Data fields

uint16_t *ByteInFIFO*

Total number of bytes that FIFO takes in RAM.

uint16_t *RAMStartAddress*

RAM Start Address of the FIFO.

uint8_t *ByteInObject*

How many bytes in an object of the FIFO.

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

14.7.   Function's return error enumerator

enum **eERRORRESULT**

There is only one error code at the same time returned by the functions. The only code that indicates that all went fine is ERR_OK.

Enumerator

ERR_OK	**0**	Succeeded
ERR__NO_DEVICE_DETECTED	**1**	No device detected
ERR__OUT_OF_RANGE	**2**	Value out of range
ERR__UNKNOWN_ELEMENT	**3**	Unknown element (type or value)
ERR__CONFIGURATION	**4**	Configuration error
ERR__NOT_SUPPORTED	**5**	Not supported
ERR__OUT_OF_MEMORY	**6**	Out of memory
ERR__NOT_AVAILABLE	**7**	Function not available
ERR__DEVICE_TIMEOUT	**8**	Device timeout
ERR__PARAMETER_ERROR	**9**	Parameter error
ERR__NO_DATA_AVAILABLE	**11**	No data available
ERR__FREQUENCY_ERROR	**12**	Frequency error
ERR__CRC_ERROR	**13**	CRC mismatch error
ERR__BAUDRATE_ERROR	**14**	Baudrate error
ERR__NOT_IN_SLEEP_MODE	**90**	Operation impossible in sleep mode
ERR__ALREADY_IN_SLEEP	**91**	Already in sleep mode
ERR__NEED_CONFIG_MODE	**93**	Device not in configuration mode
ERR__RAM_TEST_FAIL	**100**	RAM test fail
ERR__BITTIME_ERROR	**101**	Can't calculate a good Bit Time
ERR__TOO_MANY_TEF	**102**	Too many TEF to configure
ERR__TOO_MANY_TXQ	**103**	Too many TXQ to configure
ERR__TOO_MANY_FIFO	**104**	Too many FIFO to configure
ERR__SID11_NOT_AVAILABLE	**105**	SID11 not available in CAN2.0 mode
ERR__FILTER_CONSISTENCY	**106**	Filter inconsistency between Mask and filter
ERR__FILTER_TOO_LARGE	**107**	Filter too large between filter and config
ERR__BYTE_COUNT_MODULO_4	**108**	Byte count should be modulo 4
ERR__SPI_PARAMETER_ERROR	**200**	SPI parameter error
ERR__SPI_COMM_ERROR	**201**	SPI communication error
ERR__SPI_CONFIG_ERROR	**202**	SPI configuration error
ERR__SPI_TIMEOUT	**203**	SPI communication timeout
ERR__SPI_FREQUENCY_ERROR	**204**	SPI frequency error
ERR__TEST_ERROR	**255**	Test error

Can be widely distributed
Guide / **MCP251XFD driver library**
© Copyright 2020 Fabien MAILLY – All rights reserved

Page 57 of 112
Ref: MCP251XFD driver library guide
(Synchronous driver).docx

15. DRIVER'S FUNCTIONS

Here is the use of all functions related to the driver.

15.1. Init and reset

```
eERRORRESULT Init_MCP251XFD(
      MCP251XFD *pComp,
      const MCP251XFD_Config *pConf)
```

This function initializes the MCP251XFD driver and call the initialization of the interface driver (SPI). It checks parameters and perform a RESET. Next this function configures the MCP251XFD Controller and the CAN controller.

Warning

This function has to be used after component power-on otherwise the reset function call can fail when the component is used with MCP251XFD_DRIVER_SAFE_RESET

Parameters

Input	*pComp	Is the pointed structure of the device to be initialized
Input	*pConf	Is the pointed structure of the device configuration

Return

Returns an *eERRORRESULT* value enumerator. Below are some returned by the function itself but not errors returned by called functions.

- ERR__PARAMETER_ERROR when pComp or pConf is NULL, or Interface functions are NULL
- ERR__FREQUENCY_ERROR when XtalFreq or OscFreq parameters are out of range, or SysclkConfig try to set an out of range frequency
- ERR__CONFIGURATION when both XtalFreq and OscFreq are configured to 0
- ERR__NO_DEVICE_DETECTED when no communication with the device is possible
- ERR__DEVICE_TIMEOUT when the device does not respond
- ERR__SPI_FREQUENCY_ERROR when the SPI clock is too high compared to SYSCLK (max SYSCLK / 2)
- ERR__RAM_TEST_FAIL when the driver's internal device RAM test failed

```
eERRORRESULT MCP251XFD_ResetDevice(MCP251XFD *pComp)
```

Reset the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be reset

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

15.2. Read from RAM and registers

```
eERRORRESULT MCP251XFD_ReadData(
      MCP251XFD *pComp,
      uint16_t address,
      uint8_t* data,
      uint16_t size)
```

Read data from the MCP251XFD. In case of data reading data from the RAM, the size should be modulo 4.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the address where data will be read in the MCP251XFD (address will be incremented automatically)
Output	*data	Is where the data will be stored
Input	size	Is the size of the data array to read

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp or data is NULL, or Interface functions are NULL, or Address is too high
- ERR__OUT_OF_RANGE when you want to send a not multiple of 4 data to RAM
- ERR__CRC_ERROR when the CRC returned by the device is wrong

　　Ref: MCP251XFD driver library guide (Synchronous driver).docx

```
eERRORRESULT MCP251XFD_ReadSFR8(
      MCP251XFD *pComp,
      uint16_t address,
      uint8_t* data)
```
Read a byte data from an SFR register of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the SFR address where data will be read in the MCP251XFD
Output	*data	Is where the data will be stored

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_ReadData()'s returns value for more information.

```
eERRORRESULT MCP251XFD_ReadSFR16(
      MCP251XFD *pComp,
      uint16_t address,
      uint8_t* data)
```
Read a 2-bytes data from an SFR address of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the SFR address where data will be read in the MCP251XFD
Output	*data	Is where the data will be stored

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_ReadData()'s returns value for more information.

```
eERRORRESULT MCP251XFD_ReadSFR32(
      MCP251XFD *pComp,
      uint16_t address,
      uint8_t* data)
```
Read a word data (4 bytes) from an SFR address of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the SFR address where data will be read in the MCP251XFD
Output	*data	Is where the data will be stored

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_ReadData()'s returns value for more information.

```
eERRORRESULT MCP251XFD_ReadRAM32(
      MCP251XFD *pComp,
      uint16_t address,
      uint8_t* data)
```
Read a word data (4 bytes) from a RAM address of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the SFR address where data will be read in the MCP251XFD
Output	*data	Is where the data will be stored

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_ReadData()'s returns value for more information.

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

15.3.   Write to RAM and registers

```
eERRORRESULT MCP251XFD_WriteData(
      MCP251XFD *pComp,
      uint16_t address,
      uint8_t* data,
      uint16_t size)
```

Write data to the MCP251XFD. In case of data writing data to the RAM, the size should be modulo 4.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the address where data will be written in the MCP251XFD (address will be incremented automatically)
Input	*data	Is the data array to write
Input	size	Is the size of the data array to store

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp or data is NULL, or Interface functions are NULL, or Address is too high
- ERR__OUT_OF_RANGE when you want to send a not multiple of 4 data to RAM


```
eERRORRESULT MCP251XFD_WriteSFR8(
      MCP251XFD *pComp,
      uint16_t address,
      uint8_t* data,
      uint16_t size)
```

Write a byte data to an SFR register of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the address where data will be written in the MCP251XFD
Input	*data	Is the data array to store

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_WriteData()'s returns value for more information.


```
eERRORRESULT MCP251XFD_WriteSFR16(
      MCP251XFD *pComp,
      uint16_t address,
      uint8_t* data,
      uint16_t size)
```

Write a 2-bytes data to an SFR register of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the address where data will be written in the MCP251XFD
Input	*data	Is the data array to store

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_WriteData()'s returns value for more information.


```
eERRORRESULT MCP251XFD_WriteSFR32(
      MCP251XFD *pComp,
      uint16_t address,
      uint8_t* data,
      uint16_t size)
```

Write a word data (4 bytes) to an SFR register of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the address where data will be written in the MCP251XFD
Input	*data	Is the data array to store

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_WriteData()'s returns value for more information.

```
eERRORRESULT MCP251XFD_WriteRAM32(
      MCP251XFD *pComp,
      uint16_t address,
      uint8_t* data,
      uint16_t size)
```
Write a word data (4 bytes) to a RAM register of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	address	Is the address where data will be written in the MCP251XFD
Input	*data	Is the data array to store

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_WriteData()'s returns value for more information.

15.4.   Device ID

```
eERRORRESULT MCP251XFD_GetDeviceID(
      MCP251XFD *pComp,
      eMCP251XFD_Devices* device,
      uint8_t* deviceId,
      uint8_t* deviceRev)
```
Get actual device of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*device	Is the device found (MCP2517FD or MCP2518FD)
Output	*deviceId	Is the returned device ID (This parameter can be NULL if not needed)
Output	*deviceRev	Is the returned device Revision (This parameter can be NULL if not needed)

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp or device is NULL, or Interface functions are NULL

15.4.1. Enumerators

enum eMCP251XFD_Devices

List of supported devices.

Enumerator

MCP2517FD	0x00	Device MCP2517FD
MCP2518FD	0x01	Device MCP2518FD

15.5. Messages

15.5.1. Transmit messages

```
eERRORRESULT MCP251XFD_TransmitMessageObjectToFIFO(
        MCP251XFD *pComp,
        uint8_t* messageObjectToSend,
        uint8_t objectSize,
        eMCP251XFD_FIFO toFIFO,
        bool andFlush)
```

Transmit the message to the specified FIFO. This function uses the specific format of the component (T0, T1, Ti (data)). This function gets the next address where to put the message object on, send it and update the head pointer.

Warning

This function does not check if the FIFO have a room for the message or if the FIFO is actually a transmit FIFO or the actual state of the FIFO

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	*messageObjectToSend	Is the message object to send with all its data
Input	objectSize	Is the size of the message object (with its data). This value needs to be modulo 4
Input	toFIFO	Is the name of the FIFO to fill
Input	andFlush	Indicate if the FIFO will be flush to the CAN bus right after this message

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp or messageObjectToSend is NULL, or Interface functions are NULL, or toFIFO is MCP251XFD_TEF.
- ERR__BYTE_COUNT_MODULO_4 when the objectSize parameter is not modulo 4

```
eERRORRESULT MCP251XFD_TransmitMessageObjectToTXQ(
        MCP251XFD *pComp,
        uint8_t* messageObjectToSend,
        uint8_t objectSize,
        bool andFlush)
```

Transmit the message to the TXQ. This function uses the specific format of the component (T0, T1, Ti (data)). This function gets the next address where to put the message object on, send it and update the head pointer.

Warning

This function does not check if the TXQ have a room for the message or the actual state of the TXQ.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	*messageObjectToSend	Is the message object to send with all its data
Input	objectSize	Is the size of the message object (with its data). This value needs to be modulo 4
Input	andFlush	Indicate if the TXQ will be flush to the CAN bus right after this message

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_TransmitMessageObjectToFIFO()'s returns value for more information.

```
eERRORRESULT MCP251XFD_TransmitMessageToFIFO(
      MCP251XFD *pComp,
      MCP251XFD_CANMessage* messageToSend,
      eMCP251XFD_FIFO toFIFO, bool andFlush)
```

Transmit the message to the specified FIFO. This function gets the next address where to put the message object on, send it and update the head pointer.

Warning

This function does not check if the FIFO have a room for the message or if the FIFO is actually a transmit FIFO or the actual state of the FIFO

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	*messageToSend	Is the message to send with all the data attached with
Input	toFIFO	Is the name of the FIFO to fill
Input	andFlush	Indicate if the FIFO will be flush to the CAN bus right after this message

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp or messageToSend is NULL, or Interface functions are NULL, or toFIFO is MCP251XFD_TEF.
- ERR__BYTE_COUNT_MODULO_4 when the objectSize parameter is not modulo 4
- ERR__NO_DATA_AVAILABLE when messageToSend->PayloadData is NULL and the messageToSend->DLC is more than 0 bytes

```
eERRORRESULT MCP251XFD_TransmitMessageToTXQ(
      MCP251XFD *pComp,
      MCP251XFD_CANMessage* messageToSend,
      bool andFlush)
```

Transmit the message to the TXQ. This function gets the next address where to put the message object on, send it and update the head pointer.

Warning

This function does not check if the TXQ have a room for the message or the actual state of the TXQ

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	*messageToSend	Is the message to send with all the data attached with
Input	andFlush	Indicate if the TXQ will be flush to the CAN bus right after this message

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_TransmitMessageToFIFO()'s returns value for more information.

15.5.2. Receive messages

```
eERRORRESULT MCP251XFD_ReceiveMessageObjectFromFIFO(
      MCP251XFD *pComp,
      uint8_t* messageObjectGet,
      uint8_t objectSize,
      eMCP251XFD_FIFO fromFIFO)
```

Receive the message from the specified FIFO. This function uses the specific format of the component (R0, R1, (R2,) Ri (data)). This function gets the next address where to get the message object from, get it and update the tail pointer.

Warning

This function does not check if the FIFO have a message pending or if the FIFO is a receive FIFO or the actual state of the FIFO or if the TimeStamp is set or not

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*messageObjectGet	Is the message object retrieve with all its data
Input	objectSize	Is the size of the message object (with its data). This value needs to be modulo 4
Input	fromFIFO	Is the name of the FIFO to extract

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp or messageObjectGet is NULL, or Interface functions are NULL, or fromFIFO is MCP251XFD_TXQ.
- ERR__BYTE_COUNT_MODULO_4 when the objectSize parameter is not modulo 4

```
eERRORRESULT MCP251XFD_ReceiveMessageObjectFromTEF(
      MCP251XFD *pComp,
      uint8_t* messageObjectGet,
      uint8_t objectSize,
      eMCP251XFD_FIFO fromFIFO)
```

Receive a message from the TEF. This function uses the specific format of the component (TE0, TE1 (,TE2)). This function gets the next address where to get the message object from, get it and update the tail pointer.

Warning

This function does not check if the TEF have a message pending or the actual state of the TEF or if the TimeStamp is set or not

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*messageObjectGet	Is the message object retrieve with all its data
Input	objectSize	Is the size of the message object (with its data). This value needs to be modulo 4

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_ReceiveMessageObjectFromFIFO()'s returns value for more information.

```
eERRORRESULT MCP251XFD_ReceiveMessageFromFIFO(
      MCP251XFD *pComp,
      MCP251XFD_CANMessage* messageGet,
      eMCP251XFD_PayloadSize payloadSize,
      uint32_t* timeStamp,
      eMCP251XFD_FIFO fromFIFO)
```

Receive a message from the specified FIFO. This function gets the next address where to get the message object from, get it and update the tail pointer.

Warning

This function does not check if the FIFO have a message pending or if the FIFO is a receive FIFO or the actual state of the FIFO or if the TimeStamp is set or not

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*messageGet	Is the message retrieve with all the data attached with
Input	payloadSize	Indicate the payload of the FIFO (8, 12, 16, 20, 24, 32, 48 or 64)
Output	*timeStamp	Is the returned TimeStamp of the message (can be set to NULL if the TimeStamp is not set in this FIFO)
Input	fromFIFO	Is the name of the FIFO to extract

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp or messageGet is NULL, or Interface functions are NULL, or fromFIFO is MCP251XFD_TXQ.
- ERR__BYTE_COUNT_MODULO_4 when the objectSize parameter is not modulo 4
- ERR__NO_DATA_AVAILABLE when messageGet->PayloadData is NULL and the messageGet->DLC is more than 0 bytes

```
eERRORRESULT MCP251XFD_ReceiveMessageFromTEF(
      MCP251XFD *pComp,
      MCP251XFD_CANMessage* messageGet,
      uint32_t* timeStamp)
```

Receive a message from the TEF. This function gets the next address where to get the message object from, get it and update the tail pointer.

Warning

This function does not check if the TEF have a message pending or the actual state of the TEF or if the TimeStamp is set or not

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*messageGet	Is the message retrieve with all the data attached with
Output	*timeStamp	Is the returned TimeStamp of the message (can be set to NULL if the TimeStamp is not set in the TEF)

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_ReceiveMessageFromFIFO()'s returns value for more information.

The CAN message configuration structure is used as a simple structure to simplify the configuration or reading of messages. This structure avoids the complex header messages filling and decoding. This structure is used by *MCP251XFD_TransmitMessageToFIFO()*, *MCP251XFD_TransmitMessageToTXQ()*, *MCP251XFD_ReceiveMessageFromFIFO()* and *MCP251XFD_ReceiveMessageFromTEF()* functions.

Source code:

```
typedef struct MCP251XFD_CANMessage
{
  uint32_t MessageID;
  uint32_t MessageSEQ;
  setMCP251XFD_MessageCtrlFlags ControlFlags;
  eMCP251XFD_DataLength DLC;
  uint8_t* PayloadData;
} MCP251XFD_CANMessage;
```

15.5.3.1. Data fields

uint32_t MessageID

Contain the message ID to send.

uint32_t MessageSEQ

This is the context of the CAN message. This sequence will be copied in the TEF to trace the message sent.

setMCP251XFD_MessageCtrlFlags ControlFlags

Contain the CAN controls flags.

Type

enum   setMCP251XFD_MessageCtrlFlags

Initial value / default

MCP251XFD_NO_MESSAGE_CTRL_FLAGS

eMCP251XFD_DataLength DLC

Indicate how many bytes in the payload data will be sent or how many bytes in the payload data is received.

Type

enum   eMCP251XFD_DataLength

uint8_t* PayloadData

Pointer to the payload data that will be sent. PayloadData array should be at least the same size as indicate by the DLC.

15.5.4. Enumerators

enum eMCP251XFD_MessageCtrlFlags
typedef eMCP251XFD_MessageCtrlFlags setMCP251XFD_MessageCtrlFlags

Control flags of CAN message.

Enumerator

MCP251XFD_NO_MESSAGE_CTRL_FLAGS	**0x00**	No Message Control Flags
MCP251XFD_CAN20_FRAME	**0x00**	Indicate that the frame is a CAN2.0A/B
MCP251XFD_CANFD_FRAME	**0x01**	Indicate that the frame is a CAN-FD
MCP251XFD_NO_SWITCH_BITRATE	**0x00**	The data bitrate is not switched (only CAN-FD frame)
MCP251XFD_SWITCH_BITRATE	**0x02**	The data bitrate is switched (only CAN-FD frame)
MCP251XFD_REMOTE_TRANSMISSION_REQUEST	**0x04**	The frame is a Remote Transmission Request; not used in CAN FD
MCP251XFD_STANDARD_MESSAGE_ID	**0x00**	Clear the Identifier Extension Flag that set the standard ID format
MCP251XFD_EXTENDED_MESSAGE_ID	**0x08**	Set the Identifier Extension Flag that set the extended ID format
MCP251XFD_TRANSMIT_ERROR_PASSIVE	**0x10**	Error Status Indicator: In CAN to CAN gateway mode (CiCON.ESIGM=1), the transmitted ESI flag is a "logical OR" of T1.ESI and error passive state of the CAN controller; In normal mode ESI indicates the error status

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

enum **eMCP251XFD_DataLength**

Data Length Size for the CAN message.

Enumerator

MCP251XFD_DLC_0BYTE	**0b0000**	The DLC is 0 data byte
MCP251XFD_DLC_1BYTE	**0b0001**	The DLC is 1 data byte
MCP251XFD_DLC_2BYTE	**0b0010**	The DLC is 2 data bytes
MCP251XFD_DLC_3BYTE	**0b0011**	The DLC is 3 data bytes
MCP251XFD_DLC_4BYTE	**0b0100**	The DLC is 4 data bytes
MCP251XFD_DLC_5BYTE	**0b0101**	The DLC is 5 data bytes
MCP251XFD_DLC_6BYTE	**0b0110**	The DLC is 6 data bytes
MCP251XFD_DLC_7BYTE	**0b0111**	The DLC is 7 data bytes
MCP251XFD_DLC_8BYTE	**0b1000**	The DLC is 8 data bytes
MCP251XFD_DLC_12BYTE	**0b1001**	The DLC is 12 data bytes
MCP251XFD_DLC_16BYTE	**0b1010**	The DLC is 16 data bytes
MCP251XFD_DLC_20BYTE	**0b1011**	The DLC is 20 data bytes
MCP251XFD_DLC_24BYTE	**0b1100**	The DLC is 24 data bytes
MCP251XFD_DLC_32BYTE	**0b1101**	The DLC is 32 data bytes
MCP251XFD_DLC_48BYTE	**0b1110**	The DLC is 48 data bytes
MCP251XFD_DLC_64BYTE	**0b1111**	The DLC is 64 data bytes
MCP251XFD_PAYLOAD_MAX	**64**	This is the max payload byte size

15.5.5. CAN Transmit message Object Identifier structure

The CAN transmit message Object Identifier structure is an image of the frame header object of the frame in RAM which is 8 bytes size.

Source code:

```
typedef union MCP251XFD_CAN_TX_Message
{
  uint32_t Word[2];
  uint8_t Bytes[8];
  struct
  {
    MCP251XFD_CAN_TX_Message_Identifier T0;
    MCP251XFD_CAN_TX_Message_Control    T1;
  };
} MCP251XFD_CAN_TX_Message;
```

15.5.5.1. Data fields

*uint32_t **Word**[2]*

Words access of the MCP251XFD_CAN_TX_Message. The first is a MCP251XFD_CAN_TX_Message_Identifier and the second is a MCP251XFD_CAN_TX_Message_Control.

*uint8_t **Bytes**[8]*

Bytes access of the MCP251XFD_CAN_TX_Message.

*MCP251XFD_CAN_TX_Message_Identifier **T0***

CAN Transmit Message Identifier (T0).

Type

    struct   MCP251XFD_CAN_TX_Message_Identifier

*MCP251XFD_CAN_TX_Message_Control **T1***

CAN Transmit Message Control Field (T1).

Type

    struct   MCP251XFD_CAN_TX_Message_Control

15.5.5.2. Structures

15.5.5.2.1. CAN Transmit Message Identifier (T0)

The CAN Transmit Message Identifier (T0) structure is an image of the first 4-bytes in RAM of the frame header object of the frame in RAM. Source code:

```
typedef union MCP251XFD_CAN_TX_Message_Identifier
{
  uint32_t T0;
  uint8_t Bytes[4];
  struct
  {
    uint32_t SID  : 11;
    uint32_t EID  : 18;
    uint32_t SID11:  1;
    uint32_t       :  2;
  };
} MCP251XFD_CAN_TX_Message_Identifier;
```

15.5.5.2.1.1. Data fields

uint32_t T0
Word access of the MCP251XFD_CAN_TX_Message_Identifier.

uint8_t Bytes
Bytes access of the MCP251XFD_CAN_TX_Message_Identifier.

uint32_t SID
Standard Identifier of the frame.

uint32_t EID
Extended Identifier of the frame.

uint32_t SID11
In FD mode the standard ID can be extended to 12 bits using RRS.

15.5.5.2.2. CAN Transmit Message Control Field (T1)

The CAN Transmit Message Control Field (T1) structure is an image of the second 4-bytes in RAM of the frame header object of the frame in RAM.
Source code:

```
typedef union __PACKED__ MCP251XFD_CAN_TX_Message_Control
{
  uint32_t T1;
  uint8_t Bytes[4];
  struct
  {
    uint32_t DLC:  4;
    uint32_t IDE:  1;
    uint32_t RTR:  1;
    uint32_t BRS:  1;
    uint32_t FDF:  1;
    uint32_t ESI:  1;
    uint32_t SEQ: 23;
  };
} MCP251XFD_CAN_TX_Message_Control;
```

15.5.5.2.2.1. Data fields

uint32_t T1
Word access of the MCP251XFD_CAN_TX_Message_Control.

uint8_t Bytes
Bytes access of the MCP251XFD_CAN_TX_Message_Control.

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

uint32_t DLC

Data Length Code.

uint32_t IDE

Identifier Extension Flag; distinguishes between base and extended format.

uint32_t RTR

Remote Transmission Request; not used in CAN-FD.

uint32_t BRS

Bit Rate Switch; selects if data bit rate is switched.

uint32_t FDF

FD Frame; distinguishes between CAN and CAN-FD formats.

uint32_t ESI

Error Status Indicator. In CAN to CAN gateway mode (CiCON.ESIGM=1), the transmitted ESI flag is a "logical OR" of T1.ESI and error passive state of the CAN controller. In normal mode ESI indicates the error status: '1' = Transmitting node is error passive ; '0' = Transmitting node is error active.

uint32_t SEQ

Sequence to keep track of transmitted messages in Transmit Event FIFO (Only bit <6:0> for the MCP2517X. Bits <22:7> should be at '0').

15.5.6. CAN Receive message Object Identifier structure

The CAN receive message Object Identifier structure is an image of the frame header object of the frame in RAM which is 8 bytes size. To this is added the TimeStamp as R2 but not mandatory.

Source code:

```
typedef union MCP251XFD_CAN_RX_Message
{
  uint32_t Word[2];
  uint8_t Bytes[8];
  struct
  {
    MCP251XFD_CAN_RX_Message_Identifier R0;
    MCP251XFD_CAN_RX_Message_Control    R1;
    uint32_t TimeStamp;
  };
} MCP251XFD_CAN_RX_Message;
```

15.5.6.1. Data fields

uint32_t Word[2]

Words access of the MCP251XFD_CAN_RX_Message. The first is a MCP251XFD_CAN_RX_Message_Identifier and the second is a MCP251XFD_CAN_RX_Message_Control.

uint8_t Bytes[8]

Bytes access of the MCP251XFD_CAN_RX_Message.

MCP251XFD_CAN_RX_Message_Identifier R0

CAN Receive Message Identifier (R0).

Type

    struct   MCP251XFD_CAN_RX_Message_Identifier

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

MCP251XFD_CAN_RX_Message_Control *R1*

CAN Receive Message Control Field (R1).

Type

    struct   MCP251XFD_CAN_RX_Message_Control

uint32_t *TimeStamp*

Transmit Message Time Stamp. R2 (RXMSGTS) only exits in objects where CiFIFOCONm.RXTSEN is set.

15.5.6.2. Structures

15.5.6.2.1. CAN Receive Message Identifier (R0)

The CAN Receive Message Identifier (R0) structure is an image of the first 4-bytes in RAM of the frame header object of the frame in RAM. Source code:

```c
typedef union __PACKED__ MCP251XFD_CAN_RX_Message_Identifier
{
  uint32_t R0;
  uint8_t Bytes[4];
  struct
  {
    uint32_t SID  : 11; //!<  0-10 - Standard Identifier
    uint32_t EID  : 18; //!< 11-28 - Extended Identifier
    uint32_t SID11:  1; //!< 29    - In FD mode the standard ID can be extended to 12 bit using RRS
    uint32_t      :  2; //!< 30-31
  };
} MCP251XFD_CAN_RX_Message_Identifier;
```

15.5.6.2.1.1. Data fields

uint32_t *R0*

Word access of the MCP251XFD_CAN_RX_Message_Identifier.

uint8_t *Bytes*

Bytes access of the MCP251XFD_CAN_RX_Message_Identifier.

uint32_t *SID*

Standard Identifier of the frame.

uint32_t *EID*

Extended Identifier of the frame.

uint32_t *SID11*

In FD mode the standard ID can be extended to 12 bits using RRS.

15.5.6.2.2. CAN Receive Message Control Field (R1)

The CAN Receive Message Control Field (R1) structure is an image of the second 4-bytes in RAM of the frame header object of the frame in RAM.
Source code:

```c
typedef union __PACKED__ MCP251XFD_CAN_RX_Message_Control
{
  uint32_t R1;
  uint8_t Bytes[4];
  struct
  {
    uint32_t DLC    :  4;
    uint32_t IDE    :  1;
    uint32_t RTR    :  1;
    uint32_t BRS    :  1;
    uint32_t FDF    :  1;
    uint32_t ESI    :  1;
    uint32_t        :  2;
    uint32_t FILTHIT:  5;
    uint32_t        : 16;
  };
} MCP251XFD_CAN_RX_Message_Control;
```

15.5.6.2.2.1. Data fields

*uint32_t **R1***

Word access of the MCP251XFD_CAN_RX_Message_Control.

*uint8_t **Bytes***

Bytes access of the MCP251XFD_CAN_RX_Message_Control.

*uint32_t **DLC***

Data Length Code.

*uint32_t **IDE***

Identifier Extension Flag; distinguishes between base and extended format.

*uint32_t **RTR***

Remote Transmission Request; not used in CAN-FD.

*uint32_t **BRS***

Bit Rate Switch; selects if data bit rate is switched.

*uint32_t **FDF***

FD Frame; distinguishes between CAN and CAN-FD formats.

*uint32_t **ESI***

Error Status Indicator: '1' = Transmitting node is error passive ; '0' = Transmitting node is error active.

*uint32_t **FILTHIT***

Filter Hit, number of filters that matched.

15.5.7. CAN Transmit message Object Identifier structure

The Transmit Event Object Register (TEF) structure is an image of the frame header object of the frame in RAM which is 8 bytes size.

Source code:

```
typedef union __PACKED__ MCP251XFD_CAN_TX_EventObject
{
  uint32_t Word[2];
  uint8_t Bytes[8];
  struct
  {
    MCP251XFD_CAN_TX_Message_Identifier TE0;
    MCP251XFD_CAN_TX_Message_Control    TE1;
    uint32_t TimeStamp;
  };
} MCP251XFD_CAN_TX_EventObject;
```

15.5.7.1. Data fields

*uint32_t **Word[2]***

Words access of the MCP251XFD_CAN_TX_EventObject. The first is a MCP251XFD_CAN_TX_Message_Identifier and the second is a MCP251XFD_CAN_TX_Message_Control.

*uint8_t **Bytes[8]***

Bytes access of the MCP251XFD_CAN_TX_EventObject.

MCP251XFD_CAN_TX_Message_Identifier *TE0*

CAN Transmit Event Object Identifier (TE0).

Type

    struct   MCP251XFD_CAN_TX_Message_Identifier

MCP251XFD_CAN_TX_Message_Control *TE1*

CAN Transmit Event Object Control Field (TE1).

Type

    struct   MCP251XFD_CAN_TX_Message_Control

uint32_t *Word[2]*

Transmit Message Time Stamp. TE2 (TXMSGTS) only exists in objects where CiTEFCON.TEFTSEN is set.

15.6. CRC

```
eERRORRESULT MCP251XFD_ConfigureCRC(
      MCP251XFD *pComp,
      setMCP251XFD_CRCEvents interrupts)
```

At initialization if the driver has the flag MCP251XFD_DRIVER_USE_READ_WRITE_CRC then all CRC interrupts are enabled.

Parameters

Input	*pComp	Is the pointed structure of the device where the CRC will be configured
Input	interrupts	Corresponding bit to '1' enable the interrupt. Flags can be OR'ed

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
eERRORRESULT MCP251XFD_GetCRCEvents(
      MCP251XFD *pComp,
      setMCP251XFD_CRCEvents* events,
      uint16_t* lastCRCMismatch)
```

Get CRC Status of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device where the CRC status will be obtained
Output	*events	Is the return value of current events flags of the CRC. Flags can be OR'ed
Output	*lastCRCMismatch	Is the Cycle Redundancy Check from last CRC mismatch. This parameter can be NULL if the last mismatch is not needed

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp or events is NULL, or Interface functions are NULL

```
eERRORRESULT MCP251XFD_ClearCRCEvents(MCP251XFD *pComp)
```

Clear CRC Status Flags of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device where the CRC status will be cleared

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

15.6.1. Enumerators

```
enum eMCP251XFD_CRCEvents
typedef eMCP251XFD_CRCEvents setMCP251XFD_CRCEvents
```

CRC Events.

Enumerator

MCP251XFD_CRC_NO_EVENT	0x00	No CRC events
MCP251XFD_CRC_CRCERR_EVENT	0x01	CRC Error Interrupt event
MCP251XFD_CRC_FORMERR_EVENT	0x02	CRC Command Format Error Interrupt event
MCP251XFD_CRC_ALL_EVENTS	0x03	All CRC interrupts events

15.7. ECC

```
eERRORRESULT MCP251XFD_ConfigureECC(
    MCP251XFD *pComp,
    bool enableECC,
    setMCP251XFD_ECCEvents interrupts,
    uint8_t fixedParityValue)
```

At initialization if the driver has the flag MCP251XFD_DRIVER_ENABLE_ECC then ECC is enable and all ECC interrupts are enable.

Warning

If the ECC is configured after use of the device without reset, ECC interrupts can occur if the RAM is not fully initialize by setting MCP251XFD_DRIVER_INIT_SET_RAM_AT_0 in MCP251XFD::DriverConfig)

Parameters

Input	*pComp	Is the pointed structure of the device where the ECC will be configured
Input	enableECC	Is at 'true' to enable ECC or 'false' to disable ECC
Input	interrupts	Corresponding bit to '1' enable the interrupt. Flags can be OR'ed
Input	fixedParityValue	Is the value of the parity bits used during write to RAM when ECC is disabled

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
eERRORRESULT MCP251XFD_GetECCEvents(
    MCP251XFD *pComp,
    setMCP251XFD_ECCEvents* events,
    uint16_t* lastErrorAddress)
```

Get ECC Status Flags of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device where the ECC status will be obtained
Output	*events	Is the return value of current events flags of the ECC. Flags can be OR'ed
Output	*lastErrorAddress	Is the address where last ECC error occurred. This parameter can be NULL if the address is not needed

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp or events is NULL, or Interface functions are NULL

```
eERRORRESULT MCP251XFD_ClearECCEvents(MCP251XFD *pComp)
```

Clear ECC Status Flags of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device where the ECC status will be cleared

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

15.7.1. Enumerators

```
enum eMCP251XFD_ECCEvents
typedef eMCP251XFD_ECCEvents setMCP251XFD_ECCEvents
```

CRC Events.

Enumerator

MCP251XFD_ECC_NO_EVENT	0x00	No ECC events
MCP251XFD_ECC_SEC_EVENT	0x02	ECC Single Error Correction Interrupt
MCP251XFD_ECC_DED_EVENT	0x04	ECC Double Error Detection Interrupt
MCP251XFD_ECC_ALL_EVENTS	0x06	All ECC interrupts events

15.8. Pin configuration

```
eERRORRESULT MCP251XFD_ConfigurePins(
      MCP251XFD *pComp,
      eMCP251XFD_GPIO0Mode GPIO0PinMode,
      eMCP251XFD_GPIO1Mode GPIO1PinMode,
      eMCP251XFD_OutMode INTOutMode,
      eMCP251XFD_OutMode TXCANOutMode,
      bool CLKOasSOF)
```

Configure pins of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be configured
Input	GPIO0PinMode	Set the INT0/GPIO0/XSTBY pins mode
Input	GPIO1PinMode	Set the INT1/GPIO1 pins mode
Input	INTOutMode	Set the INTs (INT, INT0 and INT1) output pin mode
Input	TXCANOutMode	Set the INT0/GPIO0/XSTBY pins mode
Input	CLKOasSOF	If 'true', then SOF signal is on CLKO pin else it is a clock on CLKO pin

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
eERRORRESULT MCP251XFD_SetGPIOPinsDirection(
      MCP251XFD *pComp,
      uint8_t pinsDirection,
      uint8_t pinsChangeMask)
```

Set GPIO pins direction of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be configured
Input	pinsDirection	Return the actual level of all I/O pins. If bit is '1' then the corresponding GPIO is level high else, it is level low
Input	pinsChangeMask	If the bit is set to '1', then the corresponding GPIO must be modified

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
eERRORRESULT MCP251XFD_GetGPIOPinsInputLevel(
      MCP251XFD *pComp,
      uint8_t *pinsState)
```

Get GPIO pins input level of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be configured
Input	*pinsState	Return the actual level of all I/O pins. If bit is '1' then the corresponding GPIO is level high else, it is level low

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp or pinsState is NULL, or Interface functions are NULL

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

```
eERRORRESULT MCP251XFD_SetGPIOPinsOutputLevel(
     MCP251XFD *pComp,
     uint8_t pinsLevel,
     uint8_t pinsChangeMask)
```

Set GPIO pins output level of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be configured
Input	pinsLevel	Set the IO pins output level, if bit is '1' then the corresponding GPIO is level high else it is level low
Input	pinsChangeMask	If the bit is set to '1', then the corresponding GPIO must be modified

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

15.8.1. Enumerators

enum **eMCP251XFD_GPIO0Mode**

INT0/GPIO0/XSTBY configuration for the IOCON register.

Enumerator

MCP251XFD_PIN_AS_INT0_TX	**0b00**	INT0/GPIO0/XSTBY pin as TX Interrupt output (active low)
MCP251XFD_PIN_AS_GPIO0_IN	**0b01**	INT0/GPIO0/XSTBY pin as GPIO input
MCP251XFD_PIN_AS_GPIO0_OUT	**0b10**	INT0/GPIO0/XSTBY pin as GPIO output
MCP251XFD_PIN_AS_XSTBY	**0b11**	INT0/GPIO0/XSTBY pin as Transceiver Standby output

enum **eMCP251XFD_GPIO1Mode**

INT1/GPIO1 configuration for the IOCON register.

Enumerator

MCP251XFD_PIN_AS_INT1_RX	**0b00**	INT1/GPIO1 pin as RX Interrupt output (active low)
MCP251XFD_PIN_AS_GPIO1_IN	**0b01**	INT1/GPIO1 pin as GPIO input
MCP251XFD_PIN_AS_GPIO1_OUT	**0b10**	INT1/GPIO1 pin as GPIO output

enum **eMCP251XFD_OutMode**

Output configuration for the IOCON.INTOD and the IOCON.TXCANOD register.

Enumerator

MCP251XFD_PINS_PUSHPULL_OUT	**0b00**	Pin with Push/Pull output
MCP251XFD_PINS_OPENDRAIN_OUT	**0b01**	Pin with Open Drain output

15.8.2. Defines

#define MCP251XFD_GPIO0_Mask

Define the GPIO0 mask for the pinsChangeMask parameter.

Value

0b01

#define MCP251XFD_GPIO1_Mask

Define the GPIO1 mask for the pinsChangeMask parameter.

Value

0b10

#define MCP251XFD_GPIO0_OUTPUT

Define the GPIO0 pin direction as output for the pinsDirection parameter.

Value

0b00

#define MCP251XFD_GPIO0_INPUT

Define the GPIO0 pin direction as input for the pinsDirection parameter.

Value

0b01

#define MCP251XFD_GPIO1_OUTPUT

Define the GPIO1 pin direction as output for the pinsDirection parameter.

Value

0b00

#define MCP251XFD_GPIO1_INPUT

Define the GPIO1 pin direction as input for the pinsDirection parameter.

Value

0b10

#define MCP251XFD_GPIO0_LOW

Define the GPIO0 pin low status for the pinsLevel parameter.

Value

0b00

#define MCP251XFD_GPIO0_HIGH

Define the GPIO0 pin high status for the pinsLevel parameter.

Value

0b01

#define MCP251XFD_GPIO1_LOW

Define the GPIO1 pin low status for the pinsLevel parameter.

Value

0b00

#define MCP251XFD_GPIO1_HIGH

Define the GPIO1 pin high status for the pinsLevel parameter.

Value

0b10

15.9.   Bitrate and BitTime configuration

```
eERRORRESULT MCP251XFD_CalculateBitTimeConfiguration(
      const uint32_t fsysclk,
      const uint32_t desiredNominalBitrate,
      const uint32_t desiredDataBitrate,
      MCP251XFD_BitTimeConfig *pConf)
```

Calculate the best Bit Time configuration following desired bitrates for CAN-FD. This function call automatically the *MCP251XFD_CalculateBitrateStatistics()* function.

Parameters

Input	fsysclk	Is the SYSCLK of the device
Input	desiredNominalBitrate	Is the desired Nominal Bitrate of the CAN-FD configuration
Input	desiredDataBitrate	Is the desired Data Bitrate of the CAN-FD configuration (If not CAN-FD set it to 0 or MCP251XFD_NO_CANFD)
Output	*pConf	Is the pointed structure of the Bit Time configuration

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp or pConf is NULL, or Interface functions are NULL, or fsysclk is out of range
- ERR__BAUDRATE_ERROR when desiredNominalBitrate or desiredDataBitrate is out of range (when different to MCP251XFD_NO_CANFD)
- ERR__BITTIME_ERROR when it is impossible to find a proper BRP with this configuration

```
eERRORRESULT MCP251XFD_CalculateBitrateStatistics(
      const uint32_t fsysclk,
      MCP251XFD_BitTimeConfig *pConf,
      bool can20only)
```

Calculate bus length, sample points, bitrates and oscillator tolerance following BitTime Configuration.

Parameters

Input	fsysclk	Is the SYSCLK of the device
In/Out	*pConf	Is the pointed structure of the Bit Time configuration
Input	can20only	Indicate that parameters for Data Bitrate are not calculated if true

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp or pConf is NULL, or pConf->Stats is NULL, or Interface functions are NULL, or fsysclk is out of range

```
eERRORRESULT MCP251XFD_SetBitTimeConfiguration(
      MCP251XFD *pComp,
      MCP251XFD_BitTimeConfig *pConf,
      bool can20only)
```

Set the Nominal and Data Bit Time to registers.

Parameters

Input	*pComp	Is the pointed structure of the device to be configured
Input	*pConf	Is the pointed structure of the Bit Time configuration
Input	can20only	Indicate that parameters for Data Bitrate are not set to registers (CiDBTCFG and CiTDC)

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

15.9.1. Bit Time Configuration structure for CAN speed

This is the bit time configuration structure for nominal and data bitrates. It contains all the information to configure nominal and data bitrates for the CAN controller. This structure is used by *MCP251XFD_CalculateBitTimeConfiguration()*, *MCP251XFD_CalculateBitrateStatistics()* and *MCP251XFD_SetBitTimeConfiguration()* functions.

Source code:

```
typedef struct MCP251XFD_BitTimeConfig
{
  //--- Nominal Bit Times ---
  uint32_t NBRP;
  uint32_t NTSEG1;
  uint32_t NTSEG2;
  uint32_t NSJW;
  //--- Data Bit Times ---
  uint32_t DBRP;
  uint32_t DTSEG1;
  uint32_t DTSEG2;
  uint32_t DSJW;
  //--- Transmitter Delay Compensation ---
  eMCP251XFD_TDCMode TDCMOD;
  int32_t TDCO;
  uint32_t TDCV;
  bool EDGE_FILTER;
  //--- Result Statistics ---
  MCP251XFD_BitTimeStats *Stats;
} MCP251XFD_BitTimeConfig;
```

15.9.1.1. Data fields

uint32_t NBRP

Nominal Bit Times - Baud Rate Prescaler bits; TQ = value/Fsys.

uint32_t NTSEG1

Nominal Bit Times - Time Segment 1 bits (Propagation Segment + Phase Segment 1); Length is value x TQ.

uint32_t NTSEG2

Nominal Bit Times - Time Segment 2 bits (Phase Segment 2); Length is value x TQ.

uint32_t NSJW

Nominal Bit Times - Synchronization Jump Width bits; Length is value x TQ.

uint32_t DBRP

Data Bit Times - Baud Rate Prescaler bits; TQ = value/Fsys.

uint32_t DTSEG1

Data Bit Times - Time Segment 1 bits (Propagation Segment + Phase Segment 1); Length is value x TQ.

uint32_t DTSEG2

Data Bit Times - Time Segment 2 bits (Phase Segment 2); Length is value x TQ.

uint32_t DSJW

Data Bit Times - Synchronization Jump Width bits; Length is value x TQ.

eMCP251XFD_TDCMode TDCMOD

Transmitter Delay Compensation Mode; Secondary Sample Point (SSP).

Type

enum   *eMCP251XFD_TDCMode*

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

uint32_t TDCO

Transmitter Delay Compensation Offset; Secondary Sample Point (SSP). Two's complement: offset can be positive, zero, or negative (used as positive only here).

uint32_t TDCV

Transmitter Delay Compensation Value; Secondary Sample Point (SSP).

bool EDGE_FILTER

Enable Edge Filtering during Bus Integration state. In case of use of *MCP251XFD_CalculateBitTimeConfiguration()* with CAN-FD this parameter will be set to 'true'.

MCP251XFD_BitTimeStats *Stats

Point to a stat structure (set to NULL if no statistics are necessary).

15.9.2. Enumerators

enum eMCP251XFD_TDCMode

Control flags of CAN message.

Enumerator

MCP251XFD_TDC_DISABLED	0b00	TDC Disabled
MCP251XFD_MANUAL_MODE	0b01	Manual; Do not measure, use TDCV + TDCO from register
MCP251XFD_AUTO_MODE	0b10	Auto; measure delay and add TDCO

15.9.3. Defines

#define MCP251XFD_NO_CANFD

This value specify that the driver will not calculate CAN-FD bitrate.

Value

0

15.10. Operation modes and CAN control

---

eERRORRESULT MCP251XFD_AbortAllTransmissions*(MCP251XFD *pComp)*

Abort all pending transmissions of the MCP251XFD device.

Parameters

    Input   *pComp   Is the pointed structure of the device to be used

Return

    Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

---

eERRORRESULT MCP251XFD_GetActualOperationMode(
    MCP251XFD *pComp,
    eMCP251XFD_OperationMode* actualMode)

Get actual operation mode of the MCP251XFD device.

Parameters

    Input   *pComp   Is the pointed structure of the device to be used

    Output *actualMode   Is where the result of the actual mode will be saved

Return

    Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

---

eERRORRESULT MCP251XFD_RequestOperationMode(
    MCP251XFD *pComp,
    eMCP251XFD_OperationMode newMode,
    bool waitOperationChange)

Request operation mode change of the MCP251XFD device. In case of wait operation change, the function calls *MCP251XFD_WaitOperationModeChange()*.

Parameters

    Input   *pComp   Is the pointed structure of the device to be used

    Input   newMode   Is the new operational mode to set

    Input   waitOperationChange Set to 'true' if the function must wait for the actual operation mode change (wait up to 7ms)

Return

    Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL
- ERR__CONFIGURATION when you try to request a CAN-FD mode while the device is not configured for

---

eERRORRESULT MCP251XFD_WaitOperationModeChange(
    MCP251XFD *pComp,
    eMCP251XFD_OperationMode askedMode)

The function can wait up to 7ms. After this time, if the device doesn't change its operation mode, the function returns an ERR__DEVICE_TIMEOUT error.

Parameters

    Input   *pComp   Is the pointed structure of the device to be used

    Input   askedMode   Is the mode asked after a call of *MCP251XFD_RequestOperationMode()*

Return

    Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

        Ref: MCP251XFD driver library guide
(Synchronous driver).docx

```
inline eERRORRESULT MCP251XFD_StartCAN20(MCP251XFD *pComp)
```
Start the MCP251XFD device in CAN2.0 mode. This function asks for a mode change to CAN2.0 but do not wait for its actual change because normally the device is in configuration mode and the change to CAN2.0 mode will be instantaneous

Parameters

Input   *pComp   Is the pointed structure of the device to be used

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_RequestOperationMode()'s returns value for more information.

```
inline eERRORRESULT MCP251XFD_StartCANFD(MCP251XFD *pComp)
```
Start the MCP251XFD device in CAN-FD mode. This function asks for a mode change to CAN-FD but do not wait for its actual change because normally the device is in configuration mode and the change to CAN-FD mode will be instantaneous

Parameters

Input   *pComp   Is the pointed structure of the device to be used

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_RequestOperationMode()'s returns value for more information.

```
eERRORRESULT MCP251XFD_ConfigureCANController(
        MCP251XFD *pComp,
        setMCP251XFD_CANCtrlFlags flags,
        eMCP251XFD_Bandwidth bandwidth)
```
Configure CAN Controller of the MCP251XFD device.

Parameters

Input   *pComp   Is the pointed structure of the device to be used
Input   flags   Is all the flags for the configuration
Input   bandwidth   Is the Delay between two consecutive transmissions (in arbitration bit times)

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL
- ERR__NEED_CONFIG_MODE when you try to configure the device while not in configuration mode

15.10.1.   Enumerators

```
enum eMCP251XFD_OperationMode
```
CAN Controller Operation Modes for the CiCON.OPMOD and CiCON.REQOP registers.

Enumerator

MCP251XFD_NORMAL_CANFD_MODE	0b000	Set Normal CAN FD mode; supports mixing of CAN FD and Classic CAN 2.0 frames
MCP251XFD_SLEEP_MODE	0b001	Set Sleep mode
MCP251XFD_INTERNAL_LOOPBACK_MODE	0b010	Set Internal Loopback mode
MCP251XFD_LISTEN_ONLY_MODE	0b011	Set Listen Only mode
MCP251XFD_CONFIGURATION_MODE	0b100	Set Configuration mode
MCP251XFD_EXTERNAL_LOOPBACK_MODE	0b101	Set External Loopback mode
MCP251XFD_NORMAL_CAN20_MODE	0b110	Set Normal CAN 2.0 mode; possible error frames on CAN FD frames
MCP251XFD_RESTRICTED_OPERATION_MODE	0b111	Set Restricted Operation mode

Ref: MCP251XFD driver library guide (Synchronous driver).docx

15.11.　Sleep and Deep-Sleep modes

```
eERRORRESULT MCP251XFD_ConfigureSleepMode(
      MCP251XFD *pComp,
      bool useLowPowerMode,
      eMCP251XFD_WakeUpFilter wakeUpFilter,
      bool interruptBusWakeUp)
```

Sleep mode configuration of the MCP251XFD device. Sleep mode is a low-power mode, where register and RAM contents are preserved, and the clock is switched off. LPM is an Ultra-Low Power mode, where most of the chip is powered down. Only the logic required for wake-up is powered.

Warning

Exiting LPM is similar to a POR. The CAN FD Controller module will transition to Configuration mode. All registers will be reset, and RAM data will be lost. The device must be reconfigured.

Parameters

Input	*pComp	Is the pointed structure of the device where the sleep will be configured
Input	useLowPowerMode	Is at 'true' to use the low power mode if available or 'false' to use the simpler sleep
Input	wakeUpFilter	Indicate which filter to use for wake-up due to CAN bus activity. This feature can be used to protect the module from wake-up due to short glitches on the RXCAN pin
Input	interruptBusWakeUp	Is at 'true' to enable bus wake-up interrupt or 'false' to disable the interrupt

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL
- ERR__NOT_SUPPORTED when you try to configure a Deep Sleep Mode on a MCP2517FD device (not supported by this device)

```
eERRORRESULT MCP251XFD_EnterSleepMode(MCP251XFD *pComp)
```

This function puts the device in sleep mode.

Parameters

Input	*pComp	Is the pointed structure of the device to be used

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL
- ERR__CONFIGURATION when the Sleep Mode is not previously configured. See §10
- ERR__ALREADY_IN_SLEEP when the device is already in Sleep or Deep Sleep Mode

```
eERRORRESULT MCP251XFD_IsDeviceInSleepMode(
      MCP251XFD *pComp,
      bool* isInSleepMode)
```

Verify if the MCP251XFD device is in sleep mode. This function verifies if the device is in sleep mode by checking the OSC.OSCDIS.

Warning

In low power mode, it is impossible to check if the device is in sleep mode or not without wake it up because a simple asserting of the SPI CS will exit the LPM.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*isInSleepMode	Indicate if the device is in sleep mode

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL
- ERR__CONFIGURATION when the Sleep Mode is not previously configured. See §10
- ERR__NOT_SUPPORTED when the device is in Deep Sleep Mode. See warning. Here the function always returns isInSleepMode at 'true'

　　　　　　　Ref: MCP251XFD driver library guide
(Synchronous driver).docx

```
eERRORRESULT MCP251XFD_WakeUp(
    MCP251XFD *pComp,
    eMCP251XFD_PowerStates *fromState)
```

Manually wake up the MCP251XFD device. After a wake-up from sleep, the device will be in configuration mode. After a wake-up from low power sleep, the device is at the same state as a Power On Reset, the device has to be reconfigured.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*fromState	Is the power mode state before the wake up (Can be NULL if not necessary to know)

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL
- ERR__CONFIGURATION when the Sleep Mode is not previously configured. See §10

```
eMCP251XFD_PowerStates MCP251XFD_BusWakeUpFromState(MCP251XFD *pComp)
```

Retrieve from which state mode the MCP251XFD device get a bus wake up from. Use this function when the wake up interrupt occur (wake up from bus) and it's not from a manual wake up with the function *MCP251XFD_WakeUp()*. This function will indicate from which sleep mode (normal of low power) the wake up occurs.

Warning

If you call this function, the driver will understand that the device is awake without verifying and configure itself as well.

Parameters

Input	*pComp	Is the pointed structure of the device to be used

Return

Return the sleep mode state before waking up.

15.11.1. Enumerators

enum eMCP251XFD_PowerStates

Device power states.

Enumerator

MCP251XFD_DEVICE_SLEEP_NOT_CONFIGURED	**0x0**	Device sleep mode is not configured so the device is in normal power state
MCP251XFD_DEVICE_NORMAL_POWER_STATE	**0x1**	Device is in normal power state
MCP251XFD_DEVICE_SLEEP_STATE	**0x2**	Device is in sleep power state
MCP251XFD_DEVICE_LOWPOWER_SLEEP_STATE	**0x3**	Device is in low-power sleep power state

enum eMCP251XFD_WakeUpFilter

Wake-up Filter Time bits for the CiCON.WFT register. Pulse on RXCAN shorter than the minimum TFILTER time will be ignored; pulses longer than the maximum TFILTER time will wake-up the device.

Enumerator

MCP251XFD_T00FILTER_60ns	**0b000**	Time min = 40ns (MCP2517FD) / 50ns (MCP2518FD) max = 75ns (MCP2517FD) / 100ns (MCP2518FD)
MCP251XFD_T01FILTER_100ns	**0b001**	Time min = 70ns (MCP2517FD) / 80ns (MCP2518FD) max = 120ns (MCP2517FD) / 140ns (MCP2518FD)
MCP251XFD_T10FILTER_170ns	**0b010**	Time min = 125ns (MCP2517FD) / 130ns (MCP2518FD) max = 215ns (MCP2517FD) / 220ns (MCP2518FD)
MCP251XFD_T11FILTER_300ns	**0b011**	Time min = 225ns (MCP2517FD) / 225ns (MCP2518FD) max = 390ns (MCP2517FD) / 390ns (MCP2518FD)
MCP251XFD_TO_FILTER	**0b111**	Do not use a filter for wake-up

15.12. Time Stamp

```
eERRORRESULT MCP251XFD_ConfigureTimeStamp(
      MCP251XFD *pComp,
      bool enableTS,
      eMCP251XFD_SamplePoint samplePoint,
      uint16_t prescaler,
      bool interruptBaseCounter)
```

Configure the Time Stamp of frames in the MCP251XFD device. This function configures the 32-bit free-running counter of the Time Stamp.

Parameters

Input	*pComp	Is the pointed structure of the device where the time stamp will be configured
Input	enableTS	Is at 'true' to enable Time Stamp or 'false' to disable Time Stamp
Input	samplePoint	Is an enumerator that indicate where the Time Stamp sample point is at
Input	prescaler	Is the prescaler of the Time Stamp counter (time in µs is: 1/SYSCLK/TBCPRE)
Input	interruptBaseCounter	Is at 'true' to enable time base counter interrupt or 'false' to disable the interrupt. A rollover of the TBC will generate an interrupt if interruptBaseCounter is set

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL, or prescaler is out of range

```
eERRORRESULT MCP251XFD_SetTimeStamp(
      MCP251XFD *pComp,
      uint32_t value)
```

Set the Time Stamp counter the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	value	Is the value to set into the counter

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
eERRORRESULT MCP251XFD_GetTimeStamp(
      MCP251XFD *pComp,
      uint32_t value)
```

Get the Time Stamp counter the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*value	Is the value to get from the counter

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

15.12.1. Enumerators

enum eMCP251XFD_SamplePoint

TimeStamp sample point.

Enumerator

MCP251XFD_TS_CAN20_SOF	0b00	Time Stamp at "beginning" of Frame: CAN2.0 at sample point of SOF
MCP251XFD_TS_CAN20_SOF_CANFD_SOF	0b00	Time Stamp at "beginning" of Frame: CAN2.0 at sample point of SOF & CAN-FD at sample point of SOF
MCP251XFD_TS_CAN20_SOF_CANFD_FDF	0b10	Time Stamp at "beginning" of Frame: CAN2.0 at sample point of SOF & CAN-FD at sample point of the bit following the FDF bit
MCP251XFD_TS_CAN20_EOF	0b01	Time Stamp at "end-on-frame" of Frame: CAN2.0 at sample point of EOF
MCP251XFD_TS_CAN20_EOF_CANFD_EOF	0b01	Time Stamp at "end-on-frame" of Frame: CAN2.0 at sample point of EOF & CAN-FD at sample point of EOF

15.13.   FIFOs

```
eERRORRESULT MCP251XFD_ConfigureTEF(
      MCP251XFD *pComp,
      bool enableTEF,
      MCP251XFD_FIFO *confTEF)
```

Configure TEF of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	enableTEF	Indicate if the TEF must be activated or not
Input	*confTEF	Is the configuration structure of the TEF

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL, or confTEF is NULL with enableTEF at 'true', or confTEF->Name is not MCP251XFD_TEF
- ERR__NEED_CONFIG_MODE when the device is not in Configuration Mode

```
eERRORRESULT MCP251XFD_ConfigureTXQ(
      MCP251XFD *pComp,
      bool enableTXQ,
      MCP251XFD_FIFO *confTXQ)
```

Configure TXQ of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	enableTXQ	Indicate if the TXQ must be activated or not
Input	*confTXQ	Is the configuration structure of the TXQ

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL, or confTXQ is NULL with enableTXQ at 'true', or confTXQ->Name is not MCP251XFD_TXQ
- ERR__NEED_CONFIG_MODE when the device is not in Configuration Mode
- ERR__OUT_OF_MEMORY when the amount of RAM memory taken by the TXQ configuration is too high

```
eERRORRESULT MCP251XFD_ConfigureFIFO(
      MCP251XFD *pComp,
      MCP251XFD_FIFO *confFIFO)
```

Configure a FIFO of the MCP251XFD device. FIFO are enabled by configuring the FIFO and, in case of receive FIFO, a filter point to the FIFO.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	*confFIFO	Is the configuration structure of the FIFO

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL, or confFIFO->Name is MCP251XFD_TEF or MCP251XFD_TXQ, or confFIFO->Name is superior at MCP251XFD_FIFO31
- ERR__NEED_CONFIG_MODE when the device is not in Configuration Mode
- ERR__OUT_OF_MEMORY when the amount of RAM memory taken by the FIFO configuration is too high

```
eERRORRESULT MCP251XFD_ConfigureFIFOList(
        MCP251XFD *pComp,
        MCP251XFD_FIFO *listFIFO,
        size_t count)
```

Configure a FIFO list of the MCP251XFD device. This function configures a set of FIFO at once.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	*listFIFO	Is the list of FIFO to configure
Input	count	Is the count of FIFO in the list

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL, or listFIFO is NULL
- ERR__OUT_OF_RANGE when there are more than 33 entries (31 FIFO + TEF + TXQ)
- ERR__TOO_MANY_TEF when there are more than one TEF in the list
- ERR__TOO_MANY_TXQ when there are more than one TXQ in the list
- ERR__OUT_OF_MEMORY when the amount of RAM memory taken by the list of FIFO configuration is too high
- See MCP251XFD_ConfigureTEF()'s returns value for more information.
- See MCP251XFD_ConfigureTXQ()'s returns value for more information.
- See MCP251XFD_ConfigureFIFO()'s returns value for more information.

```
eERRORRESULT MCP251XFD_ResetFIFO(
        MCP251XFD *pComp,
        eMCP251XFD_FIFO name)
```

Reset a FIFO of the MCP251XFD device. The function will wait until the reset is effective. In Configuration Mode, the FIFO is automatically reset.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	name	Is the name of the FIFO to be reset

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL, or the name is superior to MCP251XFD_FIFO31
- ERR__DEVICE_TIMEOUT when the device takes too much time to respond to the reset

```
inline eERRORRESULT MCP251XFD_ResetTEF(MCP251XFD *pComp)
```

Reset the TEF of the MCP251XFD device. The function will wait until the reset is effective. In Configuration Mode, the TEF is automatically reset.

Parameters

Input	*pComp	Is the pointed structure of the device to be used

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_ResetFIFO()'s returns value for more information.

```
inline eERRORRESULT MCP251XFD_ResetTXQ(MCP251XFD *pComp)
```

Reset the TXQ of the MCP251XFD device. The function will wait until the reset is effective. In Configuration Mode, the TXQ is automatically reset.

Parameters

Input	*pComp	Is the pointed structure of the device to be used

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_ResetFIFO()'s returns value for more information.

Ref: MCP251XFD driver library guide
                                                            (Synchronous driver).docx

```
eERRORRESULT MCP251XFD_UpdateFIFO(
        MCP251XFD *pComp,
        eMCP251XFD_FIFO name,
        bool andFlush)
```
Update (and flush) a FIFO of the MCP251XFD device. Increment the Head/Tail of the FIFO. If flush too, a message send request is ask.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	name	Is the name of the FIFO to be update (and flush)
Input	andFlush	Indicate if the FIFO must be flush too

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL, or the name is superior to MCP251XFD_FIFO31

```
inline eERRORRESULT MCP251XFD_UpdateTEF(MCP251XFD *pComp)
```
Update the TEF of the MCP251XFD device. Increment the Head/Tail of the TEF.

Parameters

Input	*pComp	Is the pointed structure of the device to be used

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_UpdateFIFO()'s returns value for more information.

```
inline eERRORRESULT MCP251XFD_UpdateTXQ(
        MCP251XFD *pComp,
        bool andFlush)
```
Update (and flush) the TXQ of the MCP251XFD device. Increment the Head/Tail of the TXQ. If flush too, a message send request is ask.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	andFlush	Indicate if the TXQ must be flush too

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_UpdateFIFO()'s returns value for more information.

```
eERRORRESULT MCP251XFD_FlushFIFO(
        MCP251XFD *pComp,
        eMCP251XFD_FIFO name)
```
Flush a FIFO of the MCP251XFD device. A message send request is ask to the FIFO.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	name	Is the name of the FIFO to be flush

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL, or the name is superior to MCP251XFD_FIFO31
- ERR__NOT_AVAILABLE when the name is MCP251XFD_TEF

```
inline eERRORRESULT MCP251XFD_FlushTXQ(MCP251XFD *pComp)
```
Flush a TXQ of the MCP251XFD device. A message send request is ask to the TXQ.

Parameters

Input   *pComp   Is the pointed structure of the device to be used

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_FlushFIFO()'s returns value for more information.

```
inline eERRORRESULT MCP251XFD_FlushAllFIFO(MCP251XFD *pComp)
```
Flush all FIFOs (+TXQ) of the MCP251XFD device. Flush all TXQ and all transmit FIFOs.

Parameters

Input   *pComp   Is the pointed structure of the device to be used

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
eERRORRESULT MCP251XFD_GetFIFOStatus(
       MCP251XFD *pComp,
       eMCP251XFD_FIFO name,
       eMCP251XFD_FIFOstatus *statusFlags)
```
Get status of a FIFO of the MCP251XFD device. Get messages status and some interrupt flags related to this FIFO (First byte of CiFIFOSTAm).

Parameters

Input   *pComp   Is the pointed structure of the device to be used
Input   name   Is the name of the FIFO where status flags will be got
Output   *statusFlags Is the return value of status flags

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL, or statusFlags is NULL, or the name is superior to MCP251XFD_FIFO31

```
inline eERRORRESULT MCP251XFD_GetTEFStatus(
       MCP251XFD *pComp,
       eMCP251XFD_TEFstatus *statusFlags)
```
Get status of a TEF of the MCP251XFD device. Get messages status and some interrupt flags related to this TEF (First byte of CiTEFSTA).

Parameters

Input   *pComp   Is the pointed structure of the device to be used
Output   *statusFlags Is the return value of status flags

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_GetFIFOStatus()'s returns value for more information.

```
inline eERRORRESULT MCP251XFD_GetTXQStatus(
       MCP251XFD *pComp,
       eMCP251XFD_TXQstatus *statusFlags)
```
Get status of a TXQ of the MCP251XFD device. Get messages status and some interrupt flags related to this TXQ (First byte of CiTXQSTA).

Parameters

Input   *pComp   Is the pointed structure of the device to be used
Output   *statusFlags Is the return value of status flags

Return

Returns an *eERRORRESULT* value enumerator. See MCP251XFD_GetFIFOStatus()'s returns value for more information.

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

```
eERRORRESULT MCP251XFD_GetNextMessageAddressFIFO(
    MCP251XFD *pComp,
    eMCP251XFD_FIFO name,
    uint32_t *nextAddress,
    uint8_t *nextIndex)
```

Get next message address and/or index of a FIFO of the MCP251XFD device. If it's a transmit FIFO then a read of this will return the address and/or index where the next message is to be written (FIFO head). If it's a receive FIFO then a read of this will return the address and/or index where the next message is to be read (FIFO tail)

Warning

> This register is not guaranteed to read correctly in Configuration mode and should only be accessed when the module is not in Configuration mode

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	name	Is the name of the FIFO
Output	*nextAddress	Is the next user address of the FIFO. This parameter can be NULL if not needed
Output	*nextIndex	Is the next user index of the FIFO. This parameter can be NULL if not needed

Return

> Returns an *eERRORRESULT* value enumerator.
> - ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL, or the name is superior to MCP251XFD_FIFO31
> - ERR__NOT_AVAILABLE when nextIndex is not NULL and name is MCP251XFD_TEF

```
inline eERRORRESULT MCP251XFD_GetNextMessageAddressTEF(
    MCP251XFD *pComp,
    uint32_t *nextAddress)
```

A read of this register will return the address where the next object is to be read (FIFO tail). This register is not guaranteed to read correctly in Configuration mode and should only be accessed when the module is not in Configuration mode.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*nextAddress	Is the next user address of the TEF. This parameter can be NULL if not needed

Return

> Returns an *eERRORRESULT* value enumerator. See MCP251XFD_GetNextMessageAddressFIFO()'s returns value for more information.

```
inline eERRORRESULT MCP251XFD_GetNextMessageAddressTXQ(
    MCP251XFD *pComp,
    uint32_t *nextAddress,
    uint8_t *nextIndex)
```

A read of this register will return the address where the next message is to be written (TXQ head). This register is not guaranteed to read correctly in Configuration mode and should only be accessed when the module is not in Configuration mode.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*nextAddress	Is the next user address of the TXQ. This parameter can be NULL if not needed
Output	*nextIndex	Is the next user index of the TXQ. This parameter can be NULL if not needed

Return

> Returns an *eERRORRESULT* value enumerator. See MCP251XFD_GetNextMessageAddressFIFO()'s returns value for more information.

```
eERRORRESULT MCP251XFD_ClearFIFOConfiguration(
    MCP251XFD *pComp,
    eMCP251XFD_FIFO name)
```

Clear the FIFO configuration of the MCP251XFD device. Clearing FIFO configuration do not totally disable it, all filter that point to it must be disabled too

Warning

Never clear a FIFO in the middle of the FIFO list or it will destroy messages in RAM after this FIFO

Parameters

Input   *pComp   Is the pointed structure of the device to be used

Input   name   Is the name of the FIFO where the configuration will be cleared

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

15.13.1. Enumerators

enum eMCP251XFD_FIFOstatus

Transmit and Receive FIFO status.

Enumerator

MCP251XFD_TX_FIFO_FULL	**0x00**	Transmit FIFO full
MCP251XFD_TX_FIFO_NOT_FULL	**0x01**	Transmit FIFO not full
MCP251XFD_TX_FIFO_HALF_EMPTY	**0x02**	Transmit FIFO half empty
MCP251XFD_TX_FIFO_EMPTY	**0x04**	Transmit FIFO empty
MCP251XFD_TX_FIFO_ATTEMPTS_EXHAUSTED	**0x10**	Transmit FIFO attempts exhausted
MCP251XFD_TX_FIFO_BUS_ERROR	**0x20**	Transmit bus error
MCP251XFD_TX_FIFO_ARBITRATION_LOST	**0x40**	Transmit arbitration lost
MCP251XFD_TX_FIFO_ABORTED	**0x80**	Transmit aborted
MCP251XFD_RX_FIFO_EMPTY	**0x00**	Receive FIFO empty
MCP251XFD_RX_FIFO_NOT_EMPTY	**0x01**	Receive FIFO not empty
MCP251XFD_RX_FIFO_HALF_FULL	**0x02**	Receive FIFO half full
MCP251XFD_RX_FIFO_FULL	**0x04**	Receive FIFO full
MCP251XFD_RX_FIFO_OVERFLOW	**0x08**	Receive overflow

enum eMCP251XFD_TEFstatus

Transmit Event FIFO status.

Enumerator

MCP251XFD_TEF_FIFO_EMPTY	**0x00**	TEF FIFO empty
MCP251XFD_TEF_FIFO_NOT_EMPTY	**0x01**	TEF FIFO not empty
MCP251XFD_TEF_FIFO_HALF_FULL	**0x02**	TEF FIFO half full
MCP251XFD_TEF_FIFO_FULL	**0x04**	TEF FIFO full
MCP251XFD_TEF_FIFO_OVERFLOW	**0x08**	TEF overflow

enum eMCP251XFD_TXQstatus

Transmit Queue status.

Enumerator

MCP251XFD_TXQ_FIFO_FULL	**0x00**	TXQ full
MCP251XFD_TXQ_FIFO_NOT_FULL	**0x01**	TXQ not full
MCP251XFD_TXQ_FIFO_EMPTY	**0x04**	TXQ empty
MCP251XFD_TXQ_FIFO_ATTEMPTS_EXHAUSTED	**0x10**	TXQ attempts exhausted
MCP251XFD_TXQ_FIFO_BUS_ERROR	**0x20**	TXQ bus error
MCP251XFD_TXQ_FIFO_ARBITRATION_LOST	**0x40**	TXQ arbitration lost
MCP251XFD_TXQ_FIFO_ABORTED	**0x80**	TXQ aborted

15.14.   Filters

```
eERRORRESULT MCP251XFD_ConfigureDeviceNetFilter(
      MCP251XFD *pComp,
      eMCP251XFD_DNETFilter filter)
```
Configure the Device NET filter of the MCP251XFD device. When a standard frame is received and the filter is configured for extended frames, the EID part of the Filter and Mask Object can be selected to filter on data bytes

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	filter	Is the Device NET Filter to apply on all received frames and filters

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
eERRORRESULT MCP251XFD_ConfigureFilter(
      MCP251XFD *pComp,
      MCP251XFD_Filter *confFilter)
```
Configure a filter of the MCP251XFD device.

Warning

This function does not check if the pointed FIFO is a receive FIFO.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	*confFilter	Is the configuration structure of the Filter

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL, or confFilter is NULL
- ERR__CONFIGURATION when confFilter->PointTo is MCP251XFD_TEF, MCP251XFD_TXQ, or is superior to MCP251XFD_FIFO31
- ERR__FILTER_CONSISTENCY when AcceptanceID and AcceptanceMask does not correspond
- ERR__FILTER_TOO_LARGE when AcceptanceID and AcceptanceMask are too large for the specified configuration

```
eERRORRESULT MCP251XFD_ConfigureFilterList(
      MCP251XFD *pComp,
      eMCP251XFD_DNETFilter filter,
      MCP251XFD_Filter *listFilter,
      size_t count)
```
Configure a filter list and the DNCNT of the MCP251XFD device. This function configures a set of Filters at once.

Warning

This function does not check if the pointed FIFO is a receive FIFO.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	filter	Is the Device NET Filter to apply on all received frames, it call directly *MCP251XFD_ConfigureDeviceNetFilter()*
Input	*listFilter	Is the list of Filters to configure
Input	count	Is the count of Filters in the list

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL, or listFilter is NULL
- See MCP251XFD_ConfigureFilter()'s returns value for more information.

Ref: MCP251XFD driver library guide (Synchronous driver).docx

```
eERRORRESULT MCP251XFD_DisableFilter(
      MCP251XFD *pComp,
      eMCP251XFD_Filter name)
```
Disable a Filter of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	name	Is the name of the Filter to disable

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL, or the name is superior to MCP251XFD_FILTER31

15.14.1. Enumerators

enum **eMCP251XFD_DNETFilter**

Device Net Filter Bit Number bits for the CiCON.DNCNT register.

If DNCNT is greater than 0 and the received message has DLC = 0, indicating no data payload, the filter comparison will terminate with the identifier.

If DNCNT is greater than 8 and the received message has DLC = 1, indicating a payload of one data byte, the filter comparison will terminate with the 8th bit of the data.

If DNCNT is greater than 16 and the received message has DLC = 2, indicating a payload of two data bytes, the filter comparison will terminate with the 16th bit of the data.

If DNCNT is greater than 18, indicating that the user selected a number of bits greater than the total number of EID bits, the filter comparison will terminate with the 18th bit of the data.

Enumerator

MCP251XFD_D_NET_FILTER_DISABLE	0b00000	Do not compare data bytes
MCP251XFD_D_NET_FILTER_1Bit	0b00001	Compare up to data byte 0 bit 7 with EID0 (Data Byte 0[7] = EID[0])
MCP251XFD_D_NET_FILTER_2Bits	0b00010	Compare up to data byte 0 bit 6 with EID1 (Data Byte 0[7:6] = EID[0:1])
MCP251XFD_D_NET_FILTER_3Bits	0b00011	Compare up to data byte 0 bit 5 with EID2 (Data Byte 0[7:5] = EID[0:2])
MCP251XFD_D_NET_FILTER_4Bits	0b00100	Compare up to data byte 0 bit 4 with EID3 (Data Byte 0[7:4] = EID[0:3])
MCP251XFD_D_NET_FILTER_5Bits	0b00101	Compare up to data byte 0 bit 3 with EID4 (Data Byte 0[7:3] = EID[0:4])
MCP251XFD_D_NET_FILTER_6Bits	0b00110	Compare up to data byte 0 bit 2 with EID5 (Data Byte 0[7:2] = EID[0:5])
MCP251XFD_D_NET_FILTER_7Bits	0b00111	Compare up to data byte 0 bit 1 with EID6 (Data Byte 0[7:1] = EID[0:6])
MCP251XFD_D_NET_FILTER_8Bits	0b01000	Compare up to data byte 0 bit 0 with EID7 (Data Byte 0[7:0] = EID[0:7])
MCP251XFD_D_NET_FILTER_9Bits	0b01001	Compare up to data byte 1 bit 7 with EID8 (Data Byte 0[7:0] and Data Byte 1[7] = EID[0:8])
MCP251XFD_D_NET_FILTER_10Bits	0b01010	Compare up to data byte 1 bit 6 with EID9 (Data Byte 0[7:0] and Data Byte 1[7:6] = EID[0:9])
MCP251XFD_D_NET_FILTER_11Bits	0b01011	Compare up to data byte 1 bit 5 with EID10 (Data Byte 0[7:0] and Data Byte 1[7:5] = EID[0:10])
MCP251XFD_D_NET_FILTER_12Bits	0b01100	Compare up to data byte 1 bit 4 with EID11 (Data Byte 0[7:0] and Data Byte 1[7:4] = EID[0:11])
MCP251XFD_D_NET_FILTER_13Bits	0b01101	Compare up to data byte 1 bit 3 with EID12 (Data Byte 0[7:0] and Data Byte 1[7:3] = EID[0:12])
MCP251XFD_D_NET_FILTER_14Bits	0b01110	Compare up to data byte 1 bit 2 with EID13 (Data Byte 0[7:0] and Data Byte 1[7:2] = EID[0:13])
MCP251XFD_D_NET_FILTER_15Bits	0b01111	Compare up to data byte 1 bit 1 with EID14 (Data Byte 0[7:0] and Data Byte 1[7:1] = EID[0:14])

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

MCP251XFD_D_NET_FILTER_16Bits	**0b10000**	Compare up to data byte 1 bit 0 with EID15 (Data Byte 0[7:0] and Data Byte 1[7:0] = EID[0:15])
MCP251XFD_D_NET_FILTER_17Bits	**0b10001**	Compare up to data byte 2 bit 7 with EID16 (Data Byte 0[7:0] and Data Byte 1[7:0] and Byte 2[7] = EID[0:16])
MCP251XFD_D_NET_FILTER_18Bits	**0b10010**	Compare up to data byte 2 bit 6 with EID17 (Data Byte 0[7:0] and Data Byte 1[7:0] and Byte 2[7:6] = EID[0:17])

15.15. Interrupts

```
eERRORRESULT MCP251XFD_ConfigureInterrupt(
      MCP251XFD *pComp,
      setMCP251XFD_InterruptEvents interruptsFlags)
```
Configure interrupt of the MCP251XFD device.

Parameters

Input   *pComp   Is the pointed structure of the device to be used

Input   interruptsFlags   Is the set of events where interrupts will be enabled. Flags can be OR'ed

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
eERRORRESULT MCP251XFD_GetInterruptEvents(
      MCP251XFD *pComp,
      setMCP251XFD_InterruptEvents* interruptsFlags)
```
Get interrupt events of the MCP251XFD device.

Parameters

Input   *pComp   Is the pointed structure of the device to be used

Input   *interruptsFlags   Is the return value of interrupt events. Flags are OR'ed

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
eERRORRESULT MCP251XFD_GetCurrentInterruptEvent(
      MCP251XFD *pComp,
      eMCP251XFD_InterruptFlagCode* currentEvent)
```
Get the current interrupt event of the MCP251XFD device.

Parameters

Input   *pComp   Is the pointed structure of the device to be used

Input   *currentEvent   Is the return value of the current interrupt event

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
eERRORRESULT MCP251XFD_ClearInterruptEvents(
      MCP251XFD *pComp,
      setMCP251XFD_InterruptEvents interruptsFlags)
```
Get the current interrupt event of the MCP251XFD device.

Parameters

Input   *pComp   Is the pointed structure of the device to be used

Input   interruptsFlags   Is the set of events where interrupts will be cleared. Flags can be OR'ed

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

       Ref: MCP251XFD driver library guide
(Synchronous driver).docx

```
eERRORRESULT MCP251XFD_GetCurrentReceiveFIFONameAndStatusInterrupt(
      MCP251XFD *pComp,
      eMCP251XFD_FIFO *name,
      eMCP251XFD_FIFOstatus *flags)
```

This function can be called to check if there is a receive interrupt. It gives the name and the status of the FIFO. If more than one object has an interrupt pending, the interrupt or FIFO with the highest number will show up. Once the interrupt with the highest priority is cleared, the next highest priority interrupt will show up.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*name	Is the returned name of the FIFO that generate an interrupt
Output	*flags	Is the return value of status flags of the FIFO (can be NULL if it is not needed)

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL, or name is NULL

```
inline eERRORRESULT MCP251XFD_GetCurrentReceiveFIFONameInterrupt(
      MCP251XFD *pComp,
      eMCP251XFD_FIFO *name)
```

Get current receive FIFO name that generate an interrupt (if any). This function can be called to check if there is a receive interrupt. It gives the name of the FIFO. If more than one object has an interrupt pending, the interrupt or FIFO with the highest number will show up. Once the interrupt with the highest priority is cleared, the next highest priority interrupt will show up.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*name	Is the returned name of the FIFO that generate an interrupt

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
eERRORRESULT MCP251XFD_GetCurrentTransmitFIFONameAndStatusInterrupt(
      MCP251XFD *pComp,
      eMCP251XFD_FIFO *name,
      eMCP251XFD_FIFOstatus *flags)
```

Get current transmit FIFO name and status that generate an interrupt (if any). This function can be called to check if there is a transmit interrupt. It gives the name and the status of the FIFO. If more than one object has an interrupt pending, the interrupt or FIFO with the highest number will show up. Once the interrupt with the highest priority is cleared, the next highest priority interrupt will show up.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*name	Is the returned name of the FIFO that generate an interrupt
Output	*flags	Is the return value of status flags of the FIFO (can be NULL if it is not needed)

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
inline eERRORRESULT MCP251XFD_GetCurrentTransmitFIFONameInterrupt(
      MCP251XFD *pComp,
      eMCP251XFD_FIFO *name)
```

Get current transmit FIFO name that generate an interrupt (if any). This function can be called to check if there is a transmit interrupt. It gives the name of the FIFO. If more than one object has an interrupt pending, the interrupt or FIFO with the highest number will show up. Once the interrupt with the highest priority is cleared, the next highest priority interrupt will show up.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*name	Is the returned name of the FIFO that generate an interrupt

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
eERRORRESULT MCP251XFD_ClearFIFOEvents(
      MCP251XFD *pComp,
      eMCP251XFD_FIFO name,
      uint8_t events)
```
Clear selected FIFO events of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	name	Is the name of the FIFO where events will be cleared
Input	events	Are the set of events to clear. Can be ether an *eMCP251XFD_FIFOstatus*, an *eMCP251XFD_TEFstatus* or an *eMCP251XFD_TXQstatus* type.

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL, or the name is superior to MCP251XFD_FIFO31

```
inline eERRORRESULT MCP251XFD_ClearTEFOverflowEvent(MCP251XFD *pComp)
```
Clear the overflow event of the TEF.

Parameters

Input	*pComp	Is the pointed structure of the device to be used

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
inline eERRORRESULT MCP251XFD_ClearFIFOOverflowEvent(
      MCP251XFD *pComp,
      eMCP251XFD_FIFO name)
```
Clear the overflow event of the FIFO selected.

Warning

This function does not check if it's a receive FIFO

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	name	Is the name of the FIFO where event will be cleared

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
inline eERRORRESULT MCP251XFD_ClearFIFOAttemptsEvent(
      MCP251XFD *pComp,
      eMCP251XFD_FIFO name)
```
Clear the attempt event of the FIFO selected.

Warning

This function does not check if it is a transmit FIFO

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	name	Is the name of the FIFO where event will be cleared

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

*inline eERRORRESULT **MCP251XFD_ClearTXQAttemptsEvent**(MCP251XFD `*pComp`)*

Clear the attempt event of the TXQ selected.

Warning

This function does not check if it is a transmit FIFO

Parameters

Input   *pComp   Is the pointed structure of the device to be used

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

*eERRORRESULT **MCP251XFD_GetReceiveInterruptStatusOfAllFIFO**(*
    MCP251XFD `*pComp`,
    setMCP251XFD_InterruptOnFIFO `*interruptPending`,*
    setMCP251XFD_InterruptOnFIFO `*overflowStatus`)*

Get the status of receive pending interrupt and overflow pending interrupt of all FIFOs at once, OR'ed in one variable.

Parameters

Input   *pComp   Is the pointed structure of the device to be used

Output   *interruptPending   Is the return of the receive pending interrupt of all FIFOs (This parameter can be NULL)

Output   *overflowStatus   Is the return of the receive overflow pending interrupt of all FIFOs (This parameter can be NULL)

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

*inline eERRORRESULT **MCP251XFD_GetReceivePendingInterruptStatusOfAllFIFO**(*
    MCP251XFD `*pComp`,
    setMCP251XFD_InterruptOnFIFO `*interruptPending`)*

Get the status of receive overflow interrupt of all FIFOs at once, OR'ed in one variable.

Warning

This function does not check if it is a transmit FIFO

Parameters

Input   *pComp   Is the pointed structure of the device to be used

Output   *interruptPending   Is the return of the receive pending interrupt of all FIFOs

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

*inline eERRORRESULT **MCP251XFD_GetReceiveOverflowInterruptStatusOfAllFIFO**(*
    MCP251XFD `*pComp`,
    setMCP251XFD_InterruptOnFIFO `*overflowStatus`)*

Get the status of receive overflow interrupt of all FIFOs at once, OR'ed in one variable.

Warning

This function does not check if it is a transmit FIFO

Parameters

Input   *pComp   Is the pointed structure of the device to be used

Output   *overflowStatus   Is the return of the receive overflow pending interrupt of all FIFOs

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
eERRORRESULT MCP251XFD_GetTransmitInterruptStatusOfAllFIFO(
    MCP251XFD *pComp,
    setMCP251XFD_InterruptOnFIFO* interruptPending,
    setMCP251XFD_InterruptOnFIFO* attemptStatus)
```

Get the status of transmit pending interrupt and attempt exhaust pending interrupt of all FIFOs at once, OR'ed in one variable.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*interruptPending	Is the return of the transmit pending interrupt of all FIFOs (This parameter can be NULL)
Output	*attemptStatus	Is the return of the transmit attempt exhaust pending interrupt of all FIFOs (This parameter can be NULL)

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
inline eERRORRESULT MCP251XFD_GetTransmitPendingInterruptStatusOfAllFIFO(
    MCP251XFD *pComp,
    setMCP251XFD_InterruptOnFIFO* interruptPending)
```

Get the status of transmit pending interrupt of all FIFOs at once, OR'ed in one variable.

Warning

This function does not check if it is a transmit FIFO

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*interruptPending	Is the return of the transmit pending interrupt of all FIFOs

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
inline eERRORRESULT MCP251XFD_GetTransmitAttemptInterruptStatusOfAllFIFO(
    MCP251XFD *pComp,
    setMCP251XFD_InterruptOnFIFO* attemptStatus)
```

Get the status of attempt exhaust pending interrupt of all FIFOs at once, OR'ed in one variable.

Warning

This function does not check if it is a transmit FIFO

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*attemptStatus	Is the return of the transmit attempt exhaust pending interrupt of all FIFOs

Return

Returns an *eERRORRESULT* value enumerator.

- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

15.15.1. Enumerators

enum eMCP251XFD_InterruptFlagCode

Interrupt Flag Code bits for the CiVEC.ICODE.

Enumerator

MCP251XFD_TXQ_INTERRUPT	0b0000000	TXQ Interrupt (TFIF<0> set). If for RXCODE, FIFO 0 can't receive so this flag code is reserved
MCP251XFD_FIFO1_INTERRUPT	0b0000001	FIFO 1 Interrupt (TFIF<1> or RFIF<1> set)
…	…	…
MCP251XFD_FIFO31_INTERRUPT	0b0011111	FIFO 31 Interrupt (TFIF<31> or RFIF<31> set)
MCP251XFD_NO_INTERRUPT	0b1000000	No interrupt
MCP251XFD_ERROR_INTERRUPT	0b1000001	Error Interrupt (CERRIF/IE)
MCP251XFD_WAKEUP_INTERRUPT	0b1000010	Wake-up interrupt (WAKIF/WAKIE)
MCP251XFD_RECEIVE_FIFO_OVF	0b1000011	Receive FIFO Overflow Interrupt (any bit in CiRXOVIF set)
MCP251XFD_ADDRESS_ERROR_INTERRUPT	0b1000100	Address Error Interrupt (illegal FIFO address presented to system) (SERRIF/IE)
MCP251XFD_RXTX_MAB_OVF_UVF	0b1000101	RX/TX MAB Overflow/Underflow (RX: message received before previous message was saved to memory; TX: cannot feed TX MAB fast enough to transmit consistent data) (SERRIF/IE)
MCP251XFD_TBC_OVF_INTERRUPT	0b1000110	TBC Overflow (TBCIF/IE)
MCP251XFD_OPMODE_CHANGE_OCCURED	0b1000111	Operation Mode Change Occurred (MODIF/IE)
MCP251XFD_INVALID_MESSAGE_OCCURED	0b1001000	Invalid Message Occurred (IVMIF/IE)
MCP251XFD_TRANSMIT_EVENT_FIFO	0b1001001	Transmit Event FIFO Interrupt (any bit in CiTEFIF set)
MCP251XFD_TRANSMIT_ATTEMPT	0b1001010	Transmit Attempt Interrupt (any bit in CiTXATIF set)

enum eMCP251XFD_InterruptOnFIFO
typedef eMCP251XFD_InterruptOnFIFO setMCP251XFD_InterruptOnFIFO

Receive Interrupt Status for the CiRXIF, CiRXOVIF, CiTXIF and CiTXATIF registers. Can be OR'ed.

Enumerator

MCP251XFD_INTERRUPT_ON_TXQ	0x00000001	Interrupt is pending on TXQ (TFIF<0> or TFATIF<0> set)
MCP251XFD_INTERRUPT_ON_FIFO1	0x00000002	Interrupt is pending on FIFO 1 (TFIF<1> or TFATIF<1> or RFIF<1> or RFOVIF<1> set)
MCP251XFD_INTERRUPT_ON_FIFO2	0x00000004	Interrupt is pending on FIFO 2 (TFIF<2> or TFATIF<2> or RFIF<2> or RFOVIF<2> set)
MCP251XFD_INTERRUPT_ON_FIFO3	0x00000008	Interrupt is pending on FIFO 3 (TFIF<3> or TFATIF<3> or RFIF<3> or RFOVIF<3> set)
MCP251XFD_INTERRUPT_ON_FIFO4	0x00000010	Interrupt is pending on FIFO 4 (TFIF<4> or TFATIF<4> or RFIF<4> or RFOVIF<4> set)
MCP251XFD_INTERRUPT_ON_FIFO5	0x00000020	Interrupt is pending on FIFO 5 (TFIF<5> or TFATIF<5> or RFIF<5> or RFOVIF<5> set)
MCP251XFD_INTERRUPT_ON_FIFO6	0x00000040	Interrupt is pending on FIFO 6 (TFIF<6> or TFATIF<6> or RFIF<6> or RFOVIF<6> set)
MCP251XFD_INTERRUPT_ON_FIFO7	0x00000080	Interrupt is pending on FIFO 7 (TFIF<7> or TFATIF<7> or RFIF<7> or RFOVIF<7> set)
MCP251XFD_INTERRUPT_ON_FIFO8	0x00000100	Interrupt is pending on FIFO 8 (TFIF<8> or TFATIF<8> or RFIF<8> or RFOVIF<8> set)
MCP251XFD_INTERRUPT_ON_FIFO9	0x00000200	Interrupt is pending on FIFO 9 (TFIF<9> or TFATIF<9> or RFIF<9> or RFOVIF<9> set)
MCP251XFD_INTERRUPT_ON_FIFO10	0x00000400	Interrupt is pending on FIFO 10 (TFIF<10> or TFATIF<10> or RFIF<10> or RFOVIF<10> set)
MCP251XFD_INTERRUPT_ON_FIFO11	0x00000800	Interrupt is pending on FIFO 11 (TFIF<11> or TFATIF<11> or RFIF<11> or RFOVIF<11> set)
MCP251XFD_INTERRUPT_ON_FIFO12	0x00001000	Interrupt is pending on FIFO 12 (TFIF<12> or TFATIF<12> or RFIF<12> or RFOVIF<12> set)

MCP251XFD_INTERRUPT_ON_FIFO13	**0x00002000**	Interrupt is pending on FIFO 13 (TFIF<13> or TFATIF<13> or RFIF<13> or RFOVIF<13> set)
MCP251XFD_INTERRUPT_ON_FIFO14	**0x00004000**	Interrupt is pending on FIFO 14 (TFIF<14> or TFATIF<14> or RFIF<14> or RFOVIF<14> set)
MCP251XFD_INTERRUPT_ON_FIFO15	**0x00008000**	Interrupt is pending on FIFO 15 (TFIF<15> or TFATIF<15> or RFIF<15> or RFOVIF<15> set)
MCP251XFD_INTERRUPT_ON_FIFO16	**0x00010000**	Interrupt is pending on FIFO 16 (TFIF<16> or TFATIF<16> or RFIF<16> or RFOVIF<16> set)
MCP251XFD_INTERRUPT_ON_FIFO17	**0x00020000**	Interrupt is pending on FIFO 17 (TFIF<17> or TFATIF<17> or RFIF<17> or RFOVIF<17> set)
MCP251XFD_INTERRUPT_ON_FIFO18	**0x00040000**	Interrupt is pending on FIFO 18 (TFIF<18> or TFATIF<18> or RFIF<18> or RFOVIF<18> set)
MCP251XFD_INTERRUPT_ON_FIFO19	**0x00080000**	Interrupt is pending on FIFO 19 (TFIF<19> or TFATIF<19> or RFIF<19> or RFOVIF<19> set)
MCP251XFD_INTERRUPT_ON_FIFO20	**0x00100000**	Interrupt is pending on FIFO 20 (TFIF<20> or TFATIF<20> or RFIF<20> or RFOVIF<20> set)
MCP251XFD_INTERRUPT_ON_FIFO21	**0x00200000**	Interrupt is pending on FIFO 21 (TFIF<21> or TFATIF<21> or RFIF<21> or RFOVIF<21> set)
MCP251XFD_INTERRUPT_ON_FIFO22	**0x00400000**	Interrupt is pending on FIFO 22 (TFIF<22> or TFATIF<22> or RFIF<22> or RFOVIF<22> set)
MCP251XFD_INTERRUPT_ON_FIFO23	**0x00800000**	Interrupt is pending on FIFO 23 (TFIF<23> or TFATIF<23> or RFIF<23> or RFOVIF<23> set)
MCP251XFD_INTERRUPT_ON_FIFO24	**0x01000000**	Interrupt is pending on FIFO 24 (TFIF<24> or TFATIF<24> or RFIF<24> or RFOVIF<24> set)
MCP251XFD_INTERRUPT_ON_FIFO25	**0x02000000**	Interrupt is pending on FIFO 25 (TFIF<25> or TFATIF<25> or RFIF<25> or RFOVIF<25> set)
MCP251XFD_INTERRUPT_ON_FIFO26	**0x04000000**	Interrupt is pending on FIFO 26 (TFIF<26> or TFATIF<26> or RFIF<26> or RFOVIF<26> set)
MCP251XFD_INTERRUPT_ON_FIFO27	**0x08000000**	Interrupt is pending on FIFO 27 (TFIF<27> or TFATIF<27> or RFIF<27> or RFOVIF<27> set)
MCP251XFD_INTERRUPT_ON_FIFO28	**0x10000000**	Interrupt is pending on FIFO 28 (TFIF<28> or TFATIF<28> or RFIF<28> or RFOVIF<28> set)
MCP251XFD_INTERRUPT_ON_FIFO29	**0x20000000**	Interrupt is pending on FIFO 29 (TFIF<29> or TFATIF<29> or RFIF<29> or RFOVIF<29> set)
MCP251XFD_INTERRUPT_ON_FIFO30	**0x40000000**	Interrupt is pending on FIFO 30 (TFIF<30> or TFATIF<30> or RFIF<30> or RFOVIF<30> set)
MCP251XFD_INTERRUPT_ON_FIFO31	**0x80000000**	Interrupt is pending on FIFO 31 (TFIF<31> or TFATIF<31> or RFIF<31> or RFOVIF<31> set)

15.16. Error management

```
eERRORRESULT MCP251XFD_GetTransmitReceiveErrorCountAndStatus(
    MCP251XFD *pComp,
    uint8_t* transmitErrorCount,
    uint8_t* receiveErrorCount,
    eMCP251XFD_TXRXErrorStatus* status)
```

Get transmit/receive error count and status of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*transmitErrorCount	Is the result of the transmit error count (this parameter can be NULL)
Output	*receiveErrorCount	Is the result of the receive error count (this parameter can be NULL)
Output	*status	Is the return transmit/receive error status (this parameter can be NULL)

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
inline eERRORRESULT MCP251XFD_GetTransmitErrorCount(
    MCP251XFD *pComp,
    uint8_t* transmitErrorCount)
```

Get transmit error count of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*transmitErrorCount	Is the result of the transmit error count

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
inline eERRORRESULT MCP251XFD_GetReceiveErrorCount(
    MCP251XFD *pComp,
    uint8_t* receiveErrorCount)
```

Get transmit error count of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*receiveErrorCount	Is the result of the receive error count

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
inline eERRORRESULT MCP251XFD_GetTransmitReceiveErrorStatus(
    MCP251XFD *pComp,
    eMCP251XFD_TXRXErrorStatus* status)
```

Get transmit/receive error status of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*status	Is the return transmit/receive error status

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

```
eERRORRESULT MCP251XFD_GetBusDiagnostic(
      MCP251XFD *pComp,
      MCP251XFD_CiBDIAG0_Register* busDiagnostic0,
      MCP251XFD_CiBDIAG1_Register* busDiagnostic1)
```
Get Bus diagnostic of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Output	*busDiagnostic0	Is the return value that contains separate error counters for receive/transmit and for nominal/data bit rates. The counters work differently than the counters in the CiTREC register. They are simply incremented by one on every error. They are never decremented (this parameter can be NULL)
Output	*busDiagnostic1	Is the return value that keeps track of the kind of error that occurred since the last clearing of the register. The register also contains the error-free message counter (this parameter can be NULL)

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

```
eERRORRESULT MCP251XFD_ClearBusDiagnostic(
      MCP251XFD *pComp,
      bool clearBusDiagnostic0,
      bool clearBusDiagnostic1)
```
Clear Bus diagnostic of the MCP251XFD device.

Parameters

Input	*pComp	Is the pointed structure of the device to be used
Input	clearBusDiagnostic0	Set to 'true' to clear the bus diagnostic0
Input	clearBusDiagnostic1	Set to 'true' to clear the bus diagnostic1

Return

Returns an *eERRORRESULT* value enumerator.
- ERR__PARAMETER_ERROR when pComp is NULL, or Interface functions are NULL

15.16.1. Bus Diagnostic Register 0 structure

This is the Bus Diagnostic Register 0 structure for the bus diagnostic. It contains all the information about bus errors counts on nominal and data bitrates of the CAN controller. This structure is used by *MCP251XFD_GetBusDiagnostic()* function.

Source code:

```c
typedef union MCP251XFD_CiBDIAG0_Register
{
  uint32_t CiBDIAG0;
  uint8_t Bytes[4];
  struct
  {
    uint8_t NominalBitRateReceiveErrorCount;
    uint8_t NominalBitRateTransmitErrorCount;
    uint8_t DataBitRateReceiveErrorCount;
    uint8_t DataBitRateTransmitErrorCount;
  } Reg;
  struct
  {
    uint32_t NRERRCNT: 8;
    uint32_t NTERRCNT: 8;
    uint32_t DRERRCNT: 8;
    uint32_t DTERRCNT: 8;
  } Bits;
} MCP251XFD_CiBDIAG0_Register;
```

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

15.16.1.1. Data fields

uint32_t *CiBDIAG0*
This is the complete word of the CiBDIAG0 register.

uint8_t *Bytes[4]*
This is the byte access to the CiBDIAG0 register.

uint8_t *NominalBitRateReceiveErrorCount*
uint32_t *NRERRCNT: 8*
Nominal Bit Rate Receive Error Counter.

uint8_t *NominalBitRateTransmitErrorCount*
uint32_t *NTERRCNT: 8*
Nominal Bit Rate Transmit Error Counter.

uint8_t *DataBitRateReceiveErrorCount*
uint32_t *DRERRCNT: 8*
Data Bit Rate Receive Error Counter.

uint8_t *DataBitRateTransmitErrorCount*
uint32_t *DTERRCNT: 8*
Data Bit Rate Transmit Error Counter.

15.16.2.   Bus Diagnostic Register 1 structure

This is the Bus Diagnostic Register 1 structure for the bus diagnostic. It contains all the information about message error count and errors related to nominal and data bitrates of messages plus the bus of the CAN controller. This structure is used by *MCP251XFD_GetBusDiagnostic()* function.

Source code:
```
typedef union MCP251XFD_CiBDIAG1_Register
{
  uint32_t CiBDIAG1;
  uint16_t Uint16[sizeof(uint32_t) / sizeof(uint16_t)];
  uint8_t Bytes[4];
  struct
  {
    uint16_t ErrorFreeCounter;
    eMCP251XFD_DiagStatus Flags;
  } Reg;
  struct
  {
    uint32_t EFMSGCNT: 16;
    uint32_t NBIT0ERR:  1;
    uint32_t NBIT1ERR:  1;
    uint32_t NACKERR :  1;
    uint32_t NFORMERR:  1;
    uint32_t NSTUFERR:  1;
    uint32_t NCRCERR :  1;
    uint32_t         :  1;
    uint32_t TXBOERR :  1;
    uint32_t DBIT0ERR:  1;
    uint32_t DBIT1ERR:  1;
    uint32_t         :  1;
    uint32_t DFORMERR:  1;
    uint32_t DSTUFERR:  1;
    uint32_t DCRCERR :  1;
    uint32_t ESI     :  1;
    uint32_t DLCMM   :  1;
  } Bits;
} MCP251XFD_CiBDIAG1_Register;
```

Ref: MCP251XFD driver library guide
                                                                                               (Synchronous driver).docx

15.16.2.1. Data fields

uint32_t *CiBDIAG1*
This is the complete word of the CiBDIAG1 register.

Uint16_t *Uint16[2]*
This is the unsigned int16 access to the CiBDIAG1 register. The first one is for the message error count and the second for the errors related to nominal and data bitrates of messages plus the bus.

uint8_t *Bytes[4]*
This is the byte access to the CiBDIAG1 register.

uint16_t *ErrorFreeCounter*
Nominal Bit Rate Receive Error Counter.

eMCP251XFD_DiagStatus *Flags*
Transmit and Receive Error status.
Type
    enum   *eMCP251XFD_DiagStatus*

struct bitfield *Bits*
Enumerator

EFMSGCNT	Bit 0 to 15	Error Free Message Counter bits
NBIT0ERR	Bit 16	Normal Bitrate: During the transmission of a message (or acknowledge bit, or active error flag, or overload flag), the device wanted to send a dominant level (data or identifier bit logical value '0'), but the monitored bus value was recessive
NBIT1ERR	Bit 17	Normal Bitrate: During the transmission of a message (except for the arbitration field), the device wanted to send a recessive level (bit of logical value '1'), but the monitored bus value was dominant
NACKERR	Bit 18	Normal Bitrate: Transmitted message was not acknowledged
NFORMERR	Bit 19	Normal Bitrate: A fixed format part of a received frame has the wrong format
NSTUFERR	Bit 20	Normal Bitrate: More than 5 equal bits in a sequence have occurred in a part of a received message where this is not allowed
NCRCERR	Bit 21	Normal Bitrate: The CRC check sum of a received message was incorrect. The CRC of an incoming message does not match with the CRC calculated from the received data
TXBOERR	Bit 23	Device went to bus-off (and auto-recovered)
DBIT0ERR	Bit 24	Data Bitrate: During the transmission of a message (or acknowledge bit, or active error flag, or overload flag), the device wanted to send a dominant level (data or identifier bit logical value '0'), but the monitored bus value was recessive
DBIT1ERR	Bit 25	Data Bitrate: During the transmission of a message (except for the arbitration field), the device wanted to send a recessive level (bit of logical value '1'), but the monitored bus value was dominant
DFORMERR	Bit 27	Data Bitrate: A fixed format part of a received frame has the wrong format
DSTUFERR	Bit 28	Data Bitrate: More than 5 equal bits in a sequence have occurred in a part of a received message where this is not allowed
DCRCERR	Bit 29	Data Bitrate: The CRC check sum of a received message was incorrect. The CRC of an incoming message does not match with the CRC calculated from the received data
ESI	Bit 30	ESI flag of a received CAN FD message was set
DLCMM	Bit 31	DLC Mismatch bit. During a transmission or reception, the specified DLC is larger than the PLSIZE of the FIFO element

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

15.16.3. Enumerators

enum eMCP251XFD_TXRXErrorStatus

Transmit and Receive Error status.

Enumerator

MCP251XFD_TX_RX_WARNING_STATE	0x01	Transmitter or Receiver is in Error Warning State
MCP251XFD_TX_NO_ERROR	0x00	No Transmit Error
MCP251XFD_TX_WARNING_STATE	0x04	Transmitter in Error Warning State
MCP251XFD_TX_BUS_PASSIVE_STATE	0x10	Transmitter in Error Passive State
MCP251XFD_TX_BUS_OFF_STATE	0x20	Transmitter in Bus Off State
MCP251XFD_RX_WARNING_STATE	0x02	Receiver in Error Warning State
MCP251XFD_RX_BUS_PASSIVE_STATE	0x08	Receiver in Error Passive State

enum eMCP251XFD_DiagStatus

Transmit Event FIFO status.

Enumerator

MCP251XFD_DIAG_NBIT0_ERR	0x0001	Normal Bitrate: During the transmission of a message (or acknowledge bit, or active error flag, or overload flag), the device wanted to send a dominant level (data or identifier bit logical value '0'), but the monitored bus value was recessive
MCP251XFD_DIAG_NBIT1_ERR	0x0002	Normal Bitrate: During the transmission of a message (except for the arbitration field), the device wanted to send a recessive level (bit of logical value '1'), but the monitored bus value was dominant
MCP251XFD_DIAG_NACK_ERR	0x0004	Normal Bitrate: Transmitted message was not acknowledged
MCP251XFD_DIAG_NFORM_ERR	0x0008	Normal Bitrate: A fixed format part of a received frame has the wrong format
MCP251XFD_DIAG_NSTUFF_ERR	0x0010	Normal Bitrate: More than 5 equal bits in a sequence have occurred in a part of a received message where this is not allowed
MCP251XFD_DIAG_NCRC_ERR	0x0020	Normal Bitrate: The CRC check sum of a received message was incorrect. The CRC of an incoming message does not match with the CRC calculated from the received data
MCP251XFD_DIAG_TXBO_ERR	0x0080	Device went to bus-off (and auto-recovered)
MCP251XFD_DIAG_DBIT0_ERR	0x0100	Data Bitrate: During the transmission of a message (or acknowledge bit, or active error flag, or overload flag), the device wanted to send a dominant level (data or identifier bit logical value '0'), but the monitored bus value was recessive
MCP251XFD_DIAG_DBIT1_ERR	0x0200	Data Bitrate: During the transmission of a message (except for the arbitration field), the device wanted to send a recessive level (bit of logical value '1'), but the monitored bus value was dominant
MCP251XFD_DIAG_DFORM_ERR	0x0800	Data Bitrate: A fixed format part of a received frame has the wrong format
MCP251XFD_DIAG_DSTUFF_ERR	0x1000	Data Bitrate: More than 5 equal bits in a sequence have occurred in a part of a received message where this is not allowed
MCP251XFD_DIAG_DCRC_ERR	0x2000	Data Bitrate: The CRC check sum of a received message was incorrect. The CRC of an incoming message does not match with the CRC calculated from the received data
MCP251XFD_DIAG_ESI_SET	0x4000	ESI flag of a received CAN FD message was set
MCP251XFD_DIAG_DLC_MISMATCH	0x8000	DLC Mismatch bit. During a transmission or reception, the specified DLC is larger than the PLSIZE of the FIFO element

15.17. Tools

```
uint32_t MCP251XFD_MessageIDtoObjectMessageIdentifier(
      uint32_t messageID,
      bool extended,
      bool UseSID11)
```
Message ID to Object Message Identifier.

Parameters

Input	messageID	Is the message ID to convert
Input	extended	Indicate if the message ID is extended or standard
Input	UseSID11	Indicate if the message ID use the SID11

Return

Returns the Message ID.


```
uint32_t MCP251XFD_ObjectMessageIdentifierToMessageID(
      uint32_t objectMessageID,
      bool extended,
      bool UseSID11)
```
Object Message Identifier to Message ID.

Parameters

Input	objectMessageID	Is the object message ID to convert
Input	extended	Indicate if the object message ID is extended or standard
Input	UseSID11	Indicate if the object message ID use the SID11

Return

Returns the Object Message Identifier.


```
uint32_t MCP251XFD_PayloadToByte(eMCP251XFD_PayloadSize payload)
```
Payload to Byte Count.

Parameters

Input	payload	Is the enum of Message Payload Size (8, 12, 16, 20, 24, 32, 48 or 64 bytes)

Return

Returns the byte count.


```
uint32_t MCP251XFD_DLCToByte(eMCP251XFD_DataLength dlc, bool isCANFD)
```
Data Length Content to Byte Count.

Parameters

|		
Input	dlc	Is the enum of Message DLC Size (0, 1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 20, 24, 32, 48 or 64 bytes)
Input	isCANFD	Indicate if the DLC is from a CAN-FD frame or not

Return

Returns the byte count.

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

16.   TROUBLESHOOTING

16.1.   Regularly get MCP251XFD_INT_SPI_CRC_EVENT

If your SPI interface with the device is not properly configured, you can get the MCP251XFD_INT_SPI_CRC_EVENT. The SPI timing (see Table 5) is very important with this component.

Example, here the T_{SCK2CS} (32ns) is less than T_{SCK} (54ns). The instruction sent to the device is to clear this interrupt but at the end of the last CRC byte the device issue directly a new MCP251XFD_INT_SPI_CRC_EVENT for this transfer by asserting its INT pin.
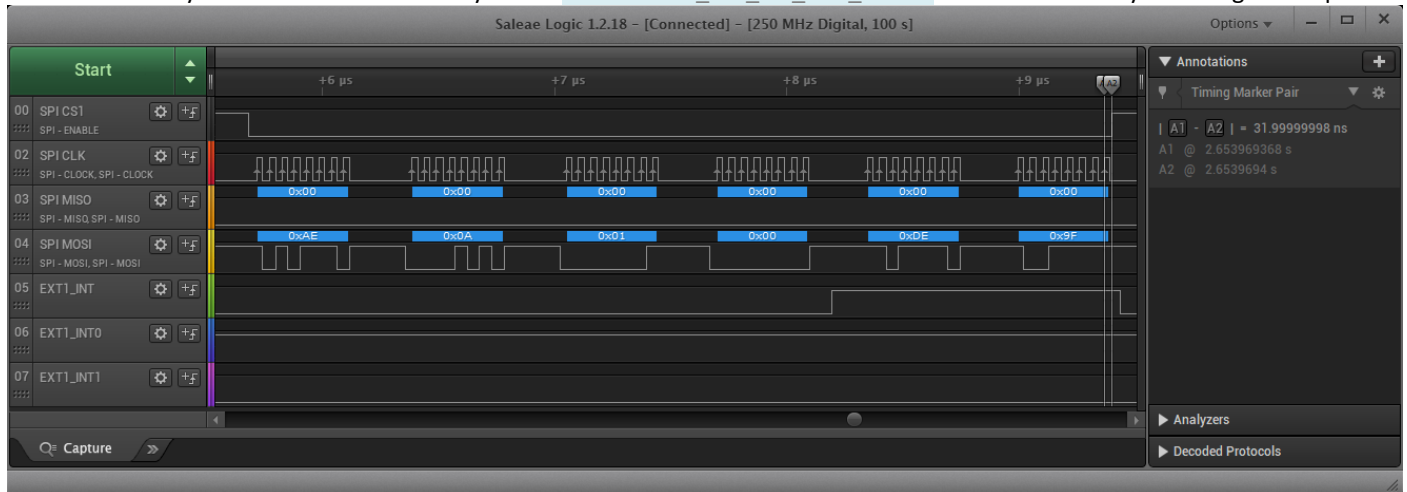
As you understand, just put a higher T_{SCK2CS} time (superior or equal to T_{SCK}) to correct this issue.

Ref: MCP251XFD driver library guide
                                                                                        (Synchronous driver).docx

17.   EXAMPLE OF "CONF_MCP251XFD.H" FILE

Source:

```
#ifndef CONF_MCP251XFD_H
#define CONF_MCP251XFD_H
//========================================================================

// If in debug mode, check NULL parameters that are mandatory in each functions of the driver
#ifdef DEBUG
#  define CHECK_NULL_PARAM
#endif

// This define the max size for the transfer buffer. Adjust for 1 max full frame that is needed of all controllers
// in use otherwise, the message will be cut into parts that slow the transfer
// In case of use write safe by all controllers set 9, which is the minimum allowed, because all data will be send
// by 4 bytes max
#define MCP251XFD_TRANS_BUF_SIZE  ( 2+1+8+64+4+2 ) // here it is set to 1 full CAN-FD frame with CRC
// (2 for Command + 1 for length + 8 for CAN frame header + 64 for CAN-FD max payload + 4 for time stamp + 2 for CRC)

//------------------------------------------------------------------------------
#endif /* CONF_MCP251XFD_H */
```

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

18.   EXAMPLE OF DRIVER INTERFACE HANDLE FUNCTIONS

This is an extract from the project for the SAMV71 Xplained Ultra board.

Example of driver interface handle functions in a .c file:

```c
//=============================================================================
// MCP251XFD_X get millisecond
//=============================================================================
uint32_t GetCurrentms_V71(void)
{
  return msCount;
}



//=============================================================================
// MCP251XFD_X compute CRC16-CMS
//=============================================================================
uint16_t ComputeCRC16_V71(const uint8_t* data, size_t size)
{
  return ComputeCRC16CMS(data, size);
}



//*****************************************************************************
//=============================================================================
// MCP251XFD SPI driver interface configuration for the ATSAMV71
//=============================================================================
eERRORRESULT MCP251XFD_InterfaceInit_V71(void *pIntDev, uint8_t chipSelect, const uint32_t sckFreq)
{
  if (pIntDev == NULL) return ERR__SPI_PARAMETER_ERROR;
  Spi *SPI_Ext = (Spi *)pIntDev;       // MCU specific: #define SPI0  ((Spi*)0x40008000U) // (SPI0  ) Base Address

  ioport_set_pin_mode(SPI0_NPCS1_GPIO, SPI0_NPCS1_FLAGS);
  ioport_disable_pin(SPI0_NPCS1_GPIO);
  ioport_set_pin_mode(SPI0_NPCS3_GPIO, SPI0_NPCS3_FLAGS);
  ioport_disable_pin(SPI0_NPCS3_GPIO);

  //--- Configure an SPI peripheral ---
  spi_enable_clock(SPI_Ext);
  spi_disable(SPI_Ext);
  if (SPIconfigured == false)
  {
    spi_reset(SPI_Ext);
    spi_set_lastxfer(SPI_Ext);
    spi_set_master_mode(SPI_Ext);
    spi_disable_mode_fault_detect(SPI_Ext); // For multimaster SPI bus. Not used here
    spi_disable_peripheral_select_decode(SPI_Ext);
    spi_set_variable_peripheral_select(SPI_Ext);
    spi_enable_tx_on_rx_empty(SPI_Ext);
  }
  spi_set_clock_polarity(SPI_Ext, chipSelect, SPI_CLK_POLARITY);
  spi_set_clock_phase(SPI_Ext, chipSelect, SPI_CLK_PHASE);
  spi_set_bits_per_transfer(SPI_Ext, chipSelect, SPI_CSR_BITS_8_BIT);
  spi_configure_cs_behavior(SPI_Ext, chipSelect, SPI_CS_KEEP_LOW);
//  spi_configure_cs_behavior(SPI_Ext, chipSelect, SPI_CS_RISE_FORCED);
//  spi_configure_cs_behavior(SPI_Ext, chipSelect, SPI_CS_RISE_NO_TX);

  int16_t div = spi_calc_baudrate_div(sckFreq, sysclk_get_peripheral_hz());
  if (div < 0) return ERR__SPI_CONFIG_ERROR;
  spi_set_baudrate_div(SPI_Ext, chipSelect, (uint8_t)div);
  spi_set_transfer_delay(SPI_Ext, chipSelect, SPI_DLYBS, SPI_DLYBCT);
//  spi_set_delay_between_chip_select(SPI_Ext, SPI_DLYBCS);

  spi_enable(SPI_Ext);
  SPIconfigured = true;
  return ERR_OK;
}
```

```
//=============================================================================
// MCP251XFD SPI transfer data for the ATSAMV71
//=============================================================================
eERRORRESULT MCP251XFD_InterfaceTransfer_V71(void *pIntDev, uint8_t chipSelect, uint8_t *txData, uint8_t *rxData,
size_t size)
{
  if (pIntDev == NULL) return ERR__SPI_PARAMETER_ERROR;
  if (txData == NULL) return ERR__SPI_PARAMETER_ERROR;
  Spi *SPI_Ext = (Spi *)pIntDev;         // MCU specific: #define SPI0  ((Spi*)0x40008000U) // (SPI0) Base Address
  uint8_t DataRead;

  __disable_irq();
  uint32_t Timeout;
  while (size > 0)
  {
    //--- Transmit data ---
    Timeout = TIMEOUT_SPI_INTERFACE;
    while (!(SPI_Ext->SPI_SR & SPI_SR_TDRE))   // SPI Tx ready ?
      if (!Timeout--) return ERR__SPI_TIMEOUT; // Timeout ? return an error

    uint32_t value = SPI_TDR_TD(*txData) | SPI_TDR_PCS(spi_get_pcs(chipSelect));
    if (size == 1) value |= SPI_TDR_LASTXFER;
    SPI_Ext->SPI_TDR = value;
    txData++;

    //--- Receive data ---
    Timeout = TIMEOUT_SPI_INTERFACE;
    while (!(SPI_Ext->SPI_SR & SPI_SR_RDRF))   // SPI Rx ready ?
      if (!Timeout--) return ERR__SPI_TIMEOUT; // Timeout ? return an error

    DataRead = (uint8_t)(SPI_Ext->SPI_RDR & 0xFF);
    if (rxData != NULL) *rxData = DataRead;
    rxData++;

    size--;
  }
  SPI_Ext->SPI_CR |= SPI_CR_LASTXFER;
  __enable_irq();

  return ERR_OK;
}
```

Ref: MCP251XFD driver library guide
(Synchronous driver).docx

Example of driver interface handle functions in a .h file:

```c
/*! @brief MCP251XFD_X get millisecond
 *
 * This function will be called when the driver needs to get current millisecond
 */
uint32_t GetCurrentms_V71(void);


/*! @brief MCP251XFD_X compute CRC16-CMS
 *
 * This function will be called when a CRC16-CMS computation is needed (ie. in CRC mode or Safe Write). In normal
mode, this can point to NULL.
 * @param[in] *data Is the pointed byte stream
 * @param[in] size Is the size of the pointed byte stream
 * @return The CRC computed
 */
uint16_t ComputeCRC16_V71(const uint8_t* data, size_t size);



//********************************************************************************************************************

/*! @brief MCP251XFD_Ext1 SPI interface configuration for the ATSAMV71
 *
 * This function will be called at driver initialization to configure the interface driver SPI
 * @param[in] *pIntDev Is the MCP251XFD_Desc.InterfaceDevice of the device that call the interface initialization
 * @param[in] chipSelect Is the Chip Select index to use for the SPI initialization
 * @param[in] sckFreq Is the SCK frequency in Hz to set at the interface initialization
 * @return Returns an #eERRORRESULT value enum
 */
eERRORRESULT MCP251XFD_InterfaceInit_V71(void *pIntDev, uint8_t chipSelect, const uint32_t sckFreq);


/*! @brief MCP251XFD_Ext1 SPI transfer for the ATSAMV71
 *
 * This function will be called at driver read/write data from/to the interface driver SPI
 * @param[in] *pIntDev Is the MCP251XFD_Desc.InterfaceDevice of the device that call this function
 * @param[in] chipSelect Is the Chip Select index to use for the SPI transfer
 * @param[in] *txData Is the buffer to be transmit to through the SPI interface
 * @param[out] *rxData Is the buffer to be received to through the SPI interface (can be NULL if it's just a send of
data)
 * @param[in] size Is the size of data to be send and received trough SPI. txData and rxData shall be at least the
same size
 * @return Returns an #eERRORRESULT value enum
 */
eERRORRESULT MCP251XFD_InterfaceTransfer_V71(void *pIntDev, uint8_t chipSelect, uint8_t *txData, uint8_t *rxData,
size_t size);
```

Ref: MCP251XFD driver library guide
(Synchronous driver).docx