SPI_MASTER

Data Structures

Here are the data structures with brief descriptions:

SPI_MASTER	Initialization parameters of SPI_MASTER APP
SPI_MASTER_CONFIG	Configuration parameters of SPI_MASTER APP
SPI_MASTER_GPIO	Port pin selection for communication
SPI_MASTER_GPIO_CONFIG	Pin configuration for the selected pins
SPI_MASTER_RUNTIME	Structure to hold the dynamic variables for the SPI_MASTER communication

SPI_MASTER

Data Fields

SPI_MASTER Struct Reference

Detailed Description

Initialization parameters of **SPI_MASTER** APP.

Definition at line **294** of file **SPI_MASTER.h**.

```
#include <SPI_MASTER.h>
```

Data Fields

XMC_USIC_CH_t *const	**channel**
const **SPI_MASTER_CONFIG_t** *const	**config**
SPI_MASTER_RUNTIME_t *const	**runtime**

Field Documentation

XMC_USIC_CH_t* const SPI_MASTER::channel

Reference to SPI channel

Definition at line **296** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_AbortReceive()**, **SPI_MASTER_AbortTransmit()**, **SPI_MASTER_ClearFlag()**, **SPI_MASTER_DisableEvent()**, **SPI_MASTER_DisableSlaveSelectSignal()**, **SPI_MASTER_EnableEvent()**, **SPI_MASTER_EnableSlaveSelectSignal()**, **SPI_MASTER_GetFlagStatus()**, **SPI_MASTER_GetReceivedWord()**, **SPI_MASTER_IsRxFIFOEmpty()**, **SPI_MASTER_IsTxFIFOFull()**, **SPI_MASTER_RXFIFO_ClearEvent()**, **SPI_MASTER_RXFIFO_DisableEvent()**, **SPI_MASTER_RXFIFO_EnableEvent()**, **SPI_MASTER_RXFIFO_GetEvent()**, **SPI_MASTER_SetBaudRate()**, **SPI_MASTER_SetRXFIFOTriggerLimit()**, **SPI_MASTER_SetTXFIFOTriggerLimit()**, **SPI_MASTER_TransmitWord()**, **SPI_MASTER_TXFIFO_ClearEvent()**, **SPI_MASTER_TXFIFO_DisableEvent()**, **SPI_MASTER_TXFIFO_EnableEvent()**, and **SPI_MASTER_TXFIFO_GetEvent()**.

const SPI_MASTER_CONFIG_t* const SPI_MASTER::config

Reference to the **SPI_MASTER** configuration structure

Definition at line **297** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_AbortReceive()**, **SPI_MASTER_AbortTransmit()**, **SPI_MASTER_EnableSlaveSelectSignal()**, **SPI_MASTER_Init()**, **SPI_MASTER_Receive()**, **SPI_MASTER_SetBaudRate()**, **SPI_MASTER_SetRXFIFOTriggerLimit()**, **SPI_MASTER_SetTXFIFOTriggerLimit()**, **SPI_MASTER_Transfer()**, and **SPI_MASTER_Transmit()**.

SPI_MASTER_RUNTIME_t* const SPI_MASTER::runtime

Reference to **SPI_MASTER** dynamic configuration structure

Definition at line **298** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_AbortReceive()**, **SPI_MASTER_AbortTransmit()**, **SPI_MASTER_IsRxBusy()**, **SPI_MASTER_IsTxBusy()**, **SPI_MASTER_SetBaudRate()**, **SPI_MASTER_SetMode()**, **SPI_MASTER_Transfer()**, and **SPI_MASTER_TransmitWord()**.

The documentation for this struct was generated from the following file:

- **SPI_MASTER.h**

SPI_MASTER

Data Fields

SPI_MASTER_CONFIG Struct Reference

Detailed Description

Configuration parameters of **SPI_MASTER** APP.

Definition at line **227** of file **SPI_MASTER.h**.

```
#include <SPI_MASTER.h>
```

Data Fields

XMC_SPI_CH_CONFIG_t *const	**channel_co**
SPI_MASTER_lInit_functionhandler	**fptr_spi_ma**
const **SPI_MASTER_GPIO_t** *const	**mosi_0_pin**
const **SPI_MASTER_GPIO_CONFIG_t** *const	**mosi_0_pin**
const **SPI_MASTER_GPIO_t** *const	**mosi_1_pin**
const **SPI_MASTER_GPIO_CONFIG_t** *const	**mosi_1_pin**
const **SPI_MASTER_GPIO_t** *const	**mosi_2_pin**
const **SPI_MASTER_GPIO_CONFIG_t** *const	**mosi_2_pin**
const **SPI_MASTER_GPIO_t** *const	**mosi_3_pin**
const **SPI_MASTER_GPIO_CONFIG_t** *const	**mosi_3_pin**
const **SPI_MASTER_GPIO_t** *const	**sclk_out_p**
const **SPI_MASTER_GPIO_CONFIG_t** *const	**sclk_out_p**
const **SPI_MASTER_GPIO_t** *const	**slave_selec**
const **SPI_MASTER_GPIO_CONFIG_t**	**slave_selec**

*const [8]	
SPI_MASTER_functionhandler	**tx_cbhandl**
SPI_MASTER_functionhandler	**rx_cbhandl**
SPI_MASTER_functionhandler	**parity_cbh**
XMC_USIC_CH_FIFO_SIZE_t	**tx_fifo_size**
XMC_USIC_CH_FIFO_SIZE_t	**rx_fifo_size**
XMC_SPI_CH_BRG_SHIFT_CLOCK_PASSIVE_LEVEL_t	**shift_clk_p**
SPI_MASTER_TRANSFER_MODE_t	**transmit_m**
SPI_MASTER_TRANSFER_MODE_t	**receive_mo**
XMC_SPI_CH_MODE_t	**spi_master**
uint8_t	**slave_selec**
uint8_t	**leading_tra**
SPI_MASTER_SR_ID_t	**tx_sr**
SPI_MASTER_SR_ID_t	**rx_sr**
SPI_MASTER_SR_ID_t	**parity_sr**

Field Documentation

XMC_SPI_CH_CONFIG_t* const SPI_MASTER_CONFIG::channel_c

Reference to SPI configuration structure

Definition at line **229** of file **SPI_MASTER.h**.

SPI_MASTER_lInit_functionhandler SPI_MASTER_CONFIG::fptr_sp

Function pointer to configure the MUX values

Definition at line **230** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_Init()**.

uint8_t SPI_MASTER_CONFIG::leading_trailing_delay

Delay before and after each frame in terms of SCLK cycles

Definition at line **258** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_SetBaudRate()**.

const SPI_MASTER_GPIO_t* const SPI_MASTER_CONFIG::mosi_0

Reference to mosi 0 pin

Definition at line **233** of file **SPI_MASTER.h**.

const SPI_MASTER_GPIO_CONFIG_t* const SPI_MASTER_CONFI

Reference to mosi 0 pin configuration

Definition at line **234** of file **SPI_MASTER.h**.

const SPI_MASTER_GPIO_t* const SPI_MASTER_CONFIG::mosi_1

Reference to mosi 1 pin

Definition at line **235** of file **SPI_MASTER.h**.

const SPI_MASTER_GPIO_CONFIG_t* const SPI_MASTER_CONFI

Reference to mosi 1 pin configuration

Definition at line **236** of file **SPI_MASTER.h**.

const SPI_MASTER_GPIO_t* const SPI_MASTER_CONFIG::mosi_2

Reference to mosi 2 pin

Definition at line **237** of file **SPI_MASTER.h**.

const SPI_MASTER_GPIO_CONFIG_t* const SPI_MASTER_CONFI

Reference to mosi 2 pin configuration

Definition at line **238** of file **SPI_MASTER.h**.

const SPI_MASTER_GPIO_t* const SPI_MASTER_CONFIG::mosi_3

Reference to mosi 3 pin

Definition at line **239** of file **SPI_MASTER.h**.

const SPI_MASTER_GPIO_CONFIG_t* const SPI_MASTER_CONFI

Reference to mosi 3 pin configuration

Definition at line **240** of file **SPI_MASTER.h**.

SPI_MASTER_functionhandler SPI_MASTER_CONFIG::parity_cbha

callback handler for end of parity error

Definition at line **247** of file **SPI_MASTER.h**.

SPI_MASTER_SR_ID_t SPI_MASTER_CONFIG::parity_sr

Service request number assigned to receive interrupts

Definition at line **261** of file **SPI_MASTER.h**.

SPI_MASTER_TRANSFER_MODE_t SPI_MASTER_CONFIG::receiv

Indicates how the receive mode is being handled

Definition at line **255** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_AbortReceive()**, **SPI_MASTER_Receive()**, and **SPI_MASTER_Transfer()**.

SPI_MASTER_functionhandler SPI_MASTER_CONFIG::rx_cbhandl

callback handler for end of reception

Definition at line **246** of file **SPI_MASTER.h**.

XMC_USIC_CH_FIFO_SIZE_t SPI_MASTER_CONFIG::rx_fifo_size

Number of FIFO entries assigned to the receive FIFO buffer

Definition at line **250** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_AbortReceive()**, and **SPI_MASTER_SetRXFIFOTriggerLimit()**.

SPI_MASTER_SR_ID_t SPI_MASTER_CONFIG::rx_sr

Service request number assigned to receive interrupts

Definition at line **260** of file **SPI_MASTER.h**.

const SPI_MASTER_GPIO_t* const SPI_MASTER_CONFIG::sclk_ou

Reference to sclk out pin

Definition at line **241** of file **SPI_MASTER.h**.

const SPI_MASTER_GPIO_CONFIG_t* const SPI_MASTER_CONFI

Reference to shift clock pin configuration

Definition at line **242** of file **SPI_MASTER.h**.

XMC_SPI_CH_BRG_SHIFT_CLOCK_PASSIVE_LEVEL_t SPI_MASTI

Baudrate Generator shift clock passive level

Definition at line **253** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_SetBaudRate()**.

uint8_t SPI_MASTER_CONFIG::slave_select_lines

Number of slave select lines being used

Definition at line **257** of file **SPI_MASTER.h**.

const SPI_MASTER_GPIO_t* const SPI_MASTER_CONFIG::slave_s

Reference to slave select pin

Definition at line **243** of file **SPI_MASTER.h**.

const SPI_MASTER_GPIO_CONFIG_t* const SPI_MASTER_CONFI

Reference to slave select pin configuration

Definition at line **244** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_EnableSlaveSelectSignal()**.

XMC_SPI_CH_MODE_t SPI_MASTER_CONFIG::spi_master_config_

Defines the SPI transmit mode being used

Definition at line **256** of file **SPI_MASTER.h**.

SPI_MASTER_TRANSFER_MODE_t SPI_MASTER_CONFIG::transm

Indicates how the transmit mode is being handled

Definition at line **254** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_AbortTransmit()**, and **SPI_MASTER_Transmit()**.

SPI_MASTER_functionhandler SPI_MASTER_CONFIG::tx_cbhandl

callback handler for end of transmission

Definition at line **245** of file **SPI_MASTER.h**.

XMC_USIC_CH_FIFO_SIZE_t SPI_MASTER_CONFIG::tx_fifo_size

Number of FIFO entries assigned to the transmit FIFO buffer

Definition at line **249** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_AbortTransmit()**, and **SPI_MASTER_SetTXFIFOTriggerLimit()**.

SPI_MASTER_SR_ID_t SPI_MASTER_CONFIG::tx_sr

Service request number assigned to transmit interrupt

Definition at line **259** of file **SPI_MASTER.h**.

The documentation for this struct was generated from the following file:

- **SPI_MASTER.h**

SPI_MASTER

Data Fields

SPI_MASTER_GPIO Struct Reference

Data structures

Detailed Description

Port pin selection for communication.

Definition at line **208** of file **SPI_MASTER.h**.

```
#include <SPI_MASTER.h>
```

Data Fields

XMC_GPIO_PORT_t *	**port**
uint8_t	**pin**

Field Documentation

uint8_t SPI_MASTER_GPIO::pin

Selected pin

Definition at line **211** of file **SPI_MASTER.h**.

XMC_GPIO_PORT_t* SPI_MASTER_GPIO::port

Reference to the port configuration

Definition at line **210** of file **SPI_MASTER.h**.

The documentation for this struct was generated from the following file:

- **SPI_MASTER.h**

SPI_MASTER

Data Fields

SPI_MASTER_GPIO_CONFIG Struct Reference

Detailed Description

Pin configuration for the selected pins.

Definition at line **217** of file **SPI_MASTER.h**.

```
#include <SPI_MASTER.h>
```

Data Fields

XMC_GPIO_CONFIG_t	**port_config**
XMC_GPIO_HWCTRL_t	**hw_control**
XMC_SPI_CH_SLAVE_SELECT_t	**slave_select_ch**

Field Documentation

XMC_GPIO_HWCTRL_t SPI_MASTER_GPIO_CONFIG::hw_control

hardware control characteristics of the pin

Definition at line **220** of file **SPI_MASTER.h**.

XMC_GPIO_CONFIG_t SPI_MASTER_GPIO_CONFIG::port_config

Properties of the port pin

Definition at line **219** of file **SPI_MASTER.h**.

XMC_SPI_CH_SLAVE_SELECT_t SPI_MASTER_GPIO_CONFIG::sla

Indicates the mapped slave select line

Definition at line **221** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_EnableSlaveSelectSignal()**.

The documentation for this struct was generated from the following file:

- **SPI_MASTER.h**

SPI_MASTER

Data Fields

SPI_MASTER_RUNTIME Struct Reference

Detailed Description

Structure to hold the dynamic variables for the **SPI_MASTER** communication.

Definition at line **267** of file **SPI_MASTER.h**.

```
#include <SPI_MASTER.h>
```

Data Fields

uint32_t	**word_length**
uint32_t	**tx_data_count**
volatile uint32_t	**tx_data_index**
uint32_t	**rx_data_count**
volatile uint32_t	**rx_data_index**
uint8_t *	**rx_data**
uint8_t *	**tx_data**
volatile XMC_SPI_CH_MODE_t	**spi_master_mode**
SPI_MASTER_INPUT_t	**dx0_input**
SPI_MASTER_INPUT_t	**dx0_input_half_duplex**
volatile bool	**rx_busy**
volatile bool	**tx_busy**
volatile bool	**tx_data_dummy**
volatile bool	**rx_data_dummy**

Field Documentation

SPI_MASTER_INPUT_t SPI_MASTER_RUNTIME::dx0_input

```
                        DX0 input channel used for
Rx input, This is utilized when
```

mode is changed to full duplex mode

Definition at line **279** of file **SPI_MASTER.h**.

SPI_MASTER_INPUT_t SPI_MASTER_RUNTIME::dx0_input_half_d

```
          DX0 input channel used for Rx input, Th
is is utilized when
```

mode is changed to half duplex mode

Definition at line **281** of file **SPI_MASTER.h**.

volatile bool SPI_MASTER_RUNTIME::rx_busy

Status flag to indicate busy when a reception is assigned

Definition at line **283** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_AbortReceive()**, **SPI_MASTER_IsRxBusy()**, **SPI_MASTER_SetBaudRate()**, **SPI_MASTER_SetMode()**, and **SPI_MASTER_Transfer()**.

uint8_t* SPI_MASTER_RUNTIME::rx_data

Pointer to the receive data buffer

Definition at line **276** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_AbortReceive()**, and **SPI_MASTER_Transfer()**.

uint32_t SPI_MASTER_RUNTIME::rx_data_count

Number of bytes of data to be received

Definition at line **273** of file **SPI_MASTER.h**.

volatile bool SPI_MASTER_RUNTIME::rx_data_dummy

```
                        Status flag to indicate, re
ceive data has to be neglected or
```

not

Definition at line **286** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_Transfer()**.

volatile uint32_t SPI_MASTER_RUNTIME::rx_data_index

```
                      Indicates the number of bytes
currently available in the
```

rx_data buffer

Definition at line **274** of file **SPI_MASTER.h**.

volatile XMC_SPI_CH_MODE_t SPI_MASTER_RUNTIME::spi_maste

Defines the SPI transmit mode being used

Definition at line **278** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_SetMode()**, **SPI_MASTER_Transfer()**, and **SPI_MASTER_TransmitWord()**.

volatile bool SPI_MASTER_RUNTIME::tx_busy

Status flag to indicate busy when a transmission is assigned

Definition at line **284** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_AbortTransmit()**, **SPI_MASTER_IsTxBusy()**, **SPI_MASTER_SetBaudRate()**, **SPI_MASTER_SetMode()**, and **SPI_MASTER_Transfer()**.

uint8_t* SPI_MASTER_RUNTIME::tx_data

Pointer to the transmit data buffer

Definition at line **277** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_AbortTransmit()**, and **SPI_MASTER_Transfer()**.

uint32_t SPI_MASTER_RUNTIME::tx_data_count

Number of bytes of data to be transmitted

Definition at line **270** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_Transfer()**.

volatile bool SPI_MASTER_RUNTIME::tx_data_dummy

Status flag to indicate, dummy data is being transmitted

Definition at line **285** of file **SPI_MASTER.h**.

Referenced by **SPI_MASTER_AbortReceive()**, **SPI_MASTER_AbortTransmit()**, and **SPI_MASTER_Transfer()**.

volatile uint32_t SPI_MASTER_RUNTIME::tx_data_index

```
                    Index to the byte to be transm
itted next in the tx_data
```

buffer

Definition at line **271** of file **SPI_MASTER.h**.

uint32_t SPI_MASTER_RUNTIME::word_length

Indicates the length of the data word

Definition at line **269** of file **SPI_MASTER.h**.

The documentation for this struct was generated from the following file:

- **SPI_MASTER.h**

SPI_MASTER

Data Structure Index

S

S

SPI_MASTER

Here is a list of all documented struct and union fields with links to the struct/union documentation for each field:

- c -

- channel : **SPI_MASTER**
- channel_config : **SPI_MASTER_CONFIG**
- config : **SPI_MASTER**

- d -

- dx0_input : **SPI_MASTER_RUNTIME**
- dx0_input_half_duplex : **SPI_MASTER_RUNTIME**

- f -

- fptr_spi_master_config : **SPI_MASTER_CONFIG**

- h -

- hw_control : **SPI_MASTER_GPIO_CONFIG**

- l -

- leading_trailing_delay : **SPI_MASTER_CONFIG**

- m -

- mosi_0_pin : **SPI_MASTER_CONFIG**
- mosi_0_pin_config : **SPI_MASTER_CONFIG**

- mosi_1_pin : **SPI_MASTER_CONFIG**
- mosi_1_pin_config : **SPI_MASTER_CONFIG**
- mosi_2_pin : **SPI_MASTER_CONFIG**
- mosi_2_pin_config : **SPI_MASTER_CONFIG**
- mosi_3_pin : **SPI_MASTER_CONFIG**
- mosi_3_pin_config : **SPI_MASTER_CONFIG**

- p -

- parity_cbhandler : **SPI_MASTER_CONFIG**
- parity_sr : **SPI_MASTER_CONFIG**
- pin : **SPI_MASTER_GPIO**
- port : **SPI_MASTER_GPIO**
- port_config : **SPI_MASTER_GPIO_CONFIG**

- r -

- receive_mode : **SPI_MASTER_CONFIG**
- runtime : **SPI_MASTER**
- rx_busy : **SPI_MASTER_RUNTIME**
- rx_cbhandler : **SPI_MASTER_CONFIG**
- rx_data : **SPI_MASTER_RUNTIME**
- rx_data_count : **SPI_MASTER_RUNTIME**
- rx_data_dummy : **SPI_MASTER_RUNTIME**
- rx_data_index : **SPI_MASTER_RUNTIME**
- rx_fifo_size : **SPI_MASTER_CONFIG**
- rx_sr : **SPI_MASTER_CONFIG**

- s -

- sclk_out_pin : **SPI_MASTER_CONFIG**
- sclk_out_pin_config : **SPI_MASTER_CONFIG**
- shift_clk_passive_level : **SPI_MASTER_CONFIG**
- slave_select_ch : **SPI_MASTER_GPIO_CONFIG**
- slave_select_lines : **SPI_MASTER_CONFIG**
- slave_select_pin : **SPI_MASTER_CONFIG**
- slave_select_pin_config : **SPI_MASTER_CONFIG**
- spi_master_config_mode : **SPI_MASTER_CONFIG**

- spi_master_mode : **SPI_MASTER_RUNTIME**

- t -

- transmit_mode : **SPI_MASTER_CONFIG**
- tx_busy : **SPI_MASTER_RUNTIME**
- tx_cbhandler : **SPI_MASTER_CONFIG**
- tx_data : **SPI_MASTER_RUNTIME**
- tx_data_count : **SPI_MASTER_RUNTIME**
- tx_data_dummy : **SPI_MASTER_RUNTIME**
- tx_data_index : **SPI_MASTER_RUNTIME**
- tx_fifo_size : **SPI_MASTER_CONFIG**
- tx_sr : **SPI_MASTER_CONFIG**

- w -

- word_length : **SPI_MASTER_RUNTIME**

SPI_MASTER

- c -

- channel : **SPI_MASTER**
- channel_config : **SPI_MASTER_CONFIG**
- config : **SPI_MASTER**

- d -

- dx0_input : **SPI_MASTER_RUNTIME**
- dx0_input_half_duplex : **SPI_MASTER_RUNTIME**

- f -

- fptr_spi_master_config : **SPI_MASTER_CONFIG**

- h -

- hw_control : **SPI_MASTER_GPIO_CONFIG**

- l -

- leading_trailing_delay : **SPI_MASTER_CONFIG**

- m -

- mosi_0_pin : **SPI_MASTER_CONFIG**
- mosi_0_pin_config : **SPI_MASTER_CONFIG**
- mosi_1_pin : **SPI_MASTER_CONFIG**

- mosi_1_pin_config : **SPI_MASTER_CONFIG**
- mosi_2_pin : **SPI_MASTER_CONFIG**
- mosi_2_pin_config : **SPI_MASTER_CONFIG**
- mosi_3_pin : **SPI_MASTER_CONFIG**
- mosi_3_pin_config : **SPI_MASTER_CONFIG**

- p -

- parity_cbhandler : **SPI_MASTER_CONFIG**
- parity_sr : **SPI_MASTER_CONFIG**
- pin : **SPI_MASTER_GPIO**
- port : **SPI_MASTER_GPIO**
- port_config : **SPI_MASTER_GPIO_CONFIG**

- r -

- receive_mode : **SPI_MASTER_CONFIG**
- runtime : **SPI_MASTER**
- rx_busy : **SPI_MASTER_RUNTIME**
- rx_cbhandler : **SPI_MASTER_CONFIG**
- rx_data : **SPI_MASTER_RUNTIME**
- rx_data_count : **SPI_MASTER_RUNTIME**
- rx_data_dummy : **SPI_MASTER_RUNTIME**
- rx_data_index : **SPI_MASTER_RUNTIME**
- rx_fifo_size : **SPI_MASTER_CONFIG**
- rx_sr : **SPI_MASTER_CONFIG**

- s -

- sclk_out_pin : **SPI_MASTER_CONFIG**
- sclk_out_pin_config : **SPI_MASTER_CONFIG**
- shift_clk_passive_level : **SPI_MASTER_CONFIG**
- slave_select_ch : **SPI_MASTER_GPIO_CONFIG**
- slave_select_lines : **SPI_MASTER_CONFIG**
- slave_select_pin : **SPI_MASTER_CONFIG**
- slave_select_pin_config : **SPI_MASTER_CONFIG**
- spi_master_config_mode : **SPI_MASTER_CONFIG**
- spi_master_mode : **SPI_MASTER_RUNTIME**

- t -

- transmit_mode : **SPI_MASTER_CONFIG**
- tx_busy : **SPI_MASTER_RUNTIME**
- tx_cbhandler : **SPI_MASTER_CONFIG**
- tx_data : **SPI_MASTER_RUNTIME**
- tx_data_count : **SPI_MASTER_RUNTIME**
- tx_data_dummy : **SPI_MASTER_RUNTIME**
- tx_data_index : **SPI_MASTER_RUNTIME**
- tx_fifo_size : **SPI_MASTER_CONFIG**
- tx_sr : **SPI_MASTER_CONFIG**

- w -

- word_length : **SPI_MASTER_RUNTIME**

SPI_MASTER

File List

Here is a list of all documented files with brief descriptions:

- SPI_MASTER.c
- SPI_MASTER.h

SPI_MASTER

SPI_MASTER.c File Reference

Detailed Description

Date
2018-06-20

NOTE: This file is generated by DAVE. Any manual modification done to this file will be lost when the code is regenerated.

Definition in file **SPI_MASTER.c**.

```
#include "spi_master.h"
```

Functions

DAVE_APP_VERSION_t	**SPI_MASTER_GetAppVersion** () Get **SPI_MASTER** APP version. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_Init** (**SPI_MASTER_t** *const handle) Initialize the SPI channel as per the configuration made in GUI. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_SetMode** (**SPI_MASTER_t** *const handle, const XMC_SPI_CH_MODE_t mode) Set the communication mode along with required port configuration. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_SetBaudRate** (**SPI_MASTER_t** *const handle, const uint32_t baud_rate) Set the required baud rate during runtime. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_Transmit** (const **SPI_MASTER_t** *const handle, uint8_t *dataptr, uint32_t count) Transmits the specified number of data words and execute the callback defined in GUI, if enabled. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_Receive** (const **SPI_MASTER_t** *const handle, uint8_t *dataptr, uint32_t count) Receives the specified number of data words and execute the callback defined

	in GUI, if enabled. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_Transfer** (const **SPI_MASTER_t** *const handle, uint8_t *tx_dataptr, uint8_t *rx_dataptr, uint32_t count) Transmits and Receives the specified number of data words and execute the receive callback if it is enabled in GUI. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_AbortReceive** (const **SPI_MASTER_t** *const handle) Stops the active data reception request. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_AbortTransmit** (const **SPI_MASTER_t** *const handle) Aborts the ongoing data transmission. More...

Function Documentation

SPI_MASTER_STATUS_t SPI_MASTER_AbortReceive (const SPI_M

Stops the active data reception request.

Parameters

handle Pointer to static and dynamic content of APP configuration.

Returns

None

Description:

If a reception is in progress, it will be stopped. When a reception request is active, user will not be able to place a new receive request till the active reception is complete. This API can stop the progressing reception to make a new receive request.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation (includes SFR declaration)
//Description:
//Transmits the string "Infineon DAVE application" to the slave.
//Starts to receive data from slave, checks if the first byte is 0x55.
//If so, aborts the reception and retransmits 0x55 to slave.
int main(void)
{
DAVE_STATUS_t status;
uint8_t Send_Data[] = "Infineon DAVE application.";
uint8_t Rec_Data[64];
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
```

```
sizeof(Send_Data));
while(SPI_MASTER_0.runtime->tx_busy);
SPI_MASTER_Receive(&SPI_MASTER_0, Rec_Data, 15U);
if(SPI_MASTER_0.runtime->rx_data[0] == 0x55)
{
SPI_MASTER_AbortReceive(&SPI_MASTER_0);
SPI_MASTER_Transmit(&SPI_MASTER_0, Rec_Data, 1);
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **813** of file **SPI_MASTER.c**.

References **SPI_MASTER::channel**, **SPI_MASTER::config**, **SPI_MASTER_CONFIG::receive_mode**, **SPI_MASTER::runtime**, **SPI_MASTER_RUNTIME::rx_busy**, **SPI_MASTER_RUNTIME::rx_data**, **SPI_MASTER_CONFIG::rx_fifo_size**, **SPI_MASTER_AbortTransmit()**, **SPI_MASTER_STATUS_FAILURE**, **SPI_MASTER_STATUS_SUCCESS**, **SPI_MASTER_TRANSFER_MODE_DIRECT**, **SPI_MASTER_TRANSFER_MODE_DMA**, and **SPI_MASTER_RUNTIME::tx_data_dummy**.

SPI_MASTER_STATUS_t SPI_MASTER_AbortTransmit (const SPI_

Aborts the ongoing data transmission.

Parameters

handle Pointer to static and dynamic content of APP configuration.

Returns

None

Description:

If there is a transmission in progress, it will be stopped. If transmit FIFO is used, the existing data will be flushed. After the transmission is stopped, user can start a new transmission without delay.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation (includes SFR declaration)
//Description:
//Transmits test data from buffer Send_Data and aborts it immediately.
//Retransmits data from NewData.
int main(void)
{
DAVE_STATUS_t status;
uint8_t Send_Data[] = "Infineon DAVE application.";
uint8_t NewData[] = "New data message";
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
sizeof(Send_Data));
if(SPI_MASTER_0.runtime->tx_busy)
{
SPI_MASTER_AbortTransmit(&SPI_MASTER_0);
SPI_MASTER_Transmit(&SPI_MASTER_0, NewData,
sizeof(NewData));
}
}
```

```
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **871** of file **SPI_MASTER.c**.

References **SPI_MASTER::channel**, **SPI_MASTER::config**, **SPI_MASTER::runtime**, **SPI_MASTER_STATUS_FAILURE**, **SPI_MASTER_STATUS_SUCCESS**, **SPI_MASTER_TRANSFER_MODE_DIRECT**, **SPI_MASTER_TRANSFER_MODE_DMA**, **SPI_MASTER_CONFIG::transmit_mode**, **SPI_MASTER_RUNTIME::tx_busy**, **SPI_MASTER_RUNTIME::tx_data**, **SPI_MASTER_RUNTIME::tx_data_dummy**, and **SPI_MASTER_CONFIG::tx_fifo_size**.

Referenced by **SPI_MASTER_AbortReceive()**.

SPI_MASTER_STATUS_t SPI_MASTER_Init (SPI_MASTER_t *const

Initialize the SPI channel as per the configuration made in GUI.

Parameters

handle Pointer to static and dynamic content of APP configuration.

Returns

SPI_MASTER_STATUS_t: Status of **SPI_MASTER** driver initialization.
SPI_MASTER_STATUS_SUCCESS - on successful

initialization.
SPI_MASTER_STATUS_FAILURE - if initialization fails.

Description:

Initializes IO pins used for the **SPI_MASTER** communication and configures USIC registers based on the settings provided in the GUI. Calculates divider values PDIV and STEP for a precise baudrate. It also enables configured interrupt flags and service request values.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation (includes SFR declaration)
int main(void)
{
DAVE_STATUS_t status;
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
while(1U)
{
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **201** of file **SPI_MASTER.c**.

References **SPI_MASTER::config**, and **SPI_MASTER_CONFIG::fptr_spi_master_config**.

SPI_MASTER_STATUS_t **SPI_MASTER_Receive** (const SPI_MASTI
uint8_t *
uint32_t
)

Receives the specified number of data words and execute the callback defined in GUI, if enabled.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
dataptr	Pointer to data in which value is written
count	number of data words (word length configured) to be read

Returns

SPI_MASTER_STATUS_t SPI_MASTER_STATUS_SUCCESS : if read is successful
SPI_MASTER_STATUS_BUSY : if SPI channel is busy with other operation

Description:

Data will be received from the SPI slave synchronously. After the requested number of data bytes are received, optionally, the user configured callback function will be executed. Data reception is accomplished using the receive mode selected in the UI. **Interrupt:**
Based on the UI configuration, either standard receive buffer(RBUF) or receive FIFO(OUT) is used for data reception. An interrupt is configured for reading received data from the bus. This function only registers a request to receive a number of data bytes from a USIC channel. If FIFO is configured for reception, the FIFO limit is dynamically configured to optimally utilize the CPU load. Before starting data reception, the receive buffers are flushed. So only those data, received after calling the

API, will be placed in the user buffer. When all the requested number of data bytes are received, the configured callback function will be executed. If a callback function is not configured, the user has to poll for the value of the variable, *handle->runtime->rx_busy* to be false. The value is updated to *false* when all the requested number of data bytes are received.

DMA:

DMA mode is available only in XMC4x family of microcontrollers. In this mode, a DMA channel is configured for receiving data from standard receive buffer(RBUF) to the user buffer. By calling this API, the DMA channel destination address is configured to the user buffer and the channel is enabled. Receive FIFO will not be used when the receive mode is DMA. Before starting data reception, the receive buffers are flushed. So only those data, received after calling the API, will be placed in the user buffer. When all the requested number of data bytes are received, the configured callback function will be executed. If a callback function is not configured, the user has to poll for the value of the variable, *handle->runtime->rx_busy* to be false. The value is updated to *false* when all the requested number of data bytes are received.

Direct

In Direct receive mode, neither interrupt nor DMA is used. The API polls the receive flag to read the received data and waits for all the requested number of bytes to be received. Based on FIFO configuration, either RBUF or OUT register is used for reading received data. Before starting data reception, the receive buffers are flushed. So only those data, received after calling the API, will be placed in the user buffer. ***Note:*** *In Direct mode, the API blocks the CPU until the count of bytes requested is received.*

Example Usage:

```
#include <DAVE.h>
//Description:
//Receives 10 bytes of data from slave.
```

```
int main(void)
{
DAVE_STATUS_t status;
uint8_t ReadData[10];
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
if(SPI_MASTER_Receive(&SPI_MASTER_0, ReadData, 10U))
{
while(SPI_MASTER_0.runtime->rx_busy)
{
}
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **322** of file **SPI_MASTER.c**.

References **SPI_MASTER::config**, **SPI_MASTER_CONFIG::receive_mode**, **SPI_MASTER_STATUS_FAILURE**, **SPI_MASTER_TRANSFER_MODE_DIRECT**, **SPI_MASTER_TRANSFER_MODE_DMA**, and **SPI_MASTER_TRANSFER_MODE_INTERRUPT**.

SPI_MASTER_STATUS_t SPI_MASTER_SetBaudRate (SPI_MASTE

const uint32

)

Set the required baud rate during runtime.

Parameters

handle	handle Pointer to static and dynamic content of APP configuration.
baud_rate	required baud rate

Returns

SPI_MASTER_STATUS_t SPI_MASTER_STATUS_SUCCESS : if updation of baud rate is successful
SPI_MASTER_STATUS_FAILURE : if updation is failed
SPI_MASTER_STATUS_BUSY : if SPI channel is busy with other operation

Description:

While setting the baud rate to avoid noise of the port pins, all the pins are changed to input. After setting the required baud again ports are initialised with the configured settings.

Example Usage:

```
#include <DAVE.h>
//Description:
//The following code changes the SPI master baud rate to 9600 and starts sending the data stored in
//the buffer.
int main(void)
{
DAVE_STATUS_t status;
SPI_MASTER_STATUS_t spi_status;
uint8_t Send_Data[] = "Infineon DAVE application.";
uint32_t baud_rate;
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
```

```
if(status == DAVE_STATUS_SUCCESS)
{
baud_rate = 9600U;
spi_status = SPI_MASTER_SetBaudRate(&SPI_MASTER_0,
baud_rate);
if(spi_status == SPI_MASTER_STATUS_SUCCESS)
{
SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
sizeof(Send_Data));
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **250** of file **SPI_MASTER.c**.

References **SPI_MASTER::channel**, **SPI_MASTER::config**, **SPI_MASTER_CONFIG::leading_trailing_delay**, **SPI_MASTER::runtime**, **SPI_MASTER_RUNTIME::rx_busy**, **SPI_MASTER_CONFIG::shift_clk_passive_level**, **SPI_MASTER_STATUS_BUSY**, **SPI_MASTER_STATUS_SUCCESS**, and **SPI_MASTER_RUNTIME::tx_busy**.

SPI_MASTER_STATUS_t SPI_MASTER_SetMode (SPI_MASTER_t *
const XMC_SPI_
)

Set the communication mode along with required port configuration.

Parameters

handle handle Pointer to static and dynamic content of APP configuration.

mode SPI working mode

Returns

SPI_MASTER_STATUS_t SPI_MASTER_STATUS_SUCCESS : if updation of settings are successful
SPI_MASTER_STATUS_FAILURE : if mode is not supported by the selected pins
SPI_MASTER_STATUS_BUSY : if SPI channel is busy with transmit or receive operation

Description:

To change the mode of communication, it is advised to generate the code in Quad/Dual mode initially. Then changing the mode will be taken care by the APP.

- If code is generated for Quad mode, it is possible to change to other modes like Dual, Half Duplex and Full Duplex
- If code is generated for Dual mode, it is possible to change to other modes like Half Duplex and Full Duplex only
- If code is generated for full-duplex mode, it is possible to change to Half Duplex only

Example Usage:

```
#include <DAVE.h>
//Precondition:
//Configure the SPI_MASTER APP operation mode as 'Quad SPI'.
//Description:
//The following code changes the SPI master device mode to Full duplex mode and starts sending the data stored in
//the buffer.
```

```
int main(void)
{
DAVE_STATUS_t status;
SPI_MASTER_STATUS_t spi_status;
uint8_t Send_Data[] = "Infineon DAVE application.";
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
spi_status = SPI_MASTER_SetMode(&SPI_MASTER_0,
XMC_SPI_CH_MODE_STANDARD);
if(spi_status == SPI_MASTER_STATUS_SUCCESS)
{
SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
sizeof(Send_Data));
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **216** of file **SPI_MASTER.c**.

References **SPI_MASTER::runtime**, **SPI_MASTER_RUNTIME::rx_busy**, **SPI_MASTER_RUNTIME::spi_master_mode**, **SPI_MASTER_STATUS_BUSY**, **SPI_MASTER_STATUS_SUCCESS**, and **SPI_MASTER_RUNTIME::tx_busy**.

SPI_MASTER_STATUS_t SPI_MASTER_Transfer (const SPI_MAST
uint8_t *
uint8_t *
uint32_t
)

Transmits and Receives the specified number of data words and execute the receive callback if it is enabled in GUI.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
tx_dataptr	Pointer to data buffer which has to be send
rx_dataptr	Pointer to data buffer where the received data has to be stored.
count	number of data words (word length configured) to be read and write

Returns

SPI_MASTER_STATUS_t SPI_MASTER_STATUS_SUCCESS : if transfer of data is successful
SPI_MASTER_STATUS_FAILURE : if transfer of data is failed (or) in other than standard full duplex mode
SPI_MASTER_STATUS_BUFFER_INVALID : if passed buffers are NULL pointers (or) length of data transfer is zero.

Description:

Transmits and receives data simultaneously using the SPI channel as a master device. API is applicable only in *Full duplex</> operation mode. Data transfer happens based on the individual modes configured for transmission and reception. Two data pins MOSI and MISO will be used for receiving and transmitting data respectively. A callback function can be configured to execute after completing the transfer when 'Interrupt' or 'DMA' mode is used. The callback function should be configured for End of receive/transfer callback* in the

'Interrupt Settings' tab. The callback function will be executed when the last word of data is received.

Example Usage:

```
#include <DAVE.h>
//Precondition: Operation mode should be 'Full Duplex"
//Description:
//Transmits and Receives 10 bytes of data from slave in parallel.
int main(void)
{
DAVE_STATUS_t status;
uint8_t ReadData[10];
uint8_t SendData[10] = {0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9,
0xA};
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
SPI_MASTER_Transfer(&SPI_MASTER_0, SendData, ReadData,
10);
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **747** of file **SPI_MASTER.c**.

References **SPI_MASTER::config**,

SPI_MASTER_CONFIG::receive_mode, **SPI_MASTER::runtime**, **SPI_MASTER_RUNTIME::rx_busy**, **SPI_MASTER_RUNTIME::rx_data**, **SPI_MASTER_RUNTIME::rx_data_dummy**, **SPI_MASTER_RUNTIME::spi_master_mode**, **SPI_MASTER_STATUS_BUFFER_INVALID**, **SPI_MASTER_STATUS_BUSY**, **SPI_MASTER_STATUS_FAILURE**, **SPI_MASTER_TRANSFER_MODE_DIRECT**, **SPI_MASTER_TRANSFER_MODE_DMA**, **SPI_MASTER_TRANSFER_MODE_INTERRUPT**, **SPI_MASTER_RUNTIME::tx_busy**, **SPI_MASTER_RUNTIME::tx_data**, **SPI_MASTER_RUNTIME::tx_data_count**, and **SPI_MASTER_RUNTIME::tx_data_dummy**.

SPI_MASTER_STATUS_t SPI_MASTER_Transmit (const SPI_MAST
uint8_t *
uint32_t
)

Transmits the specified number of data words and execute the callback defined in GUI, if enabled.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
dataptr	Pointer to data
count	number of data words (word length configured) to be transmitted

Returns

SPI_MASTER_STATUS_t SPI_MASTER_STATUS_SUCCESS : if transmit is successful
SPI_MASTER_STATUS_BUSY : if SPI channel is busy with other operation

Description:

Transmits data using the SPI channel as a master device. Transmission is accomplished using the transmit mode as configured in the UI.

Interrupt:

The data transmission is accomplished using transmit interrupt. User can configure a callback function in the APP UI. When the data is fully transmitted, the callback function will be executed. If transmit FIFO is enabled, the trigger limit is set to 1. So the transmit interrupt will be generated when all the data in FIFO is moved out of FIFO. The APP handle's runtime structure is used to store the data pointer, count, data index and status of transmission. This function only registers a data transmission request if there is no active transmission in progress. Actual data transmission happens in the transmit interrupt service routine. A trigger is generated for the transmit interrupt to start loading the data to the transmit buffer. If transmit FIFO is configured, the data is filled into the FIFO. Transmit interrupt will be generated subsequently when the transmit FIFO is empty. At this point of time, if there is some more data to be transmitted, it is loaded to the FIFO again. When FIFO is not enabled, data is transmitted one byte at a time. On transmission of each byte an interrupt is generated and the next byte is transmitted in the interrupt service routine. Callback function is executed when all the data bytes are transmitted. If a callback function is not configured, user has to poll for the value of *tx_busy* flag of the APP handle structure(*handle->runtime->tx_busy*) to check for the completion of data transmission or use **SPI_MASTER_IsTxBusy()** API.

DMA:

DMA mode is available only in XMC4x family of microcontrollers. A DMA channel is configured to provide data to the SPI channel transmit buffer. This removes the load off the CPU. This API will only configure and enable the DMA channel by specifying the data buffer and count of bytes to transmit. Rest is taken care without the CPU's intervention. User can configure a callback function in the APP UI. When the transmission is complete, the

callback function will be executed. FIFO will not be used in DMA mode. Receive start interrupt is configured for triggering the DMA channel. So each byte is transmitted in the background through the DMA channel. If the callback function is not configured, *handle->runtime->tx_busy* flag can be checked to verify if the transmission is complete. **Direct:**
Data will be transmitted using polling method. Status flags are used to check if data can be transmitted. ***Note:*** *In Direct mode, the API blocks the CPU until the count of bytes requested is transmitted.*

Example Usage:

```
#include <DAVE.h>
//Description:
//Transmits "Infineon" to the slave device.
int main(void)
{
DAVE_STATUS_t status;
uint8_t Send_Data[] = "Infineon";
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
if(SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
sizeof(Send_Data)) == SPI_MASTER_STATUS_SUCCESS)
{
while(SPI_MASTER_0.runtime->tx_busy)
{
}
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
```

```
}
return 1U;
}
```

Definition at line **292** of file **SPI_MASTER.c**.

References **SPI_MASTER::config**, **SPI_MASTER_STATUS_FAILURE**, **SPI_MASTER_TRANSFER_MODE_DIRECT**, **SPI_MASTER_TRANSFER_MODE_DMA**, **SPI_MASTER_TRANSFER_MODE_INTERRUPT**, and **SPI_MASTER_CONFIG::transmit_mode**.

Go to the source code of this file.

SPI_MASTER

SPI_MASTER.h File Reference

Detailed Description

Date
2016-06-20

NOTE: This file is generated by DAVE. Any manual modification done to this file will be lost when the code is regenerated.

Definition in file **SPI_MASTER.h**.

```
#include <xmc_gpio.h> #include <xmc_scu.h>
#include <xmc_spi.h>
#include <DAVE_Common.h>
#include "spi_master_conf.h"
#include "spi_master_extern.h"
```

Data Structures

struct	**SPI_MASTER_GPIO** Port pin selection for communication. More...
struct	**SPI_MASTER_GPIO_CONFIG** Pin configuration for the selected pins. More...
struct	**SPI_MASTER_CONFIG** Configuration parameters of **SPI_MASTER** APP. More...
struct	**SPI_MASTER_RUNTIME** Structure to hold the dynamic variables for the **SPI_MASTER** communication. More...
struct	**SPI_MASTER** Initialization parameters of **SPI_MASTER** APP. More...

Typedefs

typedef struct **SPI_MASTER_GPIO**	**SPI_MASTER_GPIO_t** Port pin selection for communication.
typedef struct **SPI_MASTER_GPIO_CONFIG**	**SPI_MASTER_GPIO_CONFI** Pin configuration for the selected pins.
typedef struct **SPI_MASTER_CONFIG**	**SPI_MASTER_CONFIG_t** Configuration parameters of **SPI_MASTER** APP.
typedef struct **SPI_MASTER_RUNTIME**	**SPI_MASTER_RUNTIME_t** Structure to hold the dynamic variables for the **SPI_MASTEI** communication.
typedef struct **SPI_MASTER**	**SPI_MASTER_t** Initialization parameters of **SPI_MASTER** APP.

Functions

DAVE_APP_VERSION_t	**SPI_MASTER_GetAppVer** Get **SPI_MASTER** APP ver
SPI_MASTER_STATUS_t	**SPI_MASTER_Init** (**SPI_M** handle) Initialize the SPI channel as made in GUI. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_SetMode** (**S** handle, const XMC_SPI_CI Set the communication mod port configuration. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_SetBaudRa** *const handle, const uint32_ Set the required baud rate (
SPI_MASTER_STATUS_t	**SPI_MASTER_Transmit** (c *const handle, uint8_t *data Transmits the specified num and execute the callback de enabled. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_Receive** (c *const handle, uint8_t *data Receives the specified num execute the callback define More...
SPI_MASTER_STATUS_t	**SPI_MASTER_Transfer** (c *const handle, uint8_t *tx_d

	*rx_dataptr, uint32_t count) Transmits and Receives the data words and execute the is enabled in GUI. More...
__STATIC_INLINE uint32_t	**SPI_MASTER_GetFlagSta** **SPI_MASTER_t** *handle, c Returns the state of the spe More...
__STATIC_INLINE void	**SPI_MASTER_ClearFlag** (*handle, const uint32_t flag Clears the status of the spe More...
__STATIC_INLINE bool	**SPI_MASTER_IsTxBusy** (*const handle) return the txbusy flag state
__STATIC_INLINE bool	**SPI_MASTER_IsRxBusy** (*const handle) return the rxbusy flag state
__STATIC_INLINE void	**SPI_MASTER_EnableSlav** **SPI_MASTER_t** *handle, c **SPI_MASTER_SS_SIGNA** Enables the specified slave
__STATIC_INLINE void	**SPI_MASTER_DisableSla** **SPI_MASTER_t** *handle) Disables the all the slave se
__STATIC_INLINE uint16_t	**SPI_MASTER_GetReceive** **SPI_MASTER_t** *const har Provides data received in th

	More...
__STATIC_INLINE void	**SPI_MASTER_TransmitW** **SPI_MASTER_t** *const har data) Transmits a word of data. M
__STATIC_INLINE void	**SPI_MASTER_EnableEve** **SPI_MASTER_t** *const har event_mask) Enables the selected protoc generation. More...
__STATIC_INLINE void	**SPI_MASTER_DisableEve** **SPI_MASTER_t** *const har event_mask) Disables selected events fr interrupt. More...
__STATIC_INLINE void	**SPI_MASTER_SetTXFIFO** **SPI_MASTER_t** *const har limit) Configures trigger limit for t More...
__STATIC_INLINE void	**SPI_MASTER_SetRXFIFO** **SPI_MASTER_t** *const har limit) Configures trigger limit for t More...
__STATIC_INLINE void	**SPI_MASTER_TXFIFO_Er** **SPI_MASTER_t** *const har event) Enables the interrupt event

	FIFO. More...
__STATIC_INLINE void	**SPI_MASTER_TXFIFO_Di** **SPI_MASTER_t** *const har event) Disables the interrupt event FIFO. More...
__STATIC_INLINE uint32_t	**SPI_MASTER_TXFIFO_Ge** **SPI_MASTER_t** *const har Gets the transmit FIFO eve
__STATIC_INLINE void	**SPI_MASTER_TXFIFO_Cl** **SPI_MASTER_t** *const har event) Clears the transmit FIFO ev register. More...
__STATIC_INLINE bool	**SPI_MASTER_IsTxFIFOFu** **SPI_MASTER_t** *const har Checks if the transmit FIFO
__STATIC_INLINE void	**SPI_MASTER_RXFIFO_En** **SPI_MASTER_t** *const har event) Enables the interrupt events FIFO. More...
__STATIC_INLINE void	**SPI_MASTER_RXFIFO_Di** **SPI_MASTER_t** *const har event) Disables the selected interr receive FIFO. More...

__STATIC_INLINE uint32_t	**SPI_MASTER_RXFIFO_G** **SPI_MASTER_t** *const har Get the receive FIFO event
__STATIC_INLINE void	**SPI_MASTER_RXFIFO_Cl** **SPI_MASTER_t** *const har event) Clears the receive FIFO ev register. More...
__STATIC_INLINE bool	**SPI_MASTER_IsRxFIFOE** **SPI_MASTER_t** *const har Checks if receive FIFO is e
SPI_MASTER_STATUS_t	**SPI_MASTER_AbortTrans** **SPI_MASTER_t** *const har Aborts the ongoing data tra
SPI_MASTER_STATUS_t	**SPI_MASTER_AbortRece** **SPI_MASTER_t** *const har Stops the active data recep

enum	**SPI_MASTER_STATUS** {
	SPI_MASTER_STATUS_
	SPI_MASTER_STATUS_F
	SPI_MASTER_STATUS_B
	SPI_MASTER_STATUS_B
	SPI_MASTER_STATUS_
	}
	Return status of the **SPI_M**

enum	**SPI_MASTER_SR_ID** { **SPI_MASTER_SR_ID_0** = **SPI_MASTER_SR_ID_1**, **SPI_MASTER_SR_ID_2**,

	SPI_MASTER_SR_ID_3, **SPI_MASTER_SR_ID_4**, **SPI_MASTER_SR_ID_5** } Service ID for Transmit, Re events. More...
enum	**SPI_MASTER_SS_SIGNA** **SPI_MASTER_SS_SIGN/** **SPI_MASTER_SS_SIGNA** **SPI_MASTER_SS_SIGNA** **SPI_MASTER_SS_SIGNA** **SPI_MASTER_SS_SIGN/** **SPI_MASTER_SS_SIGNA** **SPI_MASTER_SS_SIGNA** **SPI_MASTER_SS_SIGNA** } Slave select signals. More..

enum	**SPI_MASTER_INPUT** {
	SPI_MASTER_INPUT_A
	SPI_MASTER_INPUT_B,
	SPI_MASTER_INPUT_C,
	SPI_MASTER_INPUT_D,
	SPI_MASTER_INPUT_E,
	SPI_MASTER_INPUT_F,
	SPI_MASTER_INPUT_G,
	SPI_MASTER_INPUT_INV
	}
	Enum type which defines R
	More...

enum	**SPI_MASTER_TRANSFEF** **SPI_MASTER_TRANSFEF** **SPI_MASTER_TRANSFEF** **SPI_MASTER_TRANSFEF**

	Enum used to identify the tr either transmit or receive fu
typedef enum **SPI_MASTER_STATUS**	**SPI_MASTER_STATUS_t** Return status of the **SPI_M**
typedef enum **SPI_MASTER_SR_ID**	**SPI_MASTER_SR_ID_t** Service ID for Transmit, Re events.
typedef enum **SPI_MASTER_SS_SIGNAL**	**SPI_MASTER_SS_SIGNA** Slave select signals.
typedef enum **SPI_MASTER_INPUT**	**SPI_MASTER_INPUT_t** Enum type which defines R
typedef enum **SPI_MASTER_TRANSFER_MODE**	**SPI_MASTER_TRANSFEF** Enum used to identify the tr either transmit or receive fu

Go to the source code of this file.

SPI_MASTER

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

- s -

- SPI_MASTER_AbortReceive() : **SPI_MASTER.c** , **SPI_MASTER.h**
- SPI_MASTER_AbortTransmit() : **SPI_MASTER.h** , **SPI_MASTER.c**
- SPI_MASTER_ClearFlag() : **SPI_MASTER.h**
- SPI_MASTER_CONFIG_t : **SPI_MASTER.h**
- SPI_MASTER_DisableEvent() : **SPI_MASTER.h**
- SPI_MASTER_DisableSlaveSelectSignal() : **SPI_MASTER.h**
- SPI_MASTER_EnableEvent() : **SPI_MASTER.h**
- SPI_MASTER_EnableSlaveSelectSignal() : **SPI_MASTER.h**
- SPI_MASTER_GetAppVersion() : **SPI_MASTER.c** , **SPI_MASTER.h**
- SPI_MASTER_GetFlagStatus() : **SPI_MASTER.h**
- SPI_MASTER_GetReceivedWord() : **SPI_MASTER.h**
- SPI_MASTER_GPIO_CONFIG_t : **SPI_MASTER.h**
- SPI_MASTER_GPIO_t : **SPI_MASTER.h**
- SPI_MASTER_Init() : **SPI_MASTER.c** , **SPI_MASTER.h**
- SPI_MASTER_INPUT : **SPI_MASTER.h**
- SPI_MASTER_INPUT_A : **SPI_MASTER.h**
- SPI_MASTER_INPUT_B : **SPI_MASTER.h**
- SPI_MASTER_INPUT_C : **SPI_MASTER.h**
- SPI_MASTER_INPUT_D : **SPI_MASTER.h**
- SPI_MASTER_INPUT_E : **SPI_MASTER.h**

- SPI_MASTER_INPUT_F : **SPI_MASTER.h**
- SPI_MASTER_INPUT_G : **SPI_MASTER.h**
- SPI_MASTER_INPUT_INVALID : **SPI_MASTER.h**
- SPI_MASTER_INPUT_t : **SPI_MASTER.h**
- SPI_MASTER_IsRxBusy() : **SPI_MASTER.h**
- SPI_MASTER_IsRxFIFOEmpty() : **SPI_MASTER.h**
- SPI_MASTER_IsTxBusy() : **SPI_MASTER.h**
- SPI_MASTER_IsTxFIFOFull() : **SPI_MASTER.h**
- SPI_MASTER_Receive() : **SPI_MASTER.c** , **SPI_MASTER.h**
- SPI_MASTER_RUNTIME_t : **SPI_MASTER.h**
- SPI_MASTER_RXFIFO_ClearEvent() : **SPI_MASTER.h**
- SPI_MASTER_RXFIFO_DisableEvent() : **SPI_MASTER.h**
- SPI_MASTER_RXFIFO_EnableEvent() : **SPI_MASTER.h**
- SPI_MASTER_RXFIFO_GetEvent() : **SPI_MASTER.h**
- SPI_MASTER_SetBaudRate() : **SPI_MASTER.c** , **SPI_MASTER.h**
- SPI_MASTER_SetMode() : **SPI_MASTER.c** , **SPI_MASTER.h**
- SPI_MASTER_SetRXFIFOTriggerLimit() : **SPI_MASTER.h**
- SPI_MASTER_SetTXFIFOTriggerLimit() : **SPI_MASTER.h**
- SPI_MASTER_SR_ID : **SPI_MASTER.h**
- SPI_MASTER_SR_ID_0 : **SPI_MASTER.h**
- SPI_MASTER_SR_ID_1 : **SPI_MASTER.h**
- SPI_MASTER_SR_ID_2 : **SPI_MASTER.h**
- SPI_MASTER_SR_ID_3 : **SPI_MASTER.h**
- SPI_MASTER_SR_ID_4 : **SPI_MASTER.h**
- SPI_MASTER_SR_ID_5 : **SPI_MASTER.h**
- SPI_MASTER_SR_ID_t : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_0 : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_1 : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_2 : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_3 : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_4 : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_5 : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_6 : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_7 : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_t : **SPI_MASTER.h**
- SPI_MASTER_STATUS : **SPI_MASTER.h**

- SPI_MASTER_STATUS_BUFFER_INVALID : **SPI_MASTER.h**
- SPI_MASTER_STATUS_BUSY : **SPI_MASTER.h**
- SPI_MASTER_STATUS_FAILURE : **SPI_MASTER.h**
- SPI_MASTER_STATUS_MODE_MISMATCH : **SPI_MASTER.h**
- SPI_MASTER_STATUS_SUCCESS : **SPI_MASTER.h**
- SPI_MASTER_STATUS_t : **SPI_MASTER.h**
- SPI_MASTER_t : **SPI_MASTER.h**
- SPI_MASTER_Transfer() : **SPI_MASTER.c** , **SPI_MASTER.h**
- SPI_MASTER_TRANSFER_MODE : **SPI_MASTER.h**
- SPI_MASTER_TRANSFER_MODE_DIRECT : **SPI_MASTER.h**
- SPI_MASTER_TRANSFER_MODE_DMA : **SPI_MASTER.h**
- SPI_MASTER_TRANSFER_MODE_INTERRUPT : **SPI_MASTER.h**
- SPI_MASTER_TRANSFER_MODE_t : **SPI_MASTER.h**
- SPI_MASTER_Transmit() : **SPI_MASTER.c** , **SPI_MASTER.h**
- SPI_MASTER_TransmitWord() : **SPI_MASTER.h**
- SPI_MASTER_TXFIFO_ClearEvent() : **SPI_MASTER.h**
- SPI_MASTER_TXFIFO_DisableEvent() : **SPI_MASTER.h**
- SPI_MASTER_TXFIFO_EnableEvent() : **SPI_MASTER.h**
- SPI_MASTER_TXFIFO_GetEvent() : **SPI_MASTER.h**

SPI_MASTER

- s -

- SPI_MASTER_AbortReceive() : **SPI_MASTER.c** , **SPI_MASTER.h**
- SPI_MASTER_AbortTransmit() : **SPI_MASTER.h** , **SPI_MASTER.c**
- SPI_MASTER_ClearFlag() : **SPI_MASTER.h**
- SPI_MASTER_DisableEvent() : **SPI_MASTER.h**
- SPI_MASTER_DisableSlaveSelectSignal() : **SPI_MASTER.h**
- SPI_MASTER_EnableEvent() : **SPI_MASTER.h**
- SPI_MASTER_EnableSlaveSelectSignal() : **SPI_MASTER.h**
- SPI_MASTER_GetAppVersion() : **SPI_MASTER.c** , **SPI_MASTER.h**
- SPI_MASTER_GetFlagStatus() : **SPI_MASTER.h**
- SPI_MASTER_GetReceivedWord() : **SPI_MASTER.h**
- SPI_MASTER_Init() : **SPI_MASTER.c** , **SPI_MASTER.h**
- SPI_MASTER_IsRxBusy() : **SPI_MASTER.h**
- SPI_MASTER_IsRxFIFOEmpty() : **SPI_MASTER.h**
- SPI_MASTER_IsTxBusy() : **SPI_MASTER.h**
- SPI_MASTER_IsTxFIFOFull() : **SPI_MASTER.h**
- SPI_MASTER_Receive() : **SPI_MASTER.c** , **SPI_MASTER.h**
- SPI_MASTER_RXFIFO_ClearEvent() : **SPI_MASTER.h**
- SPI_MASTER_RXFIFO_DisableEvent() : **SPI_MASTER.h**
- SPI_MASTER_RXFIFO_EnableEvent() : **SPI_MASTER.h**
- SPI_MASTER_RXFIFO_GetEvent() : **SPI_MASTER.h**
- SPI_MASTER_SetBaudRate() : **SPI_MASTER.h** , **SPI_MASTER.c**

- SPI_MASTER_SetMode() : **SPI_MASTER.h** , **SPI_MASTER.c**
- SPI_MASTER_SetRXFIFOTriggerLimit() : **SPI_MASTER.h**
- SPI_MASTER_SetTXFIFOTriggerLimit() : **SPI_MASTER.h**
- SPI_MASTER_Transfer() : **SPI_MASTER.c** , **SPI_MASTER.h**
- SPI_MASTER_Transmit() : **SPI_MASTER.h** , **SPI_MASTER.c**
- SPI_MASTER_TransmitWord() : **SPI_MASTER.h**
- SPI_MASTER_TXFIFO_ClearEvent() : **SPI_MASTER.h**
- SPI_MASTER_TXFIFO_DisableEvent() : **SPI_MASTER.h**
- SPI_MASTER_TXFIFO_EnableEvent() : **SPI_MASTER.h**
- SPI_MASTER_TXFIFO_GetEvent() : **SPI_MASTER.h**

SPI_MASTER

- SPI_MASTER_CONFIG_t : **SPI_MASTER.h**
- SPI_MASTER_GPIO_CONFIG_t : **SPI_MASTER.h**
- SPI_MASTER_GPIO_t : **SPI_MASTER.h**
- SPI_MASTER_INPUT_t : **SPI_MASTER.h**
- SPI_MASTER_RUNTIME_t : **SPI_MASTER.h**
- SPI_MASTER_SR_ID_t : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_t : **SPI_MASTER.h**
- SPI_MASTER_STATUS_t : **SPI_MASTER.h**
- SPI_MASTER_t : **SPI_MASTER.h**
- SPI_MASTER_TRANSFER_MODE_t : **SPI_MASTER.h**

SPI_MASTER

- SPI_MASTER_INPUT : **SPI_MASTER.h**
- SPI_MASTER_SR_ID : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL : **SPI_MASTER.h**
- SPI_MASTER_STATUS : **SPI_MASTER.h**
- SPI_MASTER_TRANSFER_MODE : **SPI_MASTER.h**

SPI_MASTER

- SPI_MASTER_INPUT_A : **SPI_MASTER.h**
- SPI_MASTER_INPUT_B : **SPI_MASTER.h**
- SPI_MASTER_INPUT_C : **SPI_MASTER.h**
- SPI_MASTER_INPUT_D : **SPI_MASTER.h**
- SPI_MASTER_INPUT_E : **SPI_MASTER.h**
- SPI_MASTER_INPUT_F : **SPI_MASTER.h**
- SPI_MASTER_INPUT_G : **SPI_MASTER.h**
- SPI_MASTER_INPUT_INVALID : **SPI_MASTER.h**
- SPI_MASTER_SR_ID_0 : **SPI_MASTER.h**
- SPI_MASTER_SR_ID_1 : **SPI_MASTER.h**
- SPI_MASTER_SR_ID_2 : **SPI_MASTER.h**
- SPI_MASTER_SR_ID_3 : **SPI_MASTER.h**
- SPI_MASTER_SR_ID_4 : **SPI_MASTER.h**
- SPI_MASTER_SR_ID_5 : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_0 : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_1 : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_2 : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_3 : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_4 : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_5 : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_6 : **SPI_MASTER.h**
- SPI_MASTER_SS_SIGNAL_7 : **SPI_MASTER.h**
- SPI_MASTER_STATUS_BUFFER_INVALID : **SPI_MASTER.h**
- SPI_MASTER_STATUS_BUSY : **SPI_MASTER.h**
- SPI_MASTER_STATUS_FAILURE : **SPI_MASTER.h**
- SPI_MASTER_STATUS_MODE_MISMATCH : **SPI_MASTER.h**
- SPI_MASTER_STATUS_SUCCESS : **SPI_MASTER.h**
- SPI_MASTER_TRANSFER_MODE_DIRECT : **SPI_MASTER.h**

- SPI_MASTER_TRANSFER_MODE_DMA : **SPI_MASTER.h**
- SPI_MASTER_TRANSFER_MODE_INTERRUPT : **SPI_MASTER.h**

SPI_MASTER

SPI_MASTER.h

Go to the documentation of this file.

```
1
88 #ifndef SPI_MASTER_H
89 #define SPI_MASTER_H
90 /*********************************************************************************
91  * HEADER FILES
92
*********************************************************************************/
93 #include <xmc_gpio.h>
94 #include <xmc_scu.h>
95 #include <xmc_spi.h>
96 #include <DAVE_Common.h>
97 #include "spi_master_conf.h"
98
99 #if((SPI_MASTER_DMA_TRANSMIT_MODE == 1U) ||
(SPI_MASTER_DMA_RECEIVE_MODE == 1U))
100 #include "./GLOBAL_DMA/global_dma.h"
101 #endif
102
103 /*********************************************************************************
104  * MACROS
105
*********************************************************************************/
106 #if (!((XMC_LIB_MAJOR_VERSION == 2U) && \
107  (XMC_LIB_MINOR_VERSION >= 1U) && \
108  (XMC_LIB_PATCH_VERSION >= 6U)))
109 #error "SPI_MASTER requires XMC Peripheral Library v2.1.6 or
```

```
higher"
110 #endif
111
112
113 /*
114  * @brief Represents the maximum data size for DMA transaction*/
115 #define SPI_MASTER_DMA_MAXCOUNT (4095U)
116 /*******************************************************************************
117  * ENUMS
118
*******************************************************************************/
126 typedef enum SPI_MASTER_STATUS
127 {
128  SPI_MASTER_STATUS_SUCCESS = 0U,
129  SPI_MASTER_STATUS_FAILURE,
130  SPI_MASTER_STATUS_BUSY,
131  SPI_MASTER_STATUS_BUFFER_INVALID,
132  SPI_MASTER_STATUS_MODE_MISMATCH
136 } SPI_MASTER_STATUS_t;
137
141 typedef enum SPI_MASTER_SR_ID
142 {
143  SPI_MASTER_SR_ID_0 = 0U,
144  SPI_MASTER_SR_ID_1,
145  SPI_MASTER_SR_ID_2,
146  SPI_MASTER_SR_ID_3,
147  SPI_MASTER_SR_ID_4,
148  SPI_MASTER_SR_ID_5
149 } SPI_MASTER_SR_ID_t;
150
154 typedef enum SPI_MASTER_SS_SIGNAL
155 {
156  SPI_MASTER_SS_SIGNAL_0 = 0U,
157  SPI_MASTER_SS_SIGNAL_1,
158  SPI_MASTER_SS_SIGNAL_2,
159  SPI_MASTER_SS_SIGNAL_3,
160  SPI_MASTER_SS_SIGNAL_4,
```

```
161  SPI_MASTER_SS_SIGNAL_5,
162  SPI_MASTER_SS_SIGNAL_6,
163  SPI_MASTER_SS_SIGNAL_7
164 } SPI_MASTER_SS_SIGNAL_t;
165
169 typedef enum SPI_MASTER_INPUT
170 {
171  SPI_MASTER_INPUT_A = 0U,
172  SPI_MASTER_INPUT_B,
173  SPI_MASTER_INPUT_C,
174  SPI_MASTER_INPUT_D,
175  SPI_MASTER_INPUT_E,
176  SPI_MASTER_INPUT_F,
177  SPI_MASTER_INPUT_G,
178  SPI_MASTER_INPUT_INVALID
179 } SPI_MASTER_INPUT_t;
180
184 typedef enum SPI_MASTER_TRANSFER_MODE
185 {
186  SPI_MASTER_TRANSFER_MODE_INTERRUPT,
187  SPI_MASTER_TRANSFER_MODE_DMA,
188  SPI_MASTER_TRANSFER_MODE_DIRECT
189 } SPI_MASTER_TRANSFER_MODE_t;
194 typedef void (*SPI_MASTER_functionhandler)(void);
195 typedef SPI_MASTER_STATUS_t
(*SPI_MASTER_lInit_functionhandler)(void);
196
197 /*********************************************************************
198 * DATA STRUCTURES
199 *********************************************************************
208 typedef struct SPI_MASTER_GPIO
209 {
210  XMC_GPIO_PORT_t* port;
211  uint8_t pin;
212 } SPI_MASTER_GPIO_t;
213
217 typedef struct SPI_MASTER_GPIO_CONFIG
```

```
218 {
219  XMC_GPIO_CONFIG_t port_config;
220  XMC_GPIO_HWCTRL_t hw_control;
221  XMC_SPI_CH_SLAVE_SELECT_t slave_select_ch;
222 } SPI_MASTER_GPIO_CONFIG_t;
223
227  typedef struct SPI_MASTER_CONFIG
228 {
229  XMC_SPI_CH_CONFIG_t * const channel_config;
230  SPI_MASTER_lInit_functionhandler fptr_spi_master_config;
232  /* Port configuration */
233  const SPI_MASTER_GPIO_t* const mosi_0_pin;
234  const SPI_MASTER_GPIO_CONFIG_t* const mosi_0_pin_config;
235  const SPI_MASTER_GPIO_t* const mosi_1_pin;
236  const SPI_MASTER_GPIO_CONFIG_t* const mosi_1_pin_config;
237  const SPI_MASTER_GPIO_t* const mosi_2_pin;
238  const SPI_MASTER_GPIO_CONFIG_t* const mosi_2_pin_config;
239  const SPI_MASTER_GPIO_t* const mosi_3_pin;
240  const SPI_MASTER_GPIO_CONFIG_t* const mosi_3_pin_config;
241  const SPI_MASTER_GPIO_t* const sclk_out_pin;
242  const SPI_MASTER_GPIO_CONFIG_t* const
sclk_out_pin_config;
243  const SPI_MASTER_GPIO_t* const slave_select_pin[8];
244  const SPI_MASTER_GPIO_CONFIG_t* const
slave_select_pin_config[8];
245  SPI_MASTER_functionhandler tx_cbhandler;
246  SPI_MASTER_functionhandler rx_cbhandler;
247  SPI_MASTER_functionhandler parity_cbhandler;
248  /* FIFO configuration */
249  XMC_USIC_CH_FIFO_SIZE_t tx_fifo_size;
250  XMC_USIC_CH_FIFO_SIZE_t rx_fifo_size;
252  /* Clock Settings */
253  XMC_SPI_CH_BRG_SHIFT_CLOCK_PASSIVE_LEVEL_t
shift_clk_passive_level;
254  SPI_MASTER_TRANSFER_MODE_t transmit_mode;
255  SPI_MASTER_TRANSFER_MODE_t receive_mode;
256  XMC_SPI_CH_MODE_t spi_master_config_mode;
```

```
257  uint8_t slave_select_lines;
258  uint8_t leading_trailing_delay;
259  SPI_MASTER_SR_ID_t tx_sr;
260  SPI_MASTER_SR_ID_t rx_sr;
261  SPI_MASTER_SR_ID_t parity_sr;
262 } SPI_MASTER_CONFIG_t;
263
267 typedef struct SPI_MASTER_RUNTIME
268 {
269  uint32_t word_length;
270  uint32_t tx_data_count;
271  volatile uint32_t tx_data_index;
273  uint32_t rx_data_count;
274  volatile uint32_t rx_data_index;
276  uint8_t* rx_data;
277  uint8_t* tx_data;
278  volatile XMC_SPI_CH_MODE_t spi_master_mode;
279  SPI_MASTER_INPUT_t dx0_input;
281  SPI_MASTER_INPUT_t dx0_input_half_duplex;
283  volatile bool rx_busy;
284  volatile bool tx_busy;
285  volatile bool tx_data_dummy;
286  volatile bool rx_data_dummy;
288  } SPI_MASTER_RUNTIME_t;
289
290
294 typedef struct SPI_MASTER
295 {
296  XMC_USIC_CH_t* const channel;
297  const SPI_MASTER_CONFIG_t * const config;
298  SPI_MASTER_RUNTIME_t * const runtime;
299 #if ((SPI_MASTER_DMA_TRANSMIT_MODE == 1U) ||
(SPI_MASTER_DMA_RECEIVE_MODE == 1U))
300  const GLOBAL_DMA_t * const global_dma;
301 #endif
302 #if (SPI_MASTER_DMA_TRANSMIT_MODE == 1U)
303  const XMC_DMA_CH_CONFIG_t * const dma_ch_tx_config;
```

```
304 #endif
305 #if (SPI_MASTER_DMA_RECEIVE_MODE == 1U)
306  const XMC_DMA_CH_CONFIG_t * const dma_ch_rx_config;
307  const GLOBAL_DMA_t * const global_dma_rx;
308  const uint8_t dma_ch_rx_number;
309 #endif
310 #if (SPI_MASTER_DMA_TRANSMIT_MODE == 1U)
311  const uint8_t dma_ch_tx_number;
312 #endif
313 } SPI_MASTER_t;
314
318 /*********************************************************************************
319 * API Prototypes
320 *********************************************************************************
321 #ifdef __cplusplus
322 extern "C" {
323 #endif
324
360 DAVE_APP_VERSION_t SPI_MASTER_GetAppVersion(void);
361
402 SPI_MASTER_STATUS_t SPI_MASTER_Init(SPI_MASTER_t*
const handle);
403
466 SPI_MASTER_STATUS_t
SPI_MASTER_SetMode(SPI_MASTER_t* const handle, const
XMC_SPI_CH_MODE_t mode);
467
524 SPI_MASTER_STATUS_t
SPI_MASTER_SetBaudRate(SPI_MASTER_t* const handle, const
uint32_t baud_rate);
525
610 SPI_MASTER_STATUS_t SPI_MASTER_Transmit(const
SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count);
611
696 SPI_MASTER_STATUS_t SPI_MASTER_Receive(const
SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count);
697
```

```
755 SPI_MASTER_STATUS_t SPI_MASTER_Transfer(const
SPI_MASTER_t *const handle,
756  uint8_t* tx_dataptr,
757  uint8_t* rx_dataptr,
758  uint32_t count);
759
760 #if (SPI_MASTER_INTERRUPT_RECEIVE_MODE == 1U)
761
816 SPI_MASTER_STATUS_t SPI_MASTER_StartReceiveIRQ(const
SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count);
817 #endif
818
819 #if(SPI_MASTER_INTERRUPT_TRANSMIT_MODE == 1U)
820
885 SPI_MASTER_STATUS_t SPI_MASTER_StartTransmitIRQ(const
SPI_MASTER_t *const handle, uint8_t *addr, uint32_t count);
886 #endif
887
888
889 #if(SPI_MASTER_DMA_RECEIVE_MODE == 1U)
890
955 SPI_MASTER_STATUS_t SPI_MASTER_StartReceiveDMA(const
SPI_MASTER_t *const handle, uint8_t *addr, uint32_t block_size);
956 #endif
957
958 #if(SPI_MASTER_DMA_TRANSMIT_MODE == 1U)
959
1025 SPI_MASTER_STATUS_t
SPI_MASTER_StartTransmitDMA(const SPI_MASTER_t *const
handle, uint8_t *addr, uint32_t block_size);
1026 #endif
1027
1074 __STATIC_INLINE uint32_t SPI_MASTER_GetFlagStatus(const
SPI_MASTER_t* handle, const uint32_t flag)
1075 {
1076  XMC_ASSERT("SPI_MASTER_GetFlagStatus:handle NULL" ,
(handle != NULL));
```

```
1077  return (XMC_SPI_CH_GetStatusFlag(handle->channel) & flag);
1078 }
1079
1124 __STATIC_INLINE void SPI_MASTER_ClearFlag(const
SPI_MASTER_t* handle, const uint32_t flag_mask)
1125 {
1126  XMC_ASSERT("SPI_MASTER_ClearFlag:handle NULL" ,
(handle != NULL));
1127  XMC_SPI_CH_ClearStatusFlag(handle->channel, flag_mask);
1128 }
1129
1179 __STATIC_INLINE bool SPI_MASTER_IsTxBusy(const
SPI_MASTER_t* const handle)
1180 {
1181  XMC_ASSERT("SPI_MASTER_IsTxBusy:handle NULL", (handle
!= NULL))
1182  return (handle->runtime->tx_busy);
1183 }
1184
1231 __STATIC_INLINE bool SPI_MASTER_IsRxBusy(const
SPI_MASTER_t* const handle)
1232 {
1233  XMC_ASSERT("SPI_MASTER_IsTxBusy:handle NULL", (handle
!= NULL))
1234  return (handle->runtime->rx_busy);
1235 }
1236
1237
1285 __STATIC_INLINE void
SPI_MASTER_EnableSlaveSelectSignal(const SPI_MASTER_t*
handle, const SPI_MASTER_SS_SIGNAL_t slave)
1286 {
1287  XMC_ASSERT("SPI_MASTER_EnableSlaveSelectSignal:handle
NULL" , (handle != NULL));
1288  XMC_ASSERT("SPI_MASTER_EnableSlaveSelectSignal:Invalid
Slave selection" , ((slave == SPI_MASTER_SS_SIGNAL_0) ||
1289  (slave == SPI_MASTER_SS_SIGNAL_1) ||
```

```
1290  (slave == SPI_MASTER_SS_SIGNAL_2) ||
1291  (slave == SPI_MASTER_SS_SIGNAL_3) ||
1292  (slave == SPI_MASTER_SS_SIGNAL_4) ||
1293  (slave == SPI_MASTER_SS_SIGNAL_5) ||
1294  (slave == SPI_MASTER_SS_SIGNAL_6) ||
1295  (slave == SPI_MASTER_SS_SIGNAL_7))
1296  );
1297  XMC_SPI_CH_EnableSlaveSelect(handle->channel, handle-
>config->slave_select_pin_config[slave]->slave_select_ch);
1298 }
1299
1344 __STATIC_INLINE void
SPI_MASTER_DisableSlaveSelectSignal(const SPI_MASTER_t*
handle)
1345 {
1346  XMC_ASSERT("SPI_MASTER_Transmit:handle NULL" , (handle
!= NULL));
1347  XMC_SPI_CH_DisableSlaveSelect(handle->channel);
1348 }
1349
1365 __STATIC_INLINE uint16_t
SPI_MASTER_GetReceivedWord(const SPI_MASTER_t *const
handle)
1366 {
1367  XMC_ASSERT("SPI_MASTER_GetReceivedWord:handle
NULL" , (handle != NULL));
1368  return XMC_SPI_CH_GetReceivedData(handle->channel);
1369 }
1370
1384 __STATIC_INLINE void SPI_MASTER_TransmitWord(const
SPI_MASTER_t *const handle, const uint16_t data)
1385 {
1386  XMC_ASSERT("SPI_MASTER_TransmitWord:handle NULL" ,
(handle != NULL));
1387  XMC_SPI_CH_Transmit(handle->channel, data, handle-
>runtime->spi_master_mode);
1388 }
```

```
1389
1405 __STATIC_INLINE void SPI_MASTER_EnableEvent(const
SPI_MASTER_t *const handle, const uint32_t event_mask)
1406 {
1407  XMC_ASSERT("SPI_MASTER_EnableEvent:handle NULL" ,
(handle != NULL));
1408  XMC_SPI_CH_EnableEvent(handle->channel, event_mask);
1409 }
1410
1423 __STATIC_INLINE void SPI_MASTER_DisableEvent(const
SPI_MASTER_t *const handle, const uint32_t event_mask)
1424 {
1425  XMC_ASSERT("SPI_MASTER_DisableEvent:handle NULL" ,
(handle != NULL));
1426  XMC_SPI_CH_DisableEvent(handle->channel, event_mask);
1427 }
1428
1443 __STATIC_INLINE void
SPI_MASTER_SetTXFIFOTriggerLimit(const SPI_MASTER_t *const
handle, const uint32_t limit)
1444 {
1445  XMC_ASSERT("SPI_MASTER_SetTXFIFOTriggerLimit:handle
NULL" , (handle != NULL));
1446  XMC_USIC_CH_TXFIFO_SetSizeTriggerLimit(handle->channel,
handle->config->tx_fifo_size, limit);
1447 }
1448
1464 __STATIC_INLINE void
SPI_MASTER_SetRXFIFOTriggerLimit(const SPI_MASTER_t *const
handle, const uint32_t limit)
1465 {
1466  XMC_ASSERT("SPI_MASTER_SetRXFIFOTriggerLimit:handle
NULL" , (handle != NULL));
1467  XMC_USIC_CH_RXFIFO_SetSizeTriggerLimit(handle->channel,
handle->config->rx_fifo_size, limit);
1468 }
1469
```

```
1481 __STATIC_INLINE void
SPI_MASTER_TXFIFO_EnableEvent(const SPI_MASTER_t *const
handle, const uint32_t event)
1482 {
1483  XMC_ASSERT("SPI_MASTER_TXFIFO_EnableEvent:handle
NULL" , (handle != NULL));
1484  XMC_USIC_CH_TXFIFO_EnableEvent(handle->channel, event);
1485 }
1486
1500 __STATIC_INLINE void
SPI_MASTER_TXFIFO_DisableEvent(const SPI_MASTER_t *const
handle, const uint32_t event)
1501 {
1502  XMC_ASSERT("SPI_MASTER_TXFIFO_DisableEvent:handle
NULL" , (handle != NULL));
1503  XMC_USIC_CH_TXFIFO_DisableEvent(handle->channel,
event);
1504 }
1505
1519 __STATIC_INLINE uint32_t
SPI_MASTER_TXFIFO_GetEvent(const SPI_MASTER_t *const
handle)
1520 {
1521  XMC_ASSERT("SPI_MASTER_TXFIFO_GetEvent:handle
NULL" , (handle != NULL));
1522  return XMC_USIC_CH_TXFIFO_GetEvent(handle->channel);
1523 }
1524
1540 __STATIC_INLINE void
SPI_MASTER_TXFIFO_ClearEvent(const SPI_MASTER_t *const
handle, const uint32_t event)
1541 {
1542  XMC_ASSERT("SPI_MASTER_TXFIFO_ClearEvent:handle
NULL" , (handle != NULL));
1543  XMC_USIC_CH_TXFIFO_ClearEvent(handle->channel, event);
1544 }
1545
```

```
1559 __STATIC_INLINE bool SPI_MASTER_IsTxFIFOFull(const
SPI_MASTER_t* const handle)
1560 {
1561  XMC_ASSERT("SPI_MASTER_IsTxFIFOFull:handle NULL",
(handle != NULL))
1562  return XMC_USIC_CH_TXFIFO_IsFull(handle->channel);
1563 }
1564
1577 __STATIC_INLINE void
SPI_MASTER_RXFIFO_EnableEvent(const SPI_MASTER_t *const
handle, const uint32_t event)
1578 {
1579  XMC_ASSERT("SPI_MASTER_RXFIFO_EnableEvent:handle
NULL" , (handle != NULL));
1580  XMC_USIC_CH_RXFIFO_EnableEvent(handle->channel,
event);
1581 }
1582
1595 __STATIC_INLINE void
SPI_MASTER_RXFIFO_DisableEvent(const SPI_MASTER_t *const
handle, const uint32_t event)
1596 {
1597  XMC_ASSERT("SPI_MASTER_RXFIFO_DisableEvent:handle
NULL" , (handle != NULL));
1598  XMC_USIC_CH_RXFIFO_DisableEvent(handle->channel,
event);
1599 }
1600
1613 __STATIC_INLINE uint32_t
SPI_MASTER_RXFIFO_GetEvent(const SPI_MASTER_t *const
handle)
1614 {
1615  XMC_ASSERT("SPI_MASTER_RXFIFO_GetEvent:handle
NULL" , (handle != NULL));
1616  return XMC_USIC_CH_RXFIFO_GetEvent(handle->channel);
1617 }
1618
```

```
1632 __STATIC_INLINE void
SPI_MASTER_RXFIFO_ClearEvent(const SPI_MASTER_t *const
handle, const uint32_t event)
1633 {
1634  XMC_ASSERT("SPI_MASTER_RXFIFO_ClearEvent:handle
NULL" , (handle != NULL));
1635  XMC_USIC_CH_RXFIFO_ClearEvent(handle->channel, event);
1636 }
1637
1650 __STATIC_INLINE bool SPI_MASTER_IsRxFIFOEmpty(const
SPI_MASTER_t* const handle)
1651 {
1652  XMC_ASSERT("SPI_MASTER_IsRxFIFOEmpty:handle NULL",
(handle != NULL))
1653  return XMC_USIC_CH_RXFIFO_IsEmpty(handle->channel);
1654 }
1655
1703 SPI_MASTER_STATUS_t SPI_MASTER_AbortTransmit(const
SPI_MASTER_t *const handle);
1704
1758 SPI_MASTER_STATUS_t SPI_MASTER_AbortReceive(const
SPI_MASTER_t *const handle);
1759
1763 #include "spi_master_extern.h"
1764
1765 #ifdef __cplusplus
1766 }
1767 #endif
1768
1769 #endif /* SPI_MASTER_H */
```

SPI_MASTER

Data structures

Data Structures

struct	**SPI_MASTER_GPIO** Port pin selection for communication. More...
typedef struct **SPI_MASTER_GPIO**	**SPI_MASTER_GPIO_t** Port pin selection for communication.
typedef struct **SPI_MASTER_GPIO_CONFIG**	**SPI_MASTER_GPIO_CONFI** Pin configuration for the selected pins.
typedef struct **SPI_MASTER_CONFIG**	**SPI_MASTER_CONFIG_t** Configuration parameters of **SPI_MASTER** APP.
typedef struct **SPI_MASTER_RUNTIME**	**SPI_MASTER_RUNTIME_t** Structure to hold the dynamic variables for the **SPI_MASTEI** communication.
typedef struct **SPI_MASTER**	**SPI_MASTER_t** Initialization parameters of **SPI_MASTER** APP.

Detailed Description

SPI_MASTER

SPI_MASTER.c

Go to the documentation of this file.

```
1
105 /*******************************************************************************
106  * HEADER FILES
107
*******************************************************************************/
108 #include "spi_master.h"
109
110
111 /*******************************************************************************
112  * MACROS
113
*******************************************************************************/
114 #define SPI_MASTER_WORD_LENGTH_8_BIT (8U) /* This is
used to check while incrementing the data index */
115 #define SPI_MASTER_2_BYTES_PER_WORD (2U) /* Word
length is 16-bits */
116 #define SPI_MASTER_1_BYTE_PER_WORD (1U) /* Word length
is 8-bits */
117
118 #define SPI_MASTER_RECEIVE_INDICATION_FLAG
((uint32_t)XMC_SPI_CH_STATUS_FLAG_RECEIVE_INDICATION | \
119
(uint32_t)XMC_SPI_CH_STATUS_FLAG_ALTERNATIVE_RECEIVE_IN
120
121 #define SPI_MASTER_FIFO_RECEIVE_INDICATION_FLAG
((uint32_t)XMC_USIC_CH_RXFIFO_EVENT_STANDARD | \
```

```
122  (uint32_t)XMC_USIC_CH_RXFIFO_EVENT_ALTERNATE)
123
124 #define SPI_MASTER_RECEIVE_EVENT
((uint32_t)XMC_SPI_CH_EVENT_STANDARD_RECEIVE | \
125  (uint32_t)XMC_SPI_CH_EVENT_ALTERNATIVE_RECEIVE)
126
127 #define SPI_MASTER_FIFO_RECEIVE_EVENT
((uint32_t)XMC_USIC_CH_RXFIFO_EVENT_CONF_STANDARD | \
128
(uint32_t)XMC_USIC_CH_RXFIFO_EVENT_CONF_ALTERNATE)
129 /*********************************************************************
130  * LOCAL DATA
131 *********************************************************************
132
133
134 /*********************************************************************
135  * LOCAL ROUTINES
136
*********************************************************************
137 #if (SPI_MASTER_INTERRUPT_TRANSMIT_MODE == 1U)
138 /* Transmit interrupt handler for the APP */
139 void SPI_MASTER_lTransmitHandler(const SPI_MASTER_t *
const handle);
140 #endif
141
142 #if (SPI_MASTER_INTERRUPT_RECEIVE_MODE == 1U)
143 static SPI_MASTER_STATUS_t SPI_MASTER_lReceiveIRQ(const
SPI_MASTER_t *const handle, uint32_t count);
144 /* This is used to reconfigure the FIFO settings dynamically */
145 static void SPI_MASTER_lReconfigureRxFIFO(const
SPI_MASTER_t * const handle, uint32_t data_size);
146 /* Read data from FIFO */
147 static void SPI_MASTER_lFIFORead(const SPI_MASTER_t *
const handle, const uint32_t bytes_per_word);
148 /* Receive interrupt handler for the APP */
149 void SPI_MASTER_lReceiveHandler(const SPI_MASTER_t *
const handle);
```

```
150 #endif
151
152 #if(SPI_MASTER_DMA_RECEIVE_MODE == 1U)
153 static SPI_MASTER_STATUS_t
SPI_MASTER_lReceiveDMA(const SPI_MASTER_t *const handle,
uint32_t count);
154 #endif
155
156 #if(SPI_MASTER_DIRECT_TRANSMIT_MODE == 1U)
157 static SPI_MASTER_STATUS_t
SPI_MASTER_lStartTransmitPolling(const SPI_MASTER_t *const
handle, uint8_t* dataptr, uint32_t count);
158 #endif
159
160 #if(SPI_MASTER_DIRECT_RECEIVE_MODE == 1U)
161 static SPI_MASTER_STATUS_t
SPI_MASTER_lStartReceivePolling(const SPI_MASTER_t *const
handle, uint8_t* dataptr, uint32_t count);
162 static SPI_MASTER_STATUS_t
SPI_MASTER_lReceivePolling(const SPI_MASTER_t *const handle,
uint32_t count);
163 #endif
164
165 #ifdef SPI_MASTER_PARITY_ERROR
166 /* Protocol interrupt handler for the APP */
167 void SPI_MASTER_lProtocolHandler(const SPI_MASTER_t *
const handle);
168 #endif
169
170 /* Flush RBUF0, RBUF1 */
171 static void SPI_MASTER_lStdRBUFFlush(XMC_USIC_CH_t
*const channel);
172
173 /* This is used to reconfigure the registers while changing the SPI
mode dynamically */
174 static void SPI_MASTER_lPortConfig(const SPI_MASTER_t*
handle);
```

```
175 /* Set the mode of the port pin according to the configuration */
176 static void SPI_MASTER_lPortModeSet(const SPI_MASTER_t*
handle);
177 /* Set the mode of the port pin as input */
178 static void SPI_MASTER_lPortModeReset(const SPI_MASTER_t*
handle);
179 /* Returns whether mode change is valid or not */
180 static SPI_MASTER_STATUS_t
SPI_MASTER_lValidateModeChange(const SPI_MASTER_t * handle,
XMC_SPI_CH_MODE_t mode);
181 /*******************************************************************************
182  * API IMPLEMENTATION
183
*******************************************************************************/
184 /*
185  * API to retrieve the version of the SPI_MASTER
186  */
187 DAVE_APP_VERSION_t SPI_MASTER_GetAppVersion()
188 {
189  DAVE_APP_VERSION_t version;
190
191  version.major = SPI_MASTER_MAJOR_VERSION;
192  version.minor = SPI_MASTER_MINOR_VERSION;
193  version.patch = SPI_MASTER_PATCH_VERSION;
194
195  return version;
196 }
197
198 /*
199  * This function initializes the SPI channel, based on UI
configuration.
200  */
201 SPI_MASTER_STATUS_t SPI_MASTER_Init(SPI_MASTER_t*
const handle)
202 {
203  SPI_MASTER_STATUS_t status;
204
```

```
205  XMC_ASSERT("SPI_MASTER_Init:handle NULL" , (handle !=
NULL));
206
207  /* Configure the port registers and data input registers of SPI
channel */
208  status = handle->config->fptr_spi_master_config();
209
210  return status;
211 }
212
213 /*
214  * Change the SPI mode of communication.
215  */
216 SPI_MASTER_STATUS_t
SPI_MASTER_SetMode(SPI_MASTER_t* const handle,
217  const XMC_SPI_CH_MODE_t mode)
218 {
219  SPI_MASTER_STATUS_t status;
220
221  XMC_ASSERT("SPI_MASTER_Configure:handle NULL" , (handle
!= NULL));
222
223  status = SPI_MASTER_STATUS_SUCCESS;
224
225  if ((false == handle->runtime->tx_busy) && (false == handle-
>runtime->rx_busy))
226  {
227  if (handle->runtime->spi_master_mode != mode)
228  {
229  status = SPI_MASTER_lValidateModeChange(handle, mode);
230
231  if (SPI_MASTER_STATUS_SUCCESS == status)
232  {
233  handle->runtime->spi_master_mode = mode;
234
235  /* This changes the operating mode and related settings */
236  SPI_MASTER_lPortConfig(handle);
```

```
237  }
238  }
239  }
240  else
241  {
242  status = SPI_MASTER_STATUS_BUSY;
243  }
244  return status;
245 }
246
247 /*
248  * Set the baud rate during runtime.
249  */
250 SPI_MASTER_STATUS_t
SPI_MASTER_SetBaudRate(SPI_MASTER_t* const handle, const
uint32_t baud_rate)
251 {
252  SPI_MASTER_STATUS_t status;
253
254  if ((false == handle->runtime->tx_busy) && (false == handle-
>runtime->rx_busy))
255  {
256  /* Stops the SPI channel */
257  status = (SPI_MASTER_STATUS_t)XMC_SPI_CH_Stop(handle-
>channel);
258
259  if (SPI_MASTER_STATUS_SUCCESS == status)
260  {
261  /* Set all the pins as input */
262  SPI_MASTER_lPortModeReset(handle);
263
264  /* Update the new baud rate */
265  status =
(SPI_MASTER_STATUS_t)XMC_SPI_CH_SetBaudrate(handle-
>channel, baud_rate);
266
267  if (SPI_MASTER_STATUS_SUCCESS == status)
```

```
268  {
269  /* Configure Leading/Trailing delay */
270  XMC_SPI_CH_SetSlaveSelectDelay(handle->channel,
(uint32_t)handle->config->leading_trailing_delay);
271  }
272
273  /* Configure the clock polarity and clock delay */
274  XMC_SPI_CH_ConfigureShiftClockOutput(handle->channel,
275  handle->config->shift_clk_passive_level,
276  XMC_SPI_CH_BRG_SHIFT_CLOCK_OUTPUT_SCLK);
277  /* Start the SPI channel */
278  XMC_SPI_CH_Start(handle->channel);
279
280  /* Set the mode of the according the generated configuration */
281  SPI_MASTER_lPortModeSet(handle);
282  }
283  }
284  else
285  {
286  status = SPI_MASTER_STATUS_BUSY;
287  }
288
289  return status;
290 }
291
292 SPI_MASTER_STATUS_t SPI_MASTER_Transmit(const
SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count)
293 {
294  SPI_MASTER_STATUS_t status;
295
296  status = SPI_MASTER_STATUS_FAILURE;
297
298 #if (SPI_MASTER_INTERRUPT_TRANSMIT_MODE == 1U)
299  if (handle->config->transmit_mode ==
SPI_MASTER_TRANSFER_MODE_INTERRUPT)
300  {
301  status = SPI_MASTER_StartTransmitIRQ(handle, dataptr, count);
```

```
302  }
303 #endif
304
305 #if (SPI_MASTER_DMA_TRANSMIT_MODE == 1U)
306  if (handle->config->transmit_mode ==
SPI_MASTER_TRANSFER_MODE_DMA)
307  {
308  status = SPI_MASTER_StartTransmitDMA(handle, dataptr, count);
309  }
310 #endif
311
312 #if (SPI_MASTER_DIRECT_TRANSMIT_MODE == 1U)
313  if (handle->config->transmit_mode ==
SPI_MASTER_TRANSFER_MODE_DIRECT)
314  {
315  status = SPI_MASTER_lStartTransmitPolling(handle, dataptr,
count);
316  }
317 #endif
318
319  return status;
320 }
321
322 SPI_MASTER_STATUS_t SPI_MASTER_Receive(const
SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count)
323 {
324  SPI_MASTER_STATUS_t status;
325
326  status = SPI_MASTER_STATUS_FAILURE;
327
328 #if (SPI_MASTER_INTERRUPT_RECEIVE_MODE == 1U)
329  if (handle->config->receive_mode ==
SPI_MASTER_TRANSFER_MODE_INTERRUPT)
330  {
331  status = SPI_MASTER_StartReceiveIRQ(handle, dataptr, count);
332  }
333 #endif
```

```
334
335 #if (SPI_MASTER_DMA_RECEIVE_MODE == 1U)
336  if (handle->config->receive_mode ==
SPI_MASTER_TRANSFER_MODE_DMA)
337  {
338  status = SPI_MASTER_StartReceiveDMA(handle, dataptr, count);
339  }
340 #endif
341
342 #if (SPI_MASTER_DIRECT_RECEIVE_MODE == 1U)
343  if (handle->config->receive_mode ==
SPI_MASTER_TRANSFER_MODE_DIRECT)
344  {
345  status = SPI_MASTER_lStartReceivePolling(handle, dataptr,
count);
346  }
347 #endif
348
349  return status;
350 }
351
352 #if (SPI_MASTER_INTERRUPT_TRANSMIT_MODE == 1U)
353 /*
354  * Transmit the number of data words specified.
355  */
356 SPI_MASTER_STATUS_t SPI_MASTER_StartTransmitIRQ(const
SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count)
357 {
358  SPI_MASTER_STATUS_t status;
359  uint32_t bytes_per_word = SPI_MASTER_1_BYTE_PER_WORD;
/* This is to support the word length 8 and 16.
360  Specify the number of bytes for the configured word length */
361  SPI_MASTER_RUNTIME_t * runtime_handle;
362
363  XMC_ASSERT("SPI_MASTER_StartTransmitIRQ:handle NULL" ,
(handle != NULL));
364
```

```
365  status = SPI_MASTER_STATUS_MODE_MISMATCH;
366  runtime_handle = handle->runtime;
367
368  if (handle->config->transmit_mode ==
SPI_MASTER_TRANSFER_MODE_INTERRUPT)
369  {
370  /* Check whether SPI channel is free or not */
371  if ((dataptr != NULL) && (count > 0U))
372  {
373  status = SPI_MASTER_STATUS_BUSY;
374  /*Check data pointer is valid or not*/
375  if (false == runtime_handle->tx_busy)
376  {
377  if (handle->runtime->word_length >
SPI_MASTER_WORD_LENGTH_8_BIT)
378  {
379  bytes_per_word = SPI_MASTER_2_BYTES_PER_WORD; /*
Word length is 16-bits */
380  }
381
382  /* Obtain the address of data, size of data */
383  runtime_handle->tx_data = dataptr;
384  runtime_handle->tx_data_count = (uint32_t)count <<
(bytes_per_word - 1U);
385  /* Initialize to first index and set the busy flag */
386  runtime_handle->tx_data_index = 0U;
387  runtime_handle->tx_busy = true;
388
389  /* Enable the transmit buffer event */
390  if ((uint32_t)handle->config->tx_fifo_size > 0U)
391  {
392  /* Flush the Transmit FIFO */
393  XMC_USIC_CH_TXFIFO_Flush(handle->channel);
394  XMC_USIC_CH_TXFIFO_EnableEvent(handle->channel,
(uint32_t)XMC_USIC_CH_TXFIFO_EVENT_CONF_STANDARD);
395  }
396  else
```

```
397  {
398  XMC_USIC_CH_EnableEvent(handle->channel,
(uint32_t)XMC_USIC_CH_EVENT_TRANSMIT_BUFFER);
399  }
400  XMC_SPI_CH_SetTransmitMode(handle->channel,
runtime_handle->spi_master_mode);
401  status = SPI_MASTER_STATUS_SUCCESS;
402
403  /* Trigger the transmit buffer interrupt */
404  XMC_USIC_CH_TriggerServiceRequest(handle->channel,
(uint32_t)handle->config->tx_sr);
405  }
406  }
407  else
408  {
409  status = SPI_MASTER_STATUS_BUFFER_INVALID;
410  }
411  }
412  return status;
413 }
414 #endif
415
416 #if(SPI_MASTER_DMA_TRANSMIT_MODE == 1U)
417 SPI_MASTER_STATUS_t SPI_MASTER_StartTransmitDMA(const
SPI_MASTER_t *const handle, uint8_t *data_ptr, uint32_t block_size)
418 {
419  SPI_MASTER_STATUS_t status;
420  SPI_MASTER_RUNTIME_t * runtime_handle;
421  uint32_t dma_ctll;
422  uint32_t mode;
423
424  XMC_ASSERT("SPI_MASTER_StartTransmitDMA:handle NULL"
, (handle != NULL));
425
426  status = SPI_MASTER_STATUS_MODE_MISMATCH;
427  runtime_handle = handle->runtime;
428
```

```
429  if (handle->config->transmit_mode ==
SPI_MASTER_TRANSFER_MODE_DMA)
430  {
431  /* Check whether SPI channel is free or not */
432  if (false == runtime_handle->tx_busy)
433  {
434  /* Check data pointer is valid or not */
435  if ((data_ptr != NULL) && (block_size > 0U) && (block_size <=
SPI_MASTER_DMA_MAXCOUNT))
436  {
437  /* Obtain the address of data, size of data */
438  runtime_handle->tx_data_count = block_size;
439  /* Initialize to first index and set the busy flag */
440  runtime_handle->tx_data_index = 0U;
441  runtime_handle->tx_busy = true;
442
443  if (runtime_handle->tx_data_dummy == true)
444  {
445  dma_ctll = (uint32_t)handle->global_dma->dma->CH[handle-
>dma_ch_tx_number].CTLL;
446
447  dma_ctll = (uint32_t)(dma_ctll & (uint32_t)(~
(GPDMA0_CH_CTLL_SINC_Msk))) |
448
((uint32_t)XMC_DMA_CH_ADDRESS_COUNT_MODE_NO_CHANGE
<< GPDMA0_CH_CTLL_SINC_Pos);
449
450  handle->global_dma->dma->CH[handle-
>dma_ch_tx_number].CTLL = dma_ctll;
451  mode = (uint32_t)((uint32_t)handle->runtime->spi_master_mode &
0xfffbU);
452  }
453  else
454  {
455  runtime_handle->tx_data = data_ptr;
456  dma_ctll = handle->global_dma->dma->CH[handle-
>dma_ch_tx_number].CTLL;
```

```
457
458  dma_ctll = (uint32_t)(dma_ctll &
(~GPDMA0_CH_CTLL_SINC_Msk)) |
459
((uint32_t)XMC_DMA_CH_ADDRESS_COUNT_MODE_INCREMENT
<< GPDMA0_CH_CTLL_SINC_Pos);
460
461  handle->global_dma->dma->CH[handle-
>dma_ch_tx_number].CTLL = dma_ctll;
462  mode = (uint32_t)handle->runtime->spi_master_mode;
463  }
464
465  /* Enable transmit event generation */
466  XMC_SPI_CH_EnableEvent(handle->channel,
(uint32_t)XMC_SPI_CH_EVENT_RECEIVE_START);
467
468  XMC_DMA_CH_SetBlockSize(handle->global_dma->dma,
handle->dma_ch_tx_number, block_size);
469
470  XMC_DMA_CH_SetSourceAddress(handle->global_dma->dma,
handle->dma_ch_tx_number, (uint32_t)runtime_handle->tx_data);
471
472  XMC_SPI_CH_SetTransmitMode(handle->channel,
runtime_handle->spi_master_mode);
473
474  XMC_DMA_CH_SetDestinationAddress(handle->global_dma-
>dma,
475  handle->dma_ch_tx_number,
476  (uint32_t)&(handle->channel->TBUF[mode]));
477
478  status = SPI_MASTER_STATUS_SUCCESS;
479
480  XMC_DMA_CH_Enable(handle->global_dma->dma, handle-
>dma_ch_tx_number);
481  }
482  else
483  {
```

```
484  status = SPI_MASTER_STATUS_BUFFER_INVALID;
485  }
486  }
487  else
488  {
489  status = SPI_MASTER_STATUS_BUSY;
490  }
491  }
492
493  return status;
494 }
495 #endif
496
497 #if (SPI_MASTER_DIRECT_TRANSMIT_MODE == 1U)
498 SPI_MASTER_STATUS_t
SPI_MASTER_lStartTransmitPolling(const SPI_MASTER_t *const
handle, uint8_t* dataptr, uint32_t count)
499 {
500  SPI_MASTER_STATUS_t status;
501  uint16_t data;
502  uint32_t bytes_per_word =
SPI_MASTER_1_BYTE_PER_WORD;; /* This is to support the word
length 8 and 16.
503  Specify the number of bytes for the configured word length */
504  SPI_MASTER_RUNTIME_t * runtime_handle;
505
506  status = SPI_MASTER_STATUS_BUSY;
507  runtime_handle = handle->runtime;
508  data = 0U;
509
510  XMC_ASSERT("SPI_MASTER_lStartTransmitPolling:handle
NULL" , (handle != NULL));
511
512  /* Check whether SPI channel is free or not */
513  if ((dataptr != NULL) && (count > 0U))
514  {
515  /* Check data pointer is valid or not */
```

```
516  if (false == runtime_handle->tx_busy)
517  {
518  if (handle->runtime->word_length >
SPI_MASTER_WORD_LENGTH_8_BIT)
519  {
520  bytes_per_word = SPI_MASTER_2_BYTES_PER_WORD; /*
Word length is 16-bits */
521  }
522
523  runtime_handle->tx_busy = true;
524  /* Obtain the address of data, size of data */
525  runtime_handle->tx_data = dataptr;
526  runtime_handle->tx_data_count = (uint32_t)count <<
(bytes_per_word - 1U);
527  /* Initialize to first index and set the busy flag */
528  runtime_handle->tx_data_index = 0U;
529
530  XMC_SPI_CH_SetTransmitMode(handle->channel,
runtime_handle->spi_master_mode);
531
532  if ((uint32_t)handle->config->tx_fifo_size > 0U)
533  {
534  /* Flush the Transmit FIFO */
535  XMC_USIC_CH_TXFIFO_Flush(handle->channel);
536
537  while (runtime_handle->tx_data_index < runtime_handle-
>tx_data_count)
538  {
539  while (XMC_USIC_CH_TXFIFO_IsFull(handle->channel) == true)
540  {
541  /* Wait until FIFO is having space for next entry */
542  }
543  if (runtime_handle->tx_data_dummy == true)
544  {
545  XMC_USIC_CH_TXFIFO_PutDataHPCMode(handle->channel,
0xFFFFU, (uint32_t)runtime_handle->spi_master_mode);
546  }
```

```
547  else
548  {
549  if(bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
550  {
551  data = *((uint16_t*)&runtime_handle->tx_data[runtime_handle-
>tx_data_index]);
552  }
553  else
554  {
555  data = runtime_handle->tx_data[runtime_handle->tx_data_index];
556  }
557  XMC_USIC_CH_TXFIFO_PutDataHPCMode(handle->channel,
data, (uint32_t)runtime_handle->spi_master_mode);
558  }
559  (runtime_handle->tx_data_index)+= bytes_per_word;
560  }
561  }
562  else
563  {
564  do
565  {
566  while((uint32_t)XMC_USIC_CH_GetTransmitBufferStatus(handle-
>channel) == (uint32_t)XMC_USIC_CH_TBUF_STATUS_BUSY)
567  {
568  }
569
570  if (runtime_handle->tx_data_dummy == true)
571  {
572  XMC_USIC_CH_WriteToTBUFTCI(handle->channel, 0xFFFFU,
runtime_handle->spi_master_mode);
573  }
574  else
575  {
576  if(bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
577  {
578  data = *((uint16_t*)&runtime_handle->tx_data[runtime_handle-
>tx_data_index]);
```

```
579  }
580  else
581  {
582  data = runtime_handle->tx_data[runtime_handle->tx_data_index];
583  }
584  XMC_USIC_CH_WriteToTBUFTCI(handle->channel, data,
runtime_handle->spi_master_mode);
585  }
586  (runtime_handle->tx_data_index)+=bytes_per_word;
587
588  while ((XMC_SPI_CH_GetStatusFlag(handle->channel) &
(uint32_t)XMC_SPI_CH_STATUS_FLAG_RECEIVER_START_INDICAT
== 0U)
589  {
590
591  }
592  XMC_SPI_CH_ClearStatusFlag(handle->channel,
(uint32_t)XMC_SPI_CH_STATUS_FLAG_RECEIVER_START_INDICAT
593  } while(runtime_handle->tx_data_index < runtime_handle-
>tx_data_count);
594  }
595
596  while((uint32_t)XMC_USIC_CH_GetTransmitBufferStatus(handle-
>channel) == (uint32_t)XMC_USIC_CH_TBUF_STATUS_BUSY)
597  {
598  }
599
600  runtime_handle->tx_busy = false;
601  runtime_handle->tx_data_count = 0U;
602  runtime_handle->tx_data_index = 0U;
603  status = SPI_MASTER_STATUS_SUCCESS;
604  }
605  }
606  else
607  {
608  status = SPI_MASTER_STATUS_BUFFER_INVALID;
609  }
```

```
610  runtime_handle->rx_data_dummy = true;
611  return status;
612 }
613 #endif
614
615 #if (SPI_MASTER_DIRECT_RECEIVE_MODE == 1U)
616
617 SPI_MASTER_STATUS_t
SPI_MASTER_lStartReceivePolling(const SPI_MASTER_t *const
handle, uint8_t* dataptr, uint32_t count)
618 {
619  SPI_MASTER_STATUS_t status;
620
621  SPI_MASTER_RUNTIME_t * runtime_handle;
622  static uint8_t dummy_data[2] = {0xFFU, 0xFFU};
623
624  XMC_ASSERT("SPI_MASTER_lStartReceivePolling:handle
NULL" , (handle != NULL));
625
626  status = SPI_MASTER_STATUS_BUSY;
627  runtime_handle = handle->runtime;
628
629  if ((dataptr != NULL) && (count > 0U))
630  {
631  /*Check data pointer is valid or not*/
632  if ((false == runtime_handle->rx_busy) && (false ==
runtime_handle->tx_busy))
633  {
634  runtime_handle->rx_busy = true;
635  runtime_handle->rx_data = dataptr;
636  runtime_handle->tx_data = &dummy_data[0];
637  runtime_handle->tx_data_dummy = true;
638  runtime_handle->rx_data_dummy = false;
639
640  status = SPI_MASTER_lReceivePolling(handle, count);
641
642  runtime_handle->tx_data_dummy = false;
```

```
643  runtime_handle->rx_busy = false;
644  }
645  }
646  else
647  {
648  status = SPI_MASTER_STATUS_BUFFER_INVALID;
649  }
650  return status;
651 }
652
653
654 #endif
655
656 #if (SPI_MASTER_INTERRUPT_RECEIVE_MODE == 1U)
657 /*
658  * Receive the specified the number of data words.
659  */
660 SPI_MASTER_STATUS_t SPI_MASTER_StartReceiveIRQ(const
SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count)
661 {
662  SPI_MASTER_STATUS_t status;
663  SPI_MASTER_RUNTIME_t * runtime_handle;
664  static uint8_t dummy_data[2] = {0xFFU, 0xFFU};
665
666  XMC_ASSERT("SPI_MASTER_StartReceiveIRQ:handle NULL" ,
(handle != NULL));
667
668  status = SPI_MASTER_STATUS_MODE_MISMATCH;
669  runtime_handle = handle->runtime;
670
671  if (handle->config->receive_mode ==
SPI_MASTER_TRANSFER_MODE_INTERRUPT)
672  {
673  status = SPI_MASTER_STATUS_BUSY;
674  /* Check whether SPI channel is free or not */
675  if ((dataptr != NULL) && (count > 0U))
676  {
```

```
677  /*Check data pointer is valid or not*/
678  if ((false == runtime_handle->rx_busy) && (false ==
runtime_handle->tx_busy))
679  {
680  runtime_handle->rx_busy = true;
681  runtime_handle->rx_data = dataptr;
682  runtime_handle->tx_data = &dummy_data[0];
683  runtime_handle->tx_data_count = count;
684  runtime_handle->tx_data_dummy = true;
685  runtime_handle->rx_data_dummy = false;
686
687  status = SPI_MASTER_lReceiveIRQ(handle, count);
688
689  }
690  }
691  else
692  {
693  status = SPI_MASTER_STATUS_BUFFER_INVALID;
694  }
695  }
696  return status;
697 }
698 #endif
699
700 #if(SPI_MASTER_DMA_RECEIVE_MODE == 1U)
701 SPI_MASTER_STATUS_t SPI_MASTER_StartReceiveDMA(const
SPI_MASTER_t *const handle, uint8_t *dataptr, uint32_t block_size)
702 {
703  SPI_MASTER_STATUS_t status;
704  SPI_MASTER_RUNTIME_t * runtime_handle;
705  static uint8_t dummy_data[2] = {0xFFU, 0xFFU};
706
707  XMC_ASSERT("SPI_MASTER_StartReceiveDMA:handle NULL" ,
(handle != NULL));
708
709  status = SPI_MASTER_STATUS_MODE_MISMATCH;
710  runtime_handle = handle->runtime;
```

```
711
712  if (handle->config->receive_mode ==
SPI_MASTER_TRANSFER_MODE_DMA)
713  {
714  status = SPI_MASTER_STATUS_BUSY;
715  /* Check whether SPI channel is free or not */
716  if ((false == runtime_handle->rx_busy) && (false ==
runtime_handle->tx_busy))
717  {
718  /* Check data pointer is valid or not */
719  if ((dataptr != NULL) && (block_size > 0U) && (block_size <=
SPI_MASTER_DMA_MAXCOUNT))
720  {
721  runtime_handle->rx_busy = true;
722  runtime_handle->rx_data = dataptr;
723  runtime_handle->tx_data = &dummy_data[0];
724  runtime_handle->tx_data_count = block_size;
725  runtime_handle->tx_data_dummy = true;
726  runtime_handle->rx_data_dummy = false;
727
728  status = SPI_MASTER_lReceiveDMA(handle, block_size);
729  }
730  else
731  {
732  status = SPI_MASTER_STATUS_BUFFER_INVALID;
733  }
734  }
735  else
736  {
737  status = SPI_MASTER_STATUS_BUSY;
738  }
739  }
740  return status;
741 }
742 #endif
743
744 /*
```

```
745  * Transmit and receive the data at the same time. This is
supported for full duplex mode only.
746  */
747 SPI_MASTER_STATUS_t SPI_MASTER_Transfer(const
SPI_MASTER_t *const handle,
748  uint8_t* tx_dataptr,
749  uint8_t* rx_dataptr,
750  uint32_t count)
751 {
752  SPI_MASTER_STATUS_t status;
753  SPI_MASTER_RUNTIME_t * runtime_handle;
754
755  XMC_ASSERT("SPI_MASTER_Transfer:handle NULL" , (handle
!= NULL));
756
757  status = SPI_MASTER_STATUS_BUSY;
758  runtime_handle = handle->runtime;
759
760  if (XMC_SPI_CH_MODE_STANDARD == runtime_handle-
>spi_master_mode)
761  {
762  /* Check whether SPI channel is free or not */
763  if ((tx_dataptr != NULL) && (rx_dataptr != NULL) && (count > 0U))
764  {
765  /*Check data pointer is valid or not*/
766  if ((false == runtime_handle->rx_busy) && (false ==
runtime_handle->tx_busy))
767  {
768  runtime_handle->rx_busy = true;
769  runtime_handle->rx_data = rx_dataptr;
770  runtime_handle->tx_data = tx_dataptr;
771  runtime_handle->tx_data_count = count;
772  runtime_handle->tx_data_dummy = false;
773  runtime_handle->rx_data_dummy = false;
774
775 #if (SPI_MASTER_INTERRUPT_RECEIVE_MODE == 1U)
776  if (handle->config->receive_mode ==
```

```
SPI_MASTER_TRANSFER_MODE_INTERRUPT)
777  {
778  status = SPI_MASTER_lReceiveIRQ(handle, count);
779  }
780 #endif
781 #if (SPI_MASTER_DMA_RECEIVE_MODE == 1U)
782  if (handle->config->receive_mode ==
SPI_MASTER_TRANSFER_MODE_DMA)
783  {
784  status = SPI_MASTER_lReceiveDMA(handle, count);
785  }
786 #endif
787 #if (SPI_MASTER_DIRECT_RECEIVE_MODE == 1U)
788  if (handle->config->receive_mode ==
SPI_MASTER_TRANSFER_MODE_DIRECT)
789  {
790  status = SPI_MASTER_lReceivePolling(handle, count);
791  runtime_handle->rx_busy = false;
792  }
793 #endif
794  }
795  }
796  else
797  {
798  status = SPI_MASTER_STATUS_BUFFER_INVALID;
799  }
800  }
801  else
802  {
803  status = SPI_MASTER_STATUS_FAILURE;
804  }
805
806  return status;
807 }
808
809
810 /*
```

```
811  * Aborts the ongoing data reception.
812  */
813 SPI_MASTER_STATUS_t SPI_MASTER_AbortReceive(const
SPI_MASTER_t *const handle)
814 {
815  SPI_MASTER_STATUS_t status;
816
817  status = SPI_MASTER_STATUS_FAILURE;
818
819  if ((handle->config->receive_mode !=
SPI_MASTER_TRANSFER_MODE_DIRECT) && (handle->runtime-
>rx_busy))
820  {
821  /* Abort if any ongoing transmission w.r.t reception. */
822  status = SPI_MASTER_AbortTransmit(handle);
823
824  if (status == SPI_MASTER_STATUS_SUCCESS)
825  {
826  /* Reset the user buffer pointer to null */
827  handle->runtime->rx_busy = false;
828  handle->runtime->rx_data = NULL;
829  handle->runtime->tx_data_dummy = false;
830  /* Disable the receive interrupts */
831  if ((uint32_t)handle->config->rx_fifo_size > 0U)
832  {
833  XMC_USIC_CH_RXFIFO_DisableEvent(handle->channel,
(uint32_t)SPI_MASTER_FIFO_RECEIVE_EVENT);
834  }
835  else
836  {
837 #if (SPI_MASTER_DMA_RECEIVE_MODE == 1U)
838  if (handle->config->receive_mode ==
SPI_MASTER_TRANSFER_MODE_DMA)
839  {
840  /* Disable the receive event */
841  if(XMC_DMA_CH_IsEnabled(handle->global_dma->dma, handle-
>dma_ch_rx_number))
```

```
842  {
843  XMC_DMA_CH_Disable(handle->global_dma->dma, handle->dma_ch_rx_number);
844  while(XMC_DMA_CH_IsEnabled(handle->global_dma->dma, handle->dma_ch_rx_number)==true)
845  {
846  }
847  XMC_SPI_CH_DisableEvent(handle->channel,
848  (uint32_t) ((uint32_t)XMC_USIC_CH_EVENT_STANDARD_RECEIVE | (uint32_t)XMC_USIC_CH_EVENT_ALTERNATIVE_RECEIVE));
849  }
850  }
851  else
852 #endif
853  {
854  XMC_SPI_CH_DisableEvent(handle->channel,
855  (uint32_t) ((uint32_t)XMC_USIC_CH_EVENT_STANDARD_RECEIVE | (uint32_t)XMC_USIC_CH_EVENT_ALTERNATIVE_RECEIVE));
856  }
857  }
858  status = SPI_MASTER_STATUS_SUCCESS;
859  }
860  else
861  {
862  status = SPI_MASTER_STATUS_FAILURE;
863  }
864  }
865  return status;
866 }
867
868 /*
869  * Aborts the ongoing data transmission.
870  */
871 SPI_MASTER_STATUS_t SPI_MASTER_AbortTransmit(const SPI_MASTER_t *const handle)
```

```
872 {
873  SPI_MASTER_STATUS_t status;
874
875  status = SPI_MASTER_STATUS_FAILURE;
876
877  if ((handle->config->transmit_mode !=
SPI_MASTER_TRANSFER_MODE_DIRECT) && (handle->runtime-
>tx_busy))
878  {
879  /*Reset the user buffer pointer to null*/
880  handle->runtime->tx_busy = false;
881  handle->runtime->tx_data = NULL;
882  handle->runtime->tx_data_dummy = false;
883  /*Disable the transmit interrupts*/
884  if ((uint32_t)handle->config->tx_fifo_size > 0U)
885  {
886  /*Disable the transmit FIFO event*/
887  XMC_USIC_CH_TXFIFO_DisableEvent(handle->channel,
(uint32_t)XMC_USIC_CH_TXFIFO_EVENT_CONF_STANDARD);
888  XMC_USIC_CH_TXFIFO_Flush(handle->channel);
889  }
890  else
891  {
892 #if (SPI_MASTER_DMA_TRANSMIT_MODE == 1U)
893  if(handle->config->transmit_mode ==
SPI_MASTER_TRANSFER_MODE_DMA)
894  {
895  /*Disable the standard transmit event*/
896  if(XMC_DMA_CH_IsEnabled(handle->global_dma->dma, handle-
>dma_ch_tx_number))
897  {
898  XMC_DMA_CH_Disable(handle->global_dma->dma, handle-
>dma_ch_tx_number);
899  while(XMC_DMA_CH_IsEnabled(handle->global_dma->dma,
handle->dma_ch_tx_number)==true)
900  {
901  }
```

```
902  XMC_SPI_CH_DisableEvent(handle->channel,
(uint32_t)XMC_USIC_CH_EVENT_TRANSMIT_BUFFER);
903  }
904  }
905  else
906 #endif
907  {
908  /*Disable the standard transmit event*/
909  XMC_SPI_CH_DisableEvent(handle->channel,
(uint32_t)XMC_USIC_CH_EVENT_TRANSMIT_BUFFER);
910  }
911  }
912  status = SPI_MASTER_STATUS_SUCCESS;
913  }
914  return status;
915 }
916 /*********************************************************************************
917 ** Private API definitions **
918 *********************************************************************************
919 #if(SPI_MASTER_INTERRUPT_TRANSMIT_MODE == 1U)
920 /*
921  * Transmit interrupt handler for the APP.
922  * This is a common interrupt handling function called for different
instances of the APP.
923  *
924  */
925 void SPI_MASTER_lTransmitHandler(const SPI_MASTER_t *
const handle)
926 {
927  uint16_t data; /* Data to be loaded into the TBUF */
928  uint32_t bytes_per_word = SPI_MASTER_1_BYTE_PER_WORD;
/* This is to support the word length 8 and 16.*/
929  SPI_MASTER_RUNTIME_t * runtime_handle = handle->runtime;
930
931  if (handle->runtime->word_length >
SPI_MASTER_WORD_LENGTH_8_BIT)
932  {
```

```
933  bytes_per_word = SPI_MASTER_2_BYTES_PER_WORD; /*
Word length is 16-bits */
934  }
935
936  if (runtime_handle->tx_data_index < runtime_handle-
>tx_data_count)
937  {
938  data = 0U;
939  /*When Transmit FIFO is enabled*/
940  if ((uint32_t)handle->config->tx_fifo_size > 0U)
941  {
942  /*Fill the transmit FIFO */
943  while (XMC_USIC_CH_TXFIFO_IsFull(handle->channel) == false)
944  {
945  if (runtime_handle->tx_data_index < runtime_handle-
>tx_data_count)
946  {
947  /*Load the FIFO byte by byte till either FIFO is full or all data is
loaded*/
948  if (runtime_handle->tx_data_dummy == true)
949  {
950  XMC_USIC_CH_TXFIFO_PutDataHPCMode(handle->channel,
0xFFFFU, (uint32_t)runtime_handle->spi_master_mode);
951  }
952  else
953  {
954  if(bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
955  {
956  data = *((uint16_t*)&runtime_handle->tx_data[runtime_handle-
>tx_data_index]);
957  }
958  else
959  {
960  data = runtime_handle->tx_data[runtime_handle->tx_data_index];
961  }
962  XMC_USIC_CH_TXFIFO_PutDataHPCMode(handle->channel,
data, (uint32_t)runtime_handle->spi_master_mode);
```

```
963  }
964  (runtime_handle->tx_data_index)+= bytes_per_word;
965  }
966  else
967  {
968  break;
969  }
970  }
971  }
972  else/*When Transmit FIFO is disabled*/
973  {
974  if (runtime_handle->tx_data_dummy == true)
975  {
976  XMC_USIC_CH_WriteToTBUFTCI(handle->channel, 0xFFFFU,
(uint32_t)runtime_handle->spi_master_mode);
977  }
978  else
979  {
980  if(bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
981  {
982  data = *((uint16_t*)&runtime_handle->tx_data[runtime_handle-
>tx_data_index]);
983  }
984  else
985  {
986  data = runtime_handle->tx_data[runtime_handle->tx_data_index];
987  }
988  XMC_USIC_CH_WriteToTBUFTCI(handle->channel, data,
(uint32_t)runtime_handle->spi_master_mode);
989  }
990  (runtime_handle->tx_data_index)+= bytes_per_word;
991  }
992  }
993  else
994  {
995  if (XMC_USIC_CH_TXFIFO_IsEmpty(handle->channel) == true)
996  {
```

```
997  /* Clear the flag */
998  if ((uint32_t)handle->config->tx_fifo_size > 0U)
999  {
1000  /* Clear the transmit FIFO event */
1001  XMC_USIC_CH_TXFIFO_DisableEvent(handle->channel,
(uint32_t)XMC_USIC_CH_TXFIFO_EVENT_CONF_STANDARD);
1002  }
1003  else
1004  {
1005  /* Clear the standard transmit event */
1006  XMC_USIC_CH_DisableEvent(handle->channel,
(uint32_t)XMC_USIC_CH_EVENT_TRANSMIT_BUFFER);
1007  }
1008
1009  /* Wait for the transmit buffer to be free to ensure that all data is
transmitted */
1010  while (XMC_USIC_CH_GetTransmitBufferStatus(handle-
>channel) == XMC_USIC_CH_TBUF_STATUS_BUSY)
1011  {
1012
1013  }
1014
1015  /* All data is transmitted */
1016  runtime_handle->tx_busy = false;
1017  runtime_handle->tx_data = NULL;
1018
1019  if ((handle->config->tx_cbhandler != NULL) && (runtime_handle-
>rx_busy == false))
1020  {
1021  /* Execute the callback function provided in the SPI_MASTER
APP UI */
1022  handle->config->tx_cbhandler();
1023  }
1024  }
1025  }
1026 }
1027 #endif
```

```
1028
1029 #if (SPI_MASTER_INTERRUPT_RECEIVE_MODE == 1U)
1030
1031 SPI_MASTER_STATUS_t SPI_MASTER_lReceiveIRQ(const
SPI_MASTER_t *const handle, uint32_t count)
1032 {
1033
1034  SPI_MASTER_STATUS_t status;
1035  SPI_MASTER_RUNTIME_t * runtime_handle;
1036  uint32_t bytes_per_word =
SPI_MASTER_1_BYTE_PER_WORD;; /* This is to support the word
length 8 and 16.
1037  Specify the number of bytes for the configured word length*/
1038
1039  runtime_handle = handle->runtime;
1040  runtime_handle->rx_data_index = 0U;
1041
1042  if (handle->runtime->word_length >
SPI_MASTER_WORD_LENGTH_8_BIT)
1043  {
1044  bytes_per_word = SPI_MASTER_2_BYTES_PER_WORD; /*
Word length is 16-bits */
1045  }
1046
1047  /* If no active reception in progress, obtain the address of data
buffer and number of data bytes to be received */
1048  runtime_handle->rx_data_count = (uint32_t)count <<
(bytes_per_word - 1U);
1049
1050  /* Check if FIFO is enabled */
1051  if ((uint32_t)handle->config->rx_fifo_size > 0U)
1052  {
1053  /* Clear the receive FIFO */
1054  XMC_USIC_CH_RXFIFO_Flush(handle->channel);
1055  SPI_MASTER_lStdRBUFFlush(handle->channel);
1056
1057  /* Configure the FIFO trigger limit based on the required data
```

```
size */
1058  SPI_MASTER_lReconfigureRxFIFO(handle, runtime_handle->rx_data_count);
1059
1060  /* Enable the receive FIFO events */
1061  XMC_USIC_CH_RXFIFO_EnableEvent(handle->channel, (uint32_t)SPI_MASTER_FIFO_RECEIVE_EVENT);
1062  }
1063  else
1064  {
1065  /* Flush the RBUF0 and RBUF1 */
1066  SPI_MASTER_lStdRBUFFlush(handle->channel);
1067
1068  /* Enable the standard receive events */
1069  XMC_USIC_CH_EnableEvent(handle->channel, (uint32_t)SPI_MASTER_RECEIVE_EVENT);
1070  }
1071  /* Call the transmit, to receive the data synchronously */
1072  status = SPI_MASTER_Transmit(handle, runtime_handle->tx_data, runtime_handle->tx_data_count);
1073
1074  return status;
1075 }
1076
1077 /*
1078  * Receive interrupt handler for the APP.
1079  * This is a common interrupt handling function for different instances of the SPI_MASTER APP.
1080  */
1081 void SPI_MASTER_lReceiveHandler(const SPI_MASTER_t * const handle)
1082 {
1083  uint16_t data; /* Data to be loaded into the TBUF */
1084  uint32_t bytes_per_word = SPI_MASTER_1_BYTE_PER_WORD; /* This is to support the word length 8 and 16. */
1085  SPI_MASTER_RUNTIME_t * runtime_handle = handle->runtime;
```

```
1086
1087  data = 0U;
1088
1089  if (handle->runtime->word_length >
SPI_MASTER_WORD_LENGTH_8_BIT)
1090  {
1091  bytes_per_word = SPI_MASTER_2_BYTES_PER_WORD; /*
Word length is 16-bits */
1092  }
1093
1094  if ((uint32_t)handle->config->rx_fifo_size > 0U)
1095  {
1096  /* read the FIFO */
1097  SPI_MASTER_lFIFORead(handle, bytes_per_word);
1098  /* Reconfigure the RXFIFO trigger limit based on pending receive
bytes */
1099  SPI_MASTER_lReconfigureRxFIFO(handle, (uint32_t)
(runtime_handle->rx_data_count - runtime_handle->rx_data_index));
1100  }
1101  else
1102  {
1103  /* When RxFIFO is disabled */
1104  if ((XMC_USIC_CH_GetReceiveBufferStatus(handle->channel) &
(uint32_t)XMC_USIC_CH_RBUF_STATUS_DATA_VALID0) != 0U )
1105  {
1106  if (runtime_handle->rx_data_index < runtime_handle-
>rx_data_count)
1107  {
1108  data = XMC_SPI_CH_GetReceivedData(handle->channel);
1109
1110  runtime_handle->rx_data[runtime_handle->rx_data_index] =
(uint8_t)data;
1111
1112  if (bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
1113  {
1114  runtime_handle->rx_data[runtime_handle->rx_data_index + 1U] =
(uint8_t)((uint16_t)data >> 8);
```

```
1115  }
1116
1117  (runtime_handle->rx_data_index)+= bytes_per_word;
1118  }
1119  }
1120  if ((XMC_USIC_CH_GetReceiveBufferStatus(handle->channel) &
(uint32_t)XMC_USIC_CH_RBUF_STATUS_DATA_VALID1) != 0U)
1121  {
1122  if (runtime_handle->rx_data_index < runtime_handle-
>rx_data_count)
1123  {
1124  data = XMC_SPI_CH_GetReceivedData(handle->channel);
1125
1126  runtime_handle->rx_data[runtime_handle->rx_data_index] =
(uint8_t)data;
1127
1128  if (bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
1129  {
1130  runtime_handle->rx_data[runtime_handle->rx_data_index + 1U]
= (uint8_t)((uint16_t)data >> 8);
1131  }
1132
1133  (runtime_handle->rx_data_index)+= bytes_per_word;
1134  }
1135  }
1136
1137  if (runtime_handle->rx_data_index == runtime_handle-
>rx_data_count)
1138  {
1139  /* Disable both standard receive and alternative receive FIFO
events */
1140  if ((uint32_t)handle->config->rx_fifo_size > 0U)
1141  {
1142  /* Enable the receive FIFO events */
1143  XMC_USIC_CH_RXFIFO_DisableEvent(handle->channel,
(uint32_t)SPI_MASTER_FIFO_RECEIVE_EVENT);
1144  }
```

```
1145  else
1146  {
1147  XMC_SPI_CH_DisableEvent(handle->channel,
(uint32_t)SPI_MASTER_RECEIVE_EVENT);
1148  }
1149  /* Reception complete */
1150  runtime_handle->rx_busy = false;
1151  runtime_handle->tx_data_dummy = false;
1152  runtime_handle->rx_data_dummy = true;
1153  runtime_handle->rx_data = NULL;
1154
1155  if (handle->config->rx_cbhandler != NULL)
1156  {
1157  /* Execute the 'End of reception' callback function */
1158  handle->config->rx_cbhandler();
1159  }
1160  }
1161  }
1162 }
1163
1164 /*
1165  * Read the data from FIFO until it becomes empty.
1166  */
1167 void SPI_MASTER_lFIFORead(const SPI_MASTER_t * const
handle, const uint32_t bytes_per_word)
1168 {
1169  SPI_MASTER_RUNTIME_t * runtime_handle;
1170  uint16_t data;
1171
1172  runtime_handle = handle->runtime;
1173  data = 0U;
1174
1175  /* When Receive FIFO is enabled*/
1176  while (XMC_USIC_CH_RXFIFO_IsEmpty(handle->channel) ==
false)
1177  {
1178  if (runtime_handle->rx_data_index < runtime_handle-
```

```
>rx_data_count)
1179  {
1180  data = XMC_SPI_CH_GetReceivedData(handle->channel);
1181  runtime_handle->rx_data[runtime_handle->rx_data_index] =
(uint8_t)data;
1182
1183  if (bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
1184  {
1185  runtime_handle->rx_data[runtime_handle->rx_data_index + 1U]
= (uint8_t)((uint16_t)data >> 8);
1186  }
1187  (runtime_handle->rx_data_index)+= bytes_per_word;
1188  }
1189
1190  if (runtime_handle->rx_data_index == runtime_handle-
>rx_data_count)
1191  {
1192  /*Reception complete*/
1193  runtime_handle->rx_busy = false;
1194  runtime_handle->tx_data_dummy = false;
1195  /*Disable both standard receive and alternative receive FIFO
events*/
1196  XMC_USIC_CH_RXFIFO_DisableEvent(handle->channel,
(uint32_t)SPI_MASTER_FIFO_RECEIVE_EVENT);
1197  if (handle->config->rx_cbhandler != NULL)
1198  {
1199  /* Execute the 'End of reception' callback function */
1200  handle->config->rx_cbhandler();
1201  }
1202  break;
1203  }
1204  }
1205 }
1206
1207 /*
1208  * This function configures the FIFO settings
1209  */
```

```
1210 static void SPI_MASTER_lReconfigureRxFIFO(const
SPI_MASTER_t * const handle, uint32_t data_size)
1211 {
1212  uint32_t fifo_size;
1213  uint32_t ret_limit_val;
1214
1215  if (((uint32_t)handle->config->rx_fifo_size > 0U) && (data_size >
0U))
1216  {
1217  fifo_size = (uint32_t)0x01 << handle->config->rx_fifo_size;
1218
1219  if (handle->runtime->word_length >
SPI_MASTER_WORD_LENGTH_8_BIT)
1220  {
1221  /* Data size is divided by 2, to change the trigger limit according
the word length */
1222  data_size = (uint32_t)data_size >> 1U;
1223  }
1224
1225  /*If data size is more than FIFO size, configure the limit to the
FIFO size*/
1226  if (data_size < (fifo_size >> 1))
1227  {
1228  ret_limit_val = data_size - 1U;
1229  }
1230  else
1231  {
1232  ret_limit_val = fifo_size >> 1;
1233  }
1234
1235  /*Set the limit value*/
1236  XMC_USIC_CH_RXFIFO_SetSizeTriggerLimit(handle->channel,
handle->config->rx_fifo_size, ret_limit_val);
1237  }
1238 }
1239 #endif
1240
```

```
1241 #if (SPI_MASTER_DIRECT_RECEIVE_MODE == 1U)
1242 SPI_MASTER_STATUS_t SPI_MASTER_lReceivePolling(const
SPI_MASTER_t *const handle, uint32_t count)
1243 {
1244  SPI_MASTER_RUNTIME_t * runtime_handle;
1245  uint32_t bytes_per_word =
SPI_MASTER_1_BYTE_PER_WORD; /* This is to support the word
length 8 and 16.
1246  Specify the number of bytes for the configured word length */
1247  uint16_t data;
1248
1249  runtime_handle = handle->runtime;
1250  data = 0U;
1251  runtime_handle->rx_data_index = 0U;
1252  runtime_handle->tx_data_index = 0U;
1253
1254  if (handle->runtime->word_length >
SPI_MASTER_WORD_LENGTH_8_BIT)
1255  {
1256  bytes_per_word = SPI_MASTER_2_BYTES_PER_WORD; /*
Word length is 16-bits */
1257  }
1258
1259  runtime_handle->rx_data_count = (uint32_t)count <<
(bytes_per_word - 1U);
1260
1261  XMC_SPI_CH_SetTransmitMode(handle->channel,
runtime_handle->spi_master_mode);
1262
1263  /* Check if FIFO is enabled */
1264  if ((uint32_t)handle->config->rx_fifo_size > 0U)
1265  {
1266  /* Clear the receive FIFO */
1267  XMC_USIC_CH_RXFIFO_Flush(handle->channel);
1268  SPI_MASTER_lStdRBUFFlush(handle->channel);
1269
1270  if (runtime_handle->tx_data_dummy == true)
```

```
1271  {
1272  XMC_USIC_CH_TXFIFO_PutDataHPCMode(handle->channel,
0xFFFFU, (uint32_t)runtime_handle->spi_master_mode);
1273  }
1274  else
1275  {
1276  if(bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
1277  {
1278  data = *((uint16_t*)&runtime_handle->tx_data[runtime_handle-
>tx_data_index]);
1279  }
1280  else
1281  {
1282  data = runtime_handle->tx_data[runtime_handle-
>tx_data_index];
1283  }
1284  XMC_USIC_CH_TXFIFO_PutDataHPCMode(handle->channel,
data, (uint32_t)runtime_handle->spi_master_mode);
1285  }
1286
1287  (runtime_handle->tx_data_index)+= bytes_per_word;
1288
1289
1290  while (runtime_handle->tx_data_index < runtime_handle-
>rx_data_count)
1291  {
1292  if (runtime_handle->tx_data_dummy == true)
1293  {
1294  XMC_USIC_CH_TXFIFO_PutDataHPCMode(handle->channel,
0xFFFFU, (uint32_t)runtime_handle->spi_master_mode);
1295  }
1296  else
1297  {
1298  if(bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
1299  {
1300  data = *((uint16_t*)&runtime_handle->tx_data[runtime_handle-
>tx_data_index]);
```

```
1301  }
1302  else
1303  {
1304  data = runtime_handle->tx_data[runtime_handle-
>tx_data_index];
1305  }
1306  XMC_USIC_CH_TXFIFO_PutDataHPCMode(handle->channel,
data, (uint32_t)runtime_handle->spi_master_mode);
1307  }
1308
1309  while(XMC_USIC_CH_RXFIFO_IsEmpty(handle->channel) ==
true)
1310  {
1311
1312  }
1313
1314  data = XMC_SPI_CH_GetReceivedData(handle->channel);
1315
1316  runtime_handle->rx_data[runtime_handle->rx_data_index] =
(uint8_t)data;
1317
1318  if (bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
1319  {
1320  runtime_handle->rx_data[runtime_handle->rx_data_index + 1U]
= (uint8_t)((uint16_t)data >> 8);
1321  }
1322
1323  (runtime_handle->rx_data_index)+= bytes_per_word;
1324  (runtime_handle->tx_data_index)+= bytes_per_word;
1325  }
1326
1327  while(XMC_USIC_CH_RXFIFO_IsEmpty(handle->channel) ==
true)
1328  {
1329
1330  }
1331
```

```
1332  data = XMC_SPI_CH_GetReceivedData(handle->channel);
1333
1334  runtime_handle->rx_data[runtime_handle->rx_data_index] =
(uint8_t)data;
1335
1336  if (bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
1337  {
1338  runtime_handle->rx_data[runtime_handle->rx_data_index + 1U]
= (uint8_t)((uint16_t)data >> 8);
1339  }
1340
1341  XMC_USIC_CH_RXFIFO_ClearEvent(handle->channel,
SPI_MASTER_FIFO_RECEIVE_INDICATION_FLAG);
1342  }
1343  else
1344  {
1345  /* Flush the RBUF0 and RBUF1 */
1346  SPI_MASTER_lStdRBUFFlush(handle->channel);
1347
1348
while((uint32_t)XMC_USIC_CH_GetTransmitBufferStatus(handle-
>channel) == (uint32_t)XMC_USIC_CH_TBUF_STATUS_BUSY)
1349  {
1350  }
1351
1352  if (runtime_handle->tx_data_dummy == true)
1353  {
1354  XMC_USIC_CH_WriteToTBUFTCI(handle->channel, 0xFFFFU,
(uint32_t)runtime_handle->spi_master_mode);
1355  }
1356  else
1357  {
1358  if(bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
1359  {
1360  data = *((uint16_t*)&runtime_handle->tx_data[runtime_handle-
>tx_data_index]);
1361  }
```

```
1362  else
1363  {
1364  data = runtime_handle->tx_data[runtime_handle-
>tx_data_index];
1365  }
1366  XMC_USIC_CH_WriteToTBUFTCI(handle->channel, data,
(uint32_t)runtime_handle->spi_master_mode);
1367  }
1368
1369  (runtime_handle->tx_data_index)+= bytes_per_word;
1370
1371  while (runtime_handle->tx_data_index < runtime_handle-
>rx_data_count)
1372  {
1373
while((uint32_t)XMC_USIC_CH_GetTransmitBufferStatus(handle-
>channel) == (uint32_t)XMC_USIC_CH_TBUF_STATUS_BUSY)
1374  {
1375
1376  }
1377
1378  if (runtime_handle->tx_data_dummy == true)
1379  {
1380  XMC_USIC_CH_WriteToTBUFTCI(handle->channel, 0xFFFFU,
(uint32_t)runtime_handle->spi_master_mode);
1381  }
1382  else
1383  {
1384  if(bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
1385  {
1386  data = *((uint16_t*)&runtime_handle->tx_data[runtime_handle-
>tx_data_index]);
1387  }
1388  else
1389  {
1390  data = runtime_handle->tx_data[runtime_handle-
>tx_data_index];
```

```
1391  }
1392  XMC_USIC_CH_WriteToTBUFTCI(handle->channel, data,
(uint32_t)runtime_handle->spi_master_mode);
1393  }
1394
1395  while (XMC_USIC_CH_GetReceiveBufferStatus(handle-
>channel) == 0U)
1396  {
1397
1398  }
1399
1400  data = XMC_SPI_CH_GetReceivedData(handle->channel);
1401
1402  runtime_handle->rx_data[runtime_handle->rx_data_index] =
(uint8_t)data;
1403
1404  if (bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
1405  {
1406  runtime_handle->rx_data[runtime_handle->rx_data_index + 1U]
= (uint8_t)((uint16_t)data >> 8);
1407  }
1408
1409  (runtime_handle->rx_data_index)+= bytes_per_word;
1410  (runtime_handle->tx_data_index)+= bytes_per_word;
1411
1412  XMC_SPI_CH_ClearStatusFlag(handle->channel,
SPI_MASTER_RECEIVE_INDICATION_FLAG);
1413  }
1414
1415  while (XMC_USIC_CH_GetReceiveBufferStatus(handle-
>channel) == 0U)
1416  {
1417
1418  }
1419
1420  data = XMC_SPI_CH_GetReceivedData(handle->channel);
1421
```

```
1422  runtime_handle->rx_data[runtime_handle->rx_data_index] =
(uint8_t)data;
1423
1424  if (bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
1425  {
1426  runtime_handle->rx_data[runtime_handle->rx_data_index + 1U]
= (uint8_t)((uint16_t)data >> 8);
1427  }
1428
1429  XMC_SPI_CH_ClearStatusFlag(handle->channel,
SPI_MASTER_RECEIVE_INDICATION_FLAG);
1430  }
1431  runtime_handle->rx_data_count = 0U;
1432  runtime_handle->rx_data_index = 0U;
1433  runtime_handle->tx_data_index = 0U;
1434
1435  return SPI_MASTER_STATUS_SUCCESS;
1436 }
1437 #endif
1438
1439 #if (SPI_MASTER_DMA_RECEIVE_MODE == 1U)
1440 SPI_MASTER_STATUS_t SPI_MASTER_lReceiveDMA(const
SPI_MASTER_t *const handle, uint32_t block_size)
1441 {
1442  SPI_MASTER_STATUS_t status;
1443  SPI_MASTER_RUNTIME_t * runtime_handle;
1444
1445  runtime_handle = handle->runtime;
1446  runtime_handle->rx_data_index = 0U;
1447  runtime_handle->rx_data_count = (uint32_t)block_size;
1448
1449  SPI_MASTER_lStdRBUFFlush(handle->channel);
1450
1451  XMC_SPI_CH_EnableEvent(handle->channel,
(uint32_t)SPI_MASTER_RECEIVE_EVENT);
1452
1453  XMC_DMA_CH_SetBlockSize(handle->global_dma->dma,
```

```
handle->dma_ch_rx_number, runtime_handle->rx_data_count);
1454
1455  XMC_DMA_CH_SetSourceAddress(handle->global_dma->dma,
1456  handle->dma_ch_rx_number,
1457  (uint32_t)&(handle->channel->RBUF));
1458
1459  XMC_DMA_CH_SetDestinationAddress(handle->global_dma-
>dma, handle->dma_ch_rx_number, (uint32_t)runtime_handle-
>rx_data);
1460
1461  status = SPI_MASTER_STATUS_SUCCESS;
1462
1463  XMC_DMA_CH_Enable(handle->global_dma->dma, handle-
>dma_ch_rx_number);
1464
1465  /* Call the transmit, to receive the data synchronously */
1466  status = SPI_MASTER_Transmit(handle, runtime_handle-
>tx_data, runtime_handle->tx_data_count);
1467
1468  return status;
1469 }
1470 #endif
1471
1472 /*
1473  * Clears the receive buffers
1474  */
1475 static void SPI_MASTER_lStdRBUFFlush(XMC_USIC_CH_t
*const channel)
1476 {
1477  /* Clear RBF0 */
1478  (void)XMC_SPI_CH_GetReceivedData(channel);
1479  /* Clear RBF1 */
1480  (void)XMC_SPI_CH_GetReceivedData(channel);
1481 }
1482
1483 #if (SPI_MASTER_PARITY_ERROR == 1U)
1484 /*
```

```
1485  * Protocol interrupt handling function.
1486  * The function is common for different instances of the
SPI_MASTER APP.
1487  */
1488 void SPI_MASTER_lProtocolHandler(const SPI_MASTER_t *
const handle)
1489 {
1490  uint32_t psr_status;
1491
1492  psr_status = XMC_SPI_CH_GetStatusFlag(handle->channel);
1493
1494  /*Check for Parity detection error */
1495  if ((handle->config->parity_cbhandler != NULL) && \
1496  (psr_status &
(uint32_t)XMC_SPI_CH_STATUS_FLAG_PARITY_ERROR_EVENT_DE
1497  {
1498  handle->config->parity_cbhandler();
1499  }
1500 }
1501 #endif
1502
1503 /*
1504  * This is used to reconfigure the registers while changing the SPI
mode dynamically
1505  */
1506 static void SPI_MASTER_lPortConfig(const SPI_MASTER_t*
handle)
1507 {
1508  switch (handle->runtime->spi_master_mode)
1509  {
1510  case XMC_SPI_CH_MODE_STANDARD:
1511  /* Configure the data input line selected */
1512  XMC_SPI_CH_SetInputSource(handle->channel,
XMC_SPI_CH_INPUT_DIN0, (uint8_t)(handle->runtime->dx0_input));
1513  /* Configure the pin as input */
1514  XMC_GPIO_SetMode(handle->config->mosi_1_pin->port,
handle->config->mosi_1_pin->pin,
```

```
XMC_GPIO_MODE_INPUT_TRISTATE);
1515  /* Disable the HW control of the PINs */
1516  XMC_GPIO_SetHardwareControl(handle->config->mosi_0_pin-
>port,
1517  handle->config->mosi_0_pin->pin,
1518  XMC_GPIO_HWCTRL_DISABLED);
1519  XMC_GPIO_SetHardwareControl(handle->config->mosi_1_pin-
>port,
1520  handle->config->mosi_1_pin->pin,
1521  XMC_GPIO_HWCTRL_DISABLED);
1522
1523  break;
1524
1525  case XMC_SPI_CH_MODE_STANDARD_HALFDUPLEX:
1526  /* Configure the data input line selected */
1527  XMC_SPI_CH_SetInputSource(handle->channel,
XMC_SPI_CH_INPUT_DIN0, (uint8_t)(handle->runtime-
>dx0_input_half_duplex));
1528  /* Disable the HW control of the PINs */
1529  XMC_GPIO_SetHardwareControl(handle->config->mosi_0_pin-
>port,
1530  handle->config->mosi_0_pin->pin,
1531  XMC_GPIO_HWCTRL_DISABLED);
1532  break;
1533
1534  case XMC_SPI_CH_MODE_DUAL:
1535  case XMC_SPI_CH_MODE_QUAD:
1536  /* Configure the data input line for loopback mode */
1537  XMC_SPI_CH_SetInputSource(handle->channel,
XMC_SPI_CH_INPUT_DIN0, (uint8_t)SPI_MASTER_INPUT_G);
1538  /* Configure the pin as input */
1539  XMC_GPIO_SetMode(handle->config->mosi_1_pin->port,
1540  handle->config->mosi_1_pin->pin,
1541  handle->config->mosi_1_pin_config->port_config.mode);
1542
1543  /* Configure the Hardware control mode selected for the pin */
1544  XMC_GPIO_SetHardwareControl(handle->config->mosi_0_pin-
```

```
>port,
1545  handle->config->mosi_0_pin->pin,
1546  handle->config->mosi_0_pin_config->hw_control);
1547  XMC_GPIO_SetHardwareControl(handle->config->mosi_1_pin-
>port,
1548  handle->config->mosi_1_pin->pin,
1549  handle->config->mosi_1_pin_config->hw_control);
1550  break;
1551
1552  default:
1553  break;
1554  }
1555 }
1556
1557 /*
1558  * This is used to reassign the mode for ports after updating the
baud rate
1559  */
1560 static void SPI_MASTER_lPortModeSet(const SPI_MASTER_t*
handle)
1561 {
1562  uint32_t ss_line;
1563
1564  /* Configure the ports with actual mode */
1565  for (ss_line = 0U; ss_line < handle->config->slave_select_lines;
ss_line++)
1566  {
1567  XMC_GPIO_SetMode(handle->config-
>slave_select_pin[ss_line]->port,
1568  handle->config->slave_select_pin[ss_line]->pin,
1569  handle->config->slave_select_pin_config[ss_line]-
>port_config.mode);
1570  }
1571
1572  XMC_GPIO_SetMode(handle->config->sclk_out_pin->port,
1573  handle->config->sclk_out_pin->pin,
1574  handle->config->sclk_out_pin_config->port_config.mode);
```

```
1575
1576  switch (handle->runtime->spi_master_mode)
1577  {
1578  case XMC_SPI_CH_MODE_STANDARD:
1579  case XMC_SPI_CH_MODE_STANDARD_HALFDUPLEX:
1580  XMC_GPIO_SetMode(handle->config->mosi_0_pin->port,
1581  handle->config->mosi_0_pin->pin,
1582  handle->config->mosi_0_pin_config->port_config.mode);
1583  break;
1584
1585  case XMC_SPI_CH_MODE_DUAL:
1586  XMC_GPIO_SetMode(handle->config->mosi_0_pin->port,
1587  handle->config->mosi_0_pin->pin,
1588  handle->config->mosi_0_pin_config->port_config.mode);
1589  XMC_GPIO_SetMode(handle->config->mosi_1_pin->port,
1590  handle->config->mosi_1_pin->pin,
1591  handle->config->mosi_1_pin_config->port_config.mode);
1592  break;
1593
1594  case XMC_SPI_CH_MODE_QUAD:
1595  XMC_GPIO_SetMode(handle->config->mosi_0_pin->port,
1596  handle->config->mosi_0_pin->pin,
1597  handle->config->mosi_0_pin_config->port_config.mode);
1598  XMC_GPIO_SetMode(handle->config->mosi_1_pin->port,
1599  handle->config->mosi_1_pin->pin,
1600  handle->config->mosi_1_pin_config->port_config.mode);
1601  XMC_GPIO_SetMode(handle->config->mosi_2_pin->port,
1602  handle->config->mosi_2_pin->pin,
1603  handle->config->mosi_2_pin_config->port_config.mode);
1604  XMC_GPIO_SetMode(handle->config->mosi_3_pin->port,
1605  handle->config->mosi_3_pin->pin,
1606  handle->config->mosi_3_pin_config->port_config.mode);
1607  break;
1608
1609  default:
1610  break;
1611  }
```

```
1612 }
1613
1614 /*
1615  * This is used to make the ports as input during update of the
baud rate, to avoid the noise in output ports
1616  */
1617 static void SPI_MASTER_lPortModeReset(const
SPI_MASTER_t* handle)
1618 {
1619  uint32_t ss_line;
1620
1621  /* Configure the ports as input */
1622  for (ss_line = 0U; ss_line < handle->config->slave_select_lines;
ss_line++)
1623  {
1624  XMC_GPIO_SetMode(handle->config-
>slave_select_pin[ss_line]->port,
1625  handle->config->slave_select_pin[ss_line]->pin,
1626  XMC_GPIO_MODE_INPUT_TRISTATE);
1627  }
1628
1629  XMC_GPIO_SetMode(handle->config->sclk_out_pin->port,
handle->config->sclk_out_pin->pin,
XMC_GPIO_MODE_INPUT_TRISTATE);
1630
1631  switch (handle->runtime->spi_master_mode)
1632  {
1633  case XMC_SPI_CH_MODE_STANDARD:
1634  case XMC_SPI_CH_MODE_STANDARD_HALFDUPLEX:
1635  XMC_GPIO_SetMode(handle->config->mosi_0_pin->port,
handle->config->mosi_0_pin->pin,
XMC_GPIO_MODE_INPUT_TRISTATE);
1636  break;
1637
1638  case XMC_SPI_CH_MODE_DUAL:
1639  XMC_GPIO_SetMode(handle->config->mosi_0_pin->port,
handle->config->mosi_0_pin->pin,
```

```
XMC_GPIO_MODE_INPUT_TRISTATE);
1640  XMC_GPIO_SetMode(handle->config->mosi_1_pin->port,
handle->config->mosi_1_pin->pin,
XMC_GPIO_MODE_INPUT_TRISTATE);
1641  break;
1642
1643  case XMC_SPI_CH_MODE_QUAD:
1644  XMC_GPIO_SetMode(handle->config->mosi_0_pin->port,
handle->config->mosi_0_pin->pin,
XMC_GPIO_MODE_INPUT_TRISTATE);
1645  XMC_GPIO_SetMode(handle->config->mosi_1_pin->port,
handle->config->mosi_1_pin->pin,
XMC_GPIO_MODE_INPUT_TRISTATE);
1646  XMC_GPIO_SetMode(handle->config->mosi_2_pin->port,
handle->config->mosi_2_pin->pin,
XMC_GPIO_MODE_INPUT_TRISTATE);
1647  XMC_GPIO_SetMode(handle->config->mosi_3_pin->port,
handle->config->mosi_3_pin->pin,
XMC_GPIO_MODE_INPUT_TRISTATE);
1648  break;
1649
1650  default:
1651  break;
1652  }
1653 }
1654
1655 /*
1656  * This is used check whether the mode change is valid or not
1657  */
1658 static SPI_MASTER_STATUS_t
SPI_MASTER_lValidateModeChange(const SPI_MASTER_t * handle,
XMC_SPI_CH_MODE_t mode)
1659 {
1660  SPI_MASTER_STATUS_t status;
1661
1662  status = SPI_MASTER_STATUS_SUCCESS;
1663
```

```
1664  if ((handle->config->spi_master_config_mode ==
XMC_SPI_CH_MODE_STANDARD_HALFDUPLEX) ||
1665  (handle->config->spi_master_config_mode < mode))
1666  {
1667  status = SPI_MASTER_STATUS_FAILURE;
1668  }
1669  else if (handle->config->spi_master_config_mode ==
XMC_SPI_CH_MODE_STANDARD)
1670  {
1671  if (XMC_SPI_CH_MODE_DUAL <= mode)
1672  {
1673  status = SPI_MASTER_STATUS_FAILURE;
1674  }
1675  }
1676  else
1677  {
1678  if ((mode == XMC_SPI_CH_MODE_STANDARD) && (handle-
>runtime->dx0_input == SPI_MASTER_INPUT_INVALID))
1679  {
1680  status = SPI_MASTER_STATUS_FAILURE;
1681  }
1682
1683  else if ((mode ==
XMC_SPI_CH_MODE_STANDARD_HALFDUPLEX) && (handle-
>runtime->dx0_input_half_duplex == SPI_MASTER_INPUT_INVALID))
1684  {
1685  status = SPI_MASTER_STATUS_FAILURE;
1686  }
1687  else
1688  {
1689  /* added to abide MISRA */
1690  }
1691  }
1692  return status;
1693 }
```

SPI_MASTER

Home

Enumerations

enum **SPI_MASTER_STATUS** {
SPI_MASTER_STATUS_
SPI_MASTER_STATUS_F
SPI_MASTER_STATUS_B
SPI_MASTER_STATUS_B
SPI_MASTER_STATUS_
}
Return status of the **SPI_M**

enum **SPI_MASTER_SR_ID** {
SPI_MASTER_SR_ID_0 =
SPI_MASTER_SR_ID_1,
SPI_MASTER_SR_ID_2,
SPI_MASTER_SR_ID_3,
SPI_MASTER_SR_ID_4,
SPI_MASTER_SR_ID_5
}
Service ID for Transmit, Re
events. More...

enum **SPI_MASTER_SS_SIGNA**
SPI_MASTER_SS_SIGN
SPI_MASTER_SS_SIGNA
SPI_MASTER_SS_SIGNA
SPI_MASTER_SS_SIGNA
SPI_MASTER_SS_SIGN
SPI_MASTER_SS_SIGNA
SPI_MASTER_SS_SIGNA
SPI_MASTER_SS_SIGNA

	} Slave select signals. More..
enum	**SPI_MASTER_INPUT** { **SPI_MASTER_INPUT_A** **SPI_MASTER_INPUT_B**, **SPI_MASTER_INPUT_C**, **SPI_MASTER_INPUT_D**, **SPI_MASTER_INPUT_E**, **SPI_MASTER_INPUT_F**, **SPI_MASTER_INPUT_G**, **SPI_MASTER_INPUT_INV** } Enum type which defines R More...
enum	**SPI_MASTER_TRANSFEF** **SPI_MASTER_TRANSFEF** **SPI_MASTER_TRANSFEF** **SPI_MASTER_TRANSFEF** Enum used to identify the tr either transmit or receive fu
typedef enum **SPI_MASTER_STATUS**	**SPI_MASTER_STATUS_t** Return status of the **SPI_M**
typedef enum **SPI_MASTER_SR_ID**	**SPI_MASTER_SR_ID_t** Service ID for Transmit, Re events.
typedef enum **SPI_MASTER_SS_SIGNAL**	**SPI_MASTER_SS_SIGNA** Slave select signals.
typedef enum **SPI_MASTER_INPUT**	**SPI_MASTER_INPUT_t** Enum type which defines R

typedef enum

SPI_MASTER_TRANSFER_MODE **SPI_MASTER_TRANSFEF**

Enum used to identify the tr

either transmit or receive fu

Detailed Description

Enumeration Type Documentation

enum SPI_MASTER_INPUT

Enum type which defines Receive input list.

Enumerator	
SPI_MASTER_INPUT_A	Input-A
SPI_MASTER_INPUT_B	Input-B
SPI_MASTER_INPUT_C	Input-C
SPI_MASTER_INPUT_D	Input-D
SPI_MASTER_INPUT_E	Input-E
SPI_MASTER_INPUT_F	Input-F
SPI_MASTER_INPUT_G	Input-G
SPI_MASTER_INPUT_INVALID	This is to check during mode switch

Definition at line **169** of file **SPI_MASTER.h**.

enum SPI_MASTER_SR_ID

Service ID for Transmit, Receive and Parity events.

Enumerator	
SPI_MASTER_SR_ID_0	SR-0
SPI_MASTER_SR_ID_1	SR-1
SPI_MASTER_SR_ID_2	SR-2
SPI_MASTER_SR_ID_3	SR-3
SPI_MASTER_SR_ID_4	SR-4
SPI_MASTER_SR_ID_5	SR-5

Definition at line **141** of file **SPI_MASTER.h**.

enum SPI_MASTER_SS_SIGNAL

Slave select signals.

Enumerator	
SPI_MASTER_SS_SIGNAL_0	Slave select 0
SPI_MASTER_SS_SIGNAL_1	Slave select 1

SPI_MASTER_SS_SIGNAL_2	Slave select 2
SPI_MASTER_SS_SIGNAL_3	Slave select 3
SPI_MASTER_SS_SIGNAL_4	Slave select 4
SPI_MASTER_SS_SIGNAL_5	Slave select 5
SPI_MASTER_SS_SIGNAL_6	Slave select 6
SPI_MASTER_SS_SIGNAL_7	Slave select 7

Definition at line **154** of file **SPI_MASTER.h**.

enum SPI_MASTER_STATUS

Return status of the **SPI_MASTER** APP.

Enumerator	
SPI_MASTER_STATUS_SUCCESS	Status success
SPI_MASTER_STATUS_FAILURE	Status failure
SPI_MASTER_STATUS_BUSY	Busy state
SPI_MASTER_STATUS_BUFFER_INVALID	If input buffer and lengt

SPI_MASTER_STATUS_MODE_MISMATCH	API invoked by a handl configured with differen e.g, If SPI_MASTER_StartTra is invoked for an instan has transmit mode conf "Interrupt", will return th

Definition at line **126** of file **SPI_MASTER.h**.

enum SPI_MASTER_TRANSFER_MODE

Enum used to identify the transfer type used for either transmit or receive function.

Enumerator	
SPI_MASTER_TRANSFER_MODE_INTERRUPT	Implement data transmit or receive using interrupts
SPI_MASTER_TRANSFER_MODE_DMA	Implement data transmit or receive using DMA
SPI_MASTER_TRANSFER_MODE_DIRECT	This configuration exposes signals for external

	APP connection

Definition at line **184** of file **SPI_MASTER.h**.

SPI_MASTER

Code Directory Reference

Files

file **SPI_MASTER.c** [code]

file **SPI_MASTER.h** [code]

SPI_MASTER

Home

Methods

DAVE_APP_VERSION_t	**SPI_MASTER_GetAppVersion** (void) Get **SPI_MASTER** APP version. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_Init** (**SPI_MASTER_t** *const handle) Initialize the SPI channel as per the configuration made in GUI. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_SetMode** (**SPI_MASTER_t** *const handle, const XMC_SPI_CH_MODE_t mode) Set the communication mode along with required port configuration. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_SetBaudRate** (**SPI_MASTER_t** *const handle, const uint32_t baud_rate) Set the required baud rate during runtime. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_Transmit** (const **SPI_MASTER_t** *const handle, uint8_t *dataptr, uint32_t count) Transmits the specified number of data words and execute the callback defined in GUI, if enabled. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_Receive** (const

	SPI_MASTER_t *const handle, uint8_t *dataptr, uint32_t count) Receives the specified number of data words and execute the callback defined in GUI, if enabled. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_Transfer** (const **SPI_MASTER_t** *const handle, uint8_t *tx_dataptr, uint8_t *rx_dataptr, uint32_t count) Transmits and Receives the specified number of data words and execute the receive callback if it is enabled in GUI. More...
__STATIC_INLINE uint32_t	**SPI_MASTER_GetFlagStatus** (const **SPI_MASTER_t** *handle, const uint32_t flag) Returns the state of the specified interrupt flag. More...
__STATIC_INLINE void	**SPI_MASTER_ClearFlag** (const **SPI_MASTER_t** *handle, const uint32_t flag_mask) Clears the status of the specified interrupt flags. More...
__STATIC_INLINE bool	**SPI_MASTER_IsTxBusy** (const **SPI_MASTER_t** *const handle) return the txbusy flag state More...
__STATIC_INLINE bool	**SPI_MASTER_IsRxBusy** (const **SPI_MASTER_t** *const handle) return the rxbusy flag state More...

__STATIC_INLINE void	**SPI_MASTER_EnableSlaveSelectSignal** (const **SPI_MASTER_t** *handle, const **SPI_MASTER_SS_SIGNAL_t** slave) Enables the specified slave select line. More...
__STATIC_INLINE void	**SPI_MASTER_DisableSlaveSelectSignal** (const **SPI_MASTER_t** *handle) Disables the all the slave select lines. More...
__STATIC_INLINE uint16_t	**SPI_MASTER_GetReceivedWord** (const **SPI_MASTER_t** *const handle) Provides data received in the receive buffer. More...
__STATIC_INLINE void	**SPI_MASTER_TransmitWord** (const **SPI_MASTER_t** *const handle, const uint16_t data) Transmits a word of data. More...
__STATIC_INLINE void	**SPI_MASTER_EnableEvent** (const **SPI_MASTER_t** *const handle, const uint32_t event_mask) Enables the selected protocol events for interrupt generation. More...
__STATIC_INLINE void	**SPI_MASTER_DisableEvent** (const **SPI_MASTER_t** *const handle, const uint32_t event_mask) Disables selected events from generating interrupt. More...
__STATIC_INLINE void	**SPI_MASTER_SetTXFIFOTriggerLimit** (const **SPI_MASTER_t** *const handle,

	const uint32_t limit) Configures trigger limit for the transmit FIFO. More...
__STATIC_INLINE void	**SPI_MASTER_SetRXFIFOTriggerLimit** (const **SPI_MASTER_t** *const handle, const uint32_t limit) Configures trigger limit for the receive FIFO. More...
__STATIC_INLINE void	**SPI_MASTER_TXFIFO_EnableEvent** (const **SPI_MASTER_t** *const handle, const uint32_t event) Enables the interrupt events related to transmit FIFO. More...
__STATIC_INLINE void	**SPI_MASTER_TXFIFO_DisableEvent** (const **SPI_MASTER_t** *const handle, const uint32_t event) Disables the interrupt events related to transmit FIFO. More...
__STATIC_INLINE uint32_t	**SPI_MASTER_TXFIFO_GetEvent** (const **SPI_MASTER_t** *const handle) Gets the transmit FIFO event status. More...
__STATIC_INLINE void	**SPI_MASTER_TXFIFO_ClearEvent** (const **SPI_MASTER_t** *const handle, const uint32_t event) Clears the transmit FIFO event flags in the status register. More...
__STATIC_INLINE bool	**SPI_MASTER_IsTxFIFOFull** (const **SPI_MASTER_t** *const handle)

	Checks if the transmit FIFO is full. More...
__STATIC_INLINE void	**SPI_MASTER_RXFIFO_EnableEvent** (const **SPI_MASTER_t** *const handle, const uint32_t event) Enables the interrupt events related to transmit FIFO. More...
__STATIC_INLINE void	**SPI_MASTER_RXFIFO_DisableEvent** (const **SPI_MASTER_t** *const handle, const uint32_t event) Disables the selected interrupt events related to receive FIFO. More...
__STATIC_INLINE uint32_t	**SPI_MASTER_RXFIFO_GetEvent** (const **SPI_MASTER_t** *const handle) Get the receive FIFO events status. More...
__STATIC_INLINE void	**SPI_MASTER_RXFIFO_ClearEvent** (const **SPI_MASTER_t** *const handle, const uint32_t event) Clears the receive FIFO event flags in the status register. More...
__STATIC_INLINE bool	**SPI_MASTER_IsRxFIFOEmpty** (const **SPI_MASTER_t** *const handle) Checks if receive FIFO is empty. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_AbortTransmit** (const **SPI_MASTER_t** *const handle) Aborts the ongoing data transmission. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_AbortReceive** (const

SPI_MASTER_t *const handle)
Stops the active data reception request.
More...

Detailed Description

Methods

Function Documentation

SPI_MASTER_STATUS_t SPI_MASTER_AbortReceive (const SPI_M

Stops the active data reception request.

Parameters

handle Pointer to static and dynamic content of APP configuration.

Returns

None

Description:

If a reception is in progress, it will be stopped. When a reception request is active, user will not be able to place a new receive request till the active reception is complete. This API can stop the progressing reception to make a new receive request.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation (includes SFR declaration)
//Description:
//Transmits the string "Infineon DAVE application" to the slave.
//Starts to receive data from slave, checks if the first byte is 0x55.
//If so, aborts the reception and retransmits 0x55 to slave.
int main(void)
{
DAVE_STATUS_t status;
uint8_t Send_Data[] = "Infineon DAVE application.";
uint8_t Rec_Data[64];
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
```

```
sizeof(Send_Data));
while(SPI_MASTER_0.runtime->tx_busy);
SPI_MASTER_Receive(&SPI_MASTER_0, Rec_Data, 15U);
if(SPI_MASTER_0.runtime->rx_data[0] == 0x55)
{
SPI_MASTER_AbortReceive(&SPI_MASTER_0);
SPI_MASTER_Transmit(&SPI_MASTER_0, Rec_Data, 1);
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **813** of file **SPI_MASTER.c**.

References **SPI_MASTER::channel**, **SPI_MASTER::config**, **SPI_MASTER_CONFIG::receive_mode**, **SPI_MASTER::runtime**, **SPI_MASTER_RUNTIME::rx_busy**, **SPI_MASTER_RUNTIME::rx_data**, **SPI_MASTER_CONFIG::rx_fifo_size**, **SPI_MASTER_AbortTransmit()**, **SPI_MASTER_STATUS_FAILURE**, **SPI_MASTER_STATUS_SUCCESS**, **SPI_MASTER_TRANSFER_MODE_DIRECT**, **SPI_MASTER_TRANSFER_MODE_DMA**, and **SPI_MASTER_RUNTIME::tx_data_dummy**.

SPI_MASTER_STATUS_t SPI_MASTER_AbortTransmit (const SPI_

Aborts the ongoing data transmission.

Parameters

handle Pointer to static and dynamic content of APP configuration.

Returns

None

Description:

If there is a transmission in progress, it will be stopped. If transmit FIFO is used, the existing data will be flushed. After the transmission is stopped, user can start a new transmission without delay.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation (includes SFR declaration)
//Description:
//Transmits test data from buffer Send_Data and aborts it immediately.
//Retransmits data from NewData.
int main(void)
{
DAVE_STATUS_t status;
uint8_t Send_Data[] = "Infineon DAVE application.";
uint8_t NewData[] = "New data message";
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
sizeof(Send_Data));
if(SPI_MASTER_0.runtime->tx_busy)
{
SPI_MASTER_AbortTransmit(&SPI_MASTER_0);
SPI_MASTER_Transmit(&SPI_MASTER_0, NewData,
sizeof(NewData));
}
}
```

```
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **871** of file **SPI_MASTER.c**.

References **SPI_MASTER::channel**, **SPI_MASTER::config**, **SPI_MASTER::runtime**, **SPI_MASTER_STATUS_FAILURE**, **SPI_MASTER_STATUS_SUCCESS**, **SPI_MASTER_TRANSFER_MODE_DIRECT**, **SPI_MASTER_TRANSFER_MODE_DMA**, **SPI_MASTER_CONFIG::transmit_mode**, **SPI_MASTER_RUNTIME::tx_busy**, **SPI_MASTER_RUNTIME::tx_data**, **SPI_MASTER_RUNTIME::tx_data_dummy**, and **SPI_MASTER_CONFIG::tx_fifo_size**.

Referenced by **SPI_MASTER_AbortReceive()**.

__STATIC_INLINE void SPI_MASTER_ClearFlag (const SPI_MASTE const uint32_t)

Clears the status of the specified interrupt flags.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
flag_mask	Interrupt for which status has to be cleared Use type XMC_SPI_CH_STATUS_FLAG_t for the

bitmask of events.

Description:

During communication the events occurred has to be cleared to get the successive events.
e.g: During transmission Transmit buffer event occurs to indicating data word transfer has started. This event has to be cleared after transmission of each data word. Otherwise next event is not considered as valid.

Example Usage:

```
#include <DAVE.h>
//Description:
//It transmits "Infineon" to the SPI slave. After calling the transmit API, it will poll for the transmit shift
//indication flag to know the data has shifted out or not, and clears the flag.
int main(void)
{
DAVE_STATUS_t status;
uint8_t Send_Data[] = "Infineon";
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
sizeof(Send_Data));
while(SPI_MASTER_GetFlagStatus(&SPI_MASTER_0,
(uint32_t)XMC_SPI_CH_STATUS_FLAG_TRANSMIT_SHIFT_INDICAT
SPI_MASTER_ClearFlag(&SPI_MASTER_0,
(uint32_t)XMC_SPI_CH_STATUS_FLAG_TRANSMIT_SHIFT_INDICAT
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
```

```
}
}
return 1U;
}
```

Definition at line **1124** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**.

__STATIC_INLINE void SPI_MASTER_DisableEvent (const SPI_MA
const uint32_t
)

Disables selected events from generating interrupt.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
event	Protocol events which have to be disabled. Refer @ XMC_SPI_CH_EVENT_t for valid values. **OR** combinations of these enum item can be used as input.

Description:

Disables the SPI protocol specific events, by configuring PCR register.

After disabling the events, **SPI_MASTER_EnableEvent()** has to be invoked to re-enable the events.

Definition at line **1423** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**.

__STATIC_INLINE void SPI_MASTER_DisableSlaveSelectSignal (c

Disables the all the slave select lines.

Parameters

handle Pointer to static and dynamic content of APP configuration.

Description:

Disable all the slave signals by clearing PCR.SELO bits.

Example Usage:

```
#include <DAVE.h>
//Precondition:
//Configure to use two slaves".
//Description:
//Transmits 10 bytes of data to slave-0 and disables the slave-o. Then enable the slave-1 and transmits the data.
int main(void)
{
DAVE_STATUS_t status;
uint8_t Send_Data[] = "Infineon";
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
sizeof(Send_Data));
SPI_MASTER_DisableSlaveSelectSignal(&SPI_MASTER_0);
SPI_MASTER_EnableSlaveSelectSignal(&SPI_MASTER_0,
SPI_MASTER_SS_SIGNAL_1);
SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
sizeof(Send_Data));
}
else
{
XMC_DEBUG("main: Application initialization failed");
```

```
while(1U)
{
}
}
return 1U;
}
```

Definition at line **1344** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**.

__STATIC_INLINE void SPI_MASTER_EnableEvent (const SPI_MA
const uint32_t
)

Enables the selected protocol events for interrupt generation.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
event	Protocol events which have to be enabled. Refer @ XMC_SPI_CH_EVENT_t for valid values. **OR** combinations of these enum items can be used as input.

Description:

Enables the events by configuring CCR or PCR register based on the event. When the event is enabled, an interrupt can be generated on occurrence of the event. The API can be used for protocol events(PCR_SSC events) only when the callback functions are not registered under 'Error and Protocol Handling' group.

Definition at line **1405** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**.

__STATIC_INLINE void SPI_MASTER_EnableSlaveSelectSignal (co co)

Enables the specified slave select line.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
slave	which slave signal has to be enabled

Description:

Each slave is connected with one slave select signal. At a time only one slave can be communicate. Enable the required slave to start the communication.

Example Usage: Generate code for multiple slave by configuring in "Advanced settings tab". Transmit the data to the required slave.

```
#include <DAVE.h>
//Precondition:
//Configure to use two slaves".
//Description:
//Transmits 10 bytes of data to slave-0 and disables the slave-o. Then enable the slave-1 and transmits the data.
int main(void)
{
DAVE_STATUS_t status;
uint8_t Send_Data[] = "Infineon";
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
sizeof(Send_Data));
```

```
SPI_MASTER_DisableSlaveSelectSignal(&SPI_MASTER_0);
SPI_MASTER_EnableSlaveSelectSignal(&SPI_MASTER_0,
SPI_MASTER_SS_SIGNAL_1);
SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
sizeof(Send_Data));
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **1285** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**, **SPI_MASTER::config**, **SPI_MASTER_GPIO_CONFIG::slave_select_ch**, **SPI_MASTER_CONFIG::slave_select_pin_config**, **SPI_MASTER_SS_SIGNAL_0**, **SPI_MASTER_SS_SIGNAL_1**, **SPI_MASTER_SS_SIGNAL_2**, **SPI_MASTER_SS_SIGNAL_3**, **SPI_MASTER_SS_SIGNAL_4**, **SPI_MASTER_SS_SIGNAL_5**, **SPI_MASTER_SS_SIGNAL_6**, and **SPI_MASTER_SS_SIGNAL_7**.

DAVE_APP_VERSION_t SPI_MASTER_GetAppVersion (void)

Get **SPI_MASTER** APP version.

Returns

DAVE_APP_VERSION_t APP version information (major, minor and patch number)

Description:

The function can be used to check application software compatibility with a specific version of the APP.

Example Usage:

```
#include <DAVE.h>
int main(void)
{
DAVE_STATUS_t status;
DAVE_APP_VERSION_t app_version;
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
app_version = SPI_MASTER_GetAppVersion();
if (app_version.major != 4U)
{
// Probably, not the right version.
}
while(1U)
{
}
return 1;
}
```

Definition at line **187** of file **SPI_MASTER.c**.

__STATIC_INLINE uint32_t SPI_MASTER_GetFlagStatus (const SP
const uin
)

Returns the state of the specified interrupt flag.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
flag	Interrupt for which status is required Use type

XMC_SPI_CH_STATUS_FLAG_t for the bitmask of events.

Returns

uint32_t status of the interrupt

Description:

Returns the status of the events, by reading PSR register. This indicates the status of the all the events, for SPI communication.

Example Usage:

```
#include <DAVE.h>
//Description:
//It transmits "Infineon" to the SPI slave. After calling the transmit API, it will poll for the transmit shift
//indication flag to know the data has shifted out or not.
int main(void)
{
DAVE_STATUS_t status;
uint8_t Send_Data[] = "Infineon";
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
sizeof(Send_Data));
while(SPI_MASTER_GetFlagStatus(&SPI_MASTER_0,
XMC_SPI_CH_STATUS_FLAG_TRANSMIT_SHIFT_INDICATION));
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **1074** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**.

__STATIC_INLINE uint16_t SPI_MASTER_GetReceivedWord (const

Provides data received in the receive buffer.

Parameters

handle Pointer to static and dynamic content of APP configuration.

Returns

uint16_t Data read from the receive buffer.

Description:

This can be used in receive mode "Direct" to read the received data. If receive FIFO is not configured, function reads the value of RBUF register. Otherwise the data is read from OUTR register. User can poll for receive event or configure an interrupt by connecting external INTERRUPT APP with receive event signals. This API can be used inside the ISR to read the received data.

Definition at line **1365** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**.

SPI_MASTER_STATUS_t SPI_MASTER_Init (SPI_MASTER_t *const

Initialize the SPI channel as per the configuration made in GUI.

Parameters

handle Pointer to static and dynamic content of APP

configuration.

Returns

SPI_MASTER_STATUS_t: Status of **SPI_MASTER** driver initialization.
SPI_MASTER_STATUS_SUCCESS - on successful initialization.
SPI_MASTER_STATUS_FAILURE - if initialization fails.

Description:

Initializes IO pins used for the **SPI_MASTER** communication and configures USIC registers based on the settings provided in the GUI. Calculates divider values PDIV and STEP for a precise baudrate. It also enables configured interrupt flags and service request values.

Example Usage:

```
#include <DAVE.h> //Declarations from DAVE Code Generation (includes SFR declaration)
int main(void)
{
DAVE_STATUS_t status;
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
while(1U)
{
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
```

}

Definition at line **201** of file **SPI_MASTER.c**.

References **SPI_MASTER::config**, and **SPI_MASTER_CONFIG::fptr_spi_master_config**.

__STATIC_INLINE bool SPI_MASTER_IsRxBusy (const SPI_MASTE

return the rxbusy flag state

Parameters

handle Pointer to static and dynamic content of APP configuration.

Returns

bool : status of rxbusy flag

Description:

This is used to check whether any receive process is going or not. If no process is going then only the new request is accepted. **SPI_MASTER_AbortReceive()** can be used to stop the current process and start the new request.

Example Usage:

```
#include <DAVE.h>
//Description:
//Receives 10 bytes of data from slave.
int main(void)
{
DAVE_STATUS_t status;
uint8_t ReadData[10];
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
```

```
{
if(SPI_MASTER_Receive(&SPI_MASTER_0, ReadData, 10U))
{
while(SPI_MASTER_IsRxBusy(&SPI_MASTER_0))
{
}
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **1231** of file **SPI_MASTER.h**.

References **SPI_MASTER::runtime**, and **SPI_MASTER_RUNTIME::rx_busy**.

__STATIC_INLINE bool SPI_MASTER_IsRxFIFOEmpty (const SPI_M

Checks if receive FIFO is empty.

Parameters

handle Pointer to static and dynamic content of APP configuration.

Returns

bool true if receive FIFO is empty, false if receive FIFO has some data.

Description

When the receive FIFO is empty, received data will be put in receive FIFO. When the last received word in the FIFO is read, FIFO empty flag is set. Any attempt to read from an empty receive FIFO will set the receive FIFO error flag.

Definition at line **1650** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**.

__STATIC_INLINE bool SPI_MASTER_IsTxBusy (const SPI_MASTE

return the txbusy flag state

Parameters

handle Pointer to static and dynamic content of APP configuration.

Returns

bool : status of txbusy flag

Description:

This is used to check whether any transmit process is going or not. If no process is going then only the new request is accepted. **SPI_MASTER_AbortTransmit()** can be used to stop the current process and start the new request.

Example Usage:

```
#include <DAVE.h>
//Description:
//Transmits "Infineon" to the slave device.
int main(void)
{
DAVE_STATUS_t status;
```

```
uint8_t Send_Data[] = "Infineon";
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
if(SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
sizeof(Send_Data)) == SPI_MASTER_STATUS_SUCCESS)
{
while(SPI_MASTER_IsTxBusy(&SPI_MASTER_0))
{
}
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **1179** of file **SPI_MASTER.h**.

References **SPI_MASTER::runtime**, and **SPI_MASTER_RUNTIME::tx_busy**.

__STATIC_INLINE bool SPI_MASTER_IsTxFIFOFull (const SPI_MA

Checks if the transmit FIFO is full.

Parameters

handle Pointer to static and dynamic content of APP configuration.

Returns

bool Status of transmit FIFO filling level.
true - if transmit FIFO is full.
false - if transmit FIFO is not full.

Description

Checks if transmit FIFO is full.

Checks the status using the register TRBSR. Can be used while filling data to the transmit FIFO.

Definition at line **1559** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**.

SPI_MASTER_STATUS_t SPI_MASTER_Receive (const SPI_MASTI
uint8_t *
uint32_t
)

Receives the specified number of data words and execute the callback defined in GUI, if enabled.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
dataptr	Pointer to data in which value is written
count	number of data words (word length configured) to be read

Returns

SPI_MASTER_STATUS_t SPI_MASTER_STATUS_SUCCESS : if read is successful
SPI_MASTER_STATUS_BUSY : if SPI channel is busy with other operation

Description:

Data will be received from the SPI slave synchronously. After the requested number of data bytes are received, optionally, the user configured callback function will be executed. Data reception is accomplished using the receive mode selected in the UI. **Interrupt:**

Based on the UI configuration, either standard receive buffer(RBUF) or receive FIFO(OUT) is used for data reception. An interrupt is configured for reading received data from the bus. This function only registers a request to receive a number of data bytes from a USIC channel. If FIFO is configured for reception, the FIFO limit is dynamically configured to optimally utilize the CPU load. Before starting data reception, the receive buffers are flushed. So only those data, received after calling the API, will be placed in the user buffer. When all the requested number of data bytes are received, the configured callback function will be executed. If a callback function is not configured, the user has to poll for the value of the variable, *handle->runtime->rx_busy* to be false. The value is updated to *false* when all the requested number of data bytes are received.

DMA:

DMA mode is available only in XMC4x family of microcontrollers. In this mode, a DMA channel is configured for receiving data from standard receive buffer(RBUF) to the user buffer. By calling this API, the DMA channel destination address is configured to the user buffer and the channel is enabled. Receive FIFO will not be used when the receive mode is DMA. Before starting data reception, the receive buffers are flushed. So only those data, received after calling the API, will be placed in the user buffer. When all the requested number of data bytes are received, the configured callback function will be executed. If a callback function is not configured, the user has to poll for the value of the variable, *handle->runtime->rx_busy* to be false. The value is updated to *false* when all the requested number of data bytes are received.

Direct

In Direct receive mode, neither interrupt nor DMA is used. The

API polls the receive flag to read the received data and waits for all the requested number of bytes to be received. Based on FIFO configuration, either RBUF or OUT register is used for reading received data. Before starting data reception, the receive buffers are flushed. So only those data, received after calling the API, will be placed in the user buffer. ***Note:*** *In Direct mode, the API blocks the CPU until the count of bytes requested is received.*

Example Usage:

```
#include <DAVE.h>
//Description:
//Receives 10 bytes of data from slave.
int main(void)
{
DAVE_STATUS_t status;
uint8_t ReadData[10];
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
if(SPI_MASTER_Receive(&SPI_MASTER_0, ReadData, 10U))
{
while(SPI_MASTER_0.runtime->rx_busy)
{
}
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **322** of file **SPI_MASTER.c**.

References **SPI_MASTER::config**, **SPI_MASTER_CONFIG::receive_mode**, **SPI_MASTER_STATUS_FAILURE**, **SPI_MASTER_TRANSFER_MODE_DIRECT**, **SPI_MASTER_TRANSFER_MODE_DMA**, and **SPI_MASTER_TRANSFER_MODE_INTERRUPT**.

__STATIC_INLINE void SPI_MASTER_RXFIFO_ClearEvent (const S const u)

Clears the receive FIFO event flags in the status register.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
event	Receive FIFO events to be cleared. **Range:** XMC_USIC_CH_RXFIFO_EVENT_STANDARD, XMC_USIC_CH_RXFIFO_EVENT_ERROR, XMC_USIC_CH_RXFIFO_EVENT_ALTERNATE.

Description

USIC channel peripheral does not clear the event flags after they are read. This API clears the events provided in the *mask* value. XMC_USIC_CH_RXFIFO_EVENT enumeration can be used as input.

Definition at line **1632** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**.

__STATIC_INLINE void SPI_MASTER_RXFIFO_DisableEvent (const const)

Disables the selected interrupt events related to receive FIFO.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
event	Events to be disabled. **Range:** XMC_USIC_CH_RXFIFO_EVENT_CONF_STANDARD, XMC_USIC_CH_RXFIFO_EVENT_CONF_ERROR, XMC_USIC_CH_RXFIFO_EVENT_CONF_ALTERNATE.

Description

By disabling the interrupt events, generation of interrupt is stopped. User can poll the event flags from the status register using the API **SPI_MASTER_RXFIFO_GetEvent()**.

Definition at line **1595** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**.

__STATIC_INLINE void SPI_MASTER_RXFIFO_EnableEvent (const const)

Enables the interrupt events related to transmit FIFO.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
event	Events to be enabled. Multiple events can be bitwise OR combined.

Range:
XMC_USIC_CH_RXFIFO_EVENT_CONF_STANDARD,
XMC_USIC_CH_RXFIFO_EVENT_CONF_ERROR,
XMC_USIC_CH_RXFIFO_EVENT_CONF_ALTERNATE.

Description

Multiple events can be enabled by providing multiple events in a single call. For providing multiple events, combine the events using bitwise OR operation.

Definition at line **1577** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**.

__STATIC_INLINE uint32_t SPI_MASTER_RXFIFO_GetEvent (const

Get the receive FIFO events status.

Parameters

handle Pointer to static and dynamic content of APP configuration.

Returns

uint32_t Status of receive FIFO events.

Description

Gives the status of receive FIFO standard receive buffer event, alternative receive buffer event and receive buffer error event. The status bits are located at their bit positions in the TRBSR register in the returned value. User can make use of the XMC_USIC_CH_RXFIFO_EVENT enumeration for checking the status of return value.

Definition at line **1613** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**.

SPI_MASTER_STATUS_t SPI_MASTER_SetBaudRate (SPI_MASTE
const uint32
)

Set the required baud rate during runtime.

Parameters

handle	handle Pointer to static and dynamic content of APP configuration.
baud_rate	required baud rate

Returns

SPI_MASTER_STATUS_t SPI_MASTER_STATUS_SUCCESS : if updation of baud rate is successful
SPI_MASTER_STATUS_FAILURE : if updation is failed
SPI_MASTER_STATUS_BUSY : if SPI channel is busy with other operation

Description:

While setting the baud rate to avoid noise of the port pins, all the pins are changed to input. After setting the required baud again ports are initialised with the configured settings.

Example Usage:

```
#include <DAVE.h>
//Description:
//The following code changes the SPI master baud rate to 9600 and starts sending the data stored in
//the buffer.
int main(void)
{
DAVE_STATUS_t status;
SPI_MASTER_STATUS_t spi_status;
uint8_t Send_Data[] = "Infineon DAVE application.";
```

```
uint32_t baud_rate;
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
baud_rate = 9600U;
spi_status = SPI_MASTER_SetBaudRate(&SPI_MASTER_0,
baud_rate);
if(spi_status == SPI_MASTER_STATUS_SUCCESS)
{
SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
sizeof(Send_Data));
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **250** of file **SPI_MASTER.c**.

References **SPI_MASTER::channel**, **SPI_MASTER::config**, **SPI_MASTER_CONFIG::leading_trailing_delay**, **SPI_MASTER::runtime**, **SPI_MASTER_RUNTIME::rx_busy**, **SPI_MASTER_CONFIG::shift_clk_passive_level**, **SPI_MASTER_STATUS_BUSY**, **SPI_MASTER_STATUS_SUCCESS**, and **SPI_MASTER_RUNTIME::tx_busy**.

SPI_MASTER_STATUS_t SPI_MASTER_SetMode (SPI_MASTER_t *

const XMC_SPI_

)

Set the communication mode along with required port configuration.

Parameters

handle handle Pointer to static and dynamic content of APP configuration.

mode SPI working mode

Returns

SPI_MASTER_STATUS_t SPI_MASTER_STATUS_SUCCESS : if updation of settings are successful
SPI_MASTER_STATUS_FAILURE : if mode is not supported by the selected pins
SPI_MASTER_STATUS_BUSY : if SPI channel is busy with transmit or receive operation

Description:

To change the mode of communication, it is advised to generate the code in Quad/Dual mode initially. Then changing the mode will be taken care by the APP.

- If code is generated for Quad mode, it is possible to change to other modes like Dual, Half Duplex and Full Duplex
- If code is generated for Dual mode, it is possible to change to other modes like Half Duplex and Full Duplex only
- If code is generated for full-duplex mode, it is possible to change to Half Duplex only

Example Usage:

```
#include <DAVE.h>
//Precondition:
//Configure the SPI_MASTER APP operation mode as 'Quad SPI'.
```

```
//Description:
//The following code changes the SPI master device mode to Full
duplex mode and starts sending the data stored in
//the buffer.
int main(void)
{
DAVE_STATUS_t status;
SPI_MASTER_STATUS_t spi_status;
uint8_t Send_Data[] = "Infineon DAVE application.";
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
spi_status = SPI_MASTER_SetMode(&SPI_MASTER_0,
XMC_SPI_CH_MODE_STANDARD);
if(spi_status == SPI_MASTER_STATUS_SUCCESS)
{
SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
sizeof(Send_Data));
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **216** of file **SPI_MASTER.c**.

References **SPI_MASTER::runtime**,
SPI_MASTER_RUNTIME::rx_busy,
SPI_MASTER_RUNTIME::spi_master_mode,
SPI_MASTER_STATUS_BUSY,

SPI_MASTER_STATUS_SUCCESS, and **SPI_MASTER_RUNTIME::tx_busy**.

__STATIC_INLINE void SPI_MASTER_SetRXFIFOTriggerLimit (cons cons)

Configures trigger limit for the receive FIFO.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
size	Value of receive FIFO filling level, transition above which the interrupt should be generated. : 0 to receive FIFO size. e.g, If receive FIFO size is 16, and limit is configured as 8, FIFO receive buffer interrupt will be generated when the FIFO filling level rises from 8 to 9.

Description

Receive FIFO trigger limit is configured by setting its value in the RBCTR register. Receive FIFO is configured to generate interrupt when the FIFO filling level rises above the trigger limit.

Definition at line **1464** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**, **SPI_MASTER::config**, and **SPI_MASTER_CONFIG::rx_fifo_size**.

__STATIC_INLINE void SPI_MASTER_SetTXFIFOTriggerLimit (cons cons)

Configures trigger limit for the transmit FIFO.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
limit	Value of transmit FIFO filling level, transition below which the interrupt should be generated. : 0 to transmit FIFO size. e.g, If transmit FIFO size is 16, and limit is configured as 8, FIFO standard transmit buffer interrupt will be generated when the FIFO filling level drops from 8 to 7.

Description

Transmit FIFO trigger limit is configured by setting its value in the TBCTR register. Transmit FIFO is configured to generate interrupt when the FIFO filling level drops below the trigger limit.

Definition at line **1443** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**, **SPI_MASTER::config**, and **SPI_MASTER_CONFIG::tx_fifo_size**.

SPI_MASTER_STATUS_t SPI_MASTER_Transfer (const SPI_MAST
uint8_t *
uint8_t *
uint32_t
)

Transmits and Receives the specified number of data words and execute the receive callback if it is enabled in GUI.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
tx_dataptr	Pointer to data buffer which has to be send
rx_dataptr	Pointer to data buffer where the received data has

	to be stored.
count	number of data words (word length configured) to be read and write

Returns

SPI_MASTER_STATUS_t SPI_MASTER_STATUS_SUCCESS : if transfer of data is successful
SPI_MASTER_STATUS_FAILURE : if transfer of data is failed (or) in other than standard full duplex mode
SPI_MASTER_STATUS_BUFFER_INVALID : if passed buffers are NULL pointers (or) length of data transfer is zero.

Description:

Transmits and receives data simultaneously using the SPI channel as a master device. API is applicable only in *Full duplex</> operation mode. Data transfer happens based on the individual modes configured for transmission and reception. Two data pins MOSI and MISO will be used for receiving and transmitting data respectively. A callback function can be configured to execute after completing the transfer when 'Interrupt' or 'DMA' mode is used. The callback function should be configured for End of receive/transfer callback* in the 'Interrupt Settings' tab. The callback function will be executed when the last word of data is received.

Example Usage:

```
#include <DAVE.h>
//Precondition: Operation mode should be 'Full Duplex"
//Description:
//Transmits and Receives 10 bytes of data from slave in parallel.
int main(void)
{
DAVE_STATUS_t status;
uint8_t ReadData[10];
uint8_t SendData[10] = {0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9,
```

```
0xA};
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
SPI_MASTER_Transfer(&SPI_MASTER_0, SendData, ReadData,
10);
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **747** of file **SPI_MASTER.c**.

References **SPI_MASTER::config**, **SPI_MASTER_CONFIG::receive_mode**, **SPI_MASTER::runtime**, **SPI_MASTER_RUNTIME::rx_busy**, **SPI_MASTER_RUNTIME::rx_data**, **SPI_MASTER_RUNTIME::rx_data_dummy**, **SPI_MASTER_RUNTIME::spi_master_mode**, **SPI_MASTER_STATUS_BUFFER_INVALID**, **SPI_MASTER_STATUS_BUSY**, **SPI_MASTER_STATUS_FAILURE**, **SPI_MASTER_TRANSFER_MODE_DIRECT**, **SPI_MASTER_TRANSFER_MODE_DMA**, **SPI_MASTER_TRANSFER_MODE_INTERRUPT**, **SPI_MASTER_RUNTIME::tx_busy**, **SPI_MASTER_RUNTIME::tx_data**, **SPI_MASTER_RUNTIME::tx_data_count**, and **SPI_MASTER_RUNTIME::tx_data_dummy**.

SPI_MASTER_STATUS_t SPI_MASTER_Transmit (const SPI_MAST
uint8_t *
uint32_t
)

Transmits the specified number of data words and execute the callback defined in GUI, if enabled.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
dataptr	Pointer to data
count	number of data words (word length configured) to be transmitted

Returns

SPI_MASTER_STATUS_t SPI_MASTER_STATUS_SUCCESS : if transmit is successful
SPI_MASTER_STATUS_BUSY : if SPI channel is busy with other operation

Description:

Transmits data using the SPI channel as a master device. Transmission is accomplished using the transmit mode as configured in the UI.

Interrupt:

The data transmission is accomplished using transmit interrupt. User can configure a callback function in the APP UI. When the data is fully transmitted, the callback function will be executed. If transmit FIFO is enabled, the trigger limit is set to 1. So the transmit interrupt will be generated when all the data in FIFO is moved out of FIFO. The APP handle's runtime structure is used to store the data pointer, count, data index and status of transmission. This function only registers a data transmission request if there is no active transmission in progress. Actual data transmission happens in the transmit interrupt service routine. A

trigger is generated for the transmit interrupt to start loading the data to the transmit buffer. If transmit FIFO is configured, the data is filled into the FIFO. Transmit interrupt will be generated subsequently when the transmit FIFO is empty. At this point of time, if there is some more data to be transmitted, it is loaded to the FIFO again. When FIFO is not enabled, data is transmitted one byte at a time. On transmission of each byte an interrupt is generated and the next byte is transmitted in the interrupt service routine. Callback function is executed when all the data bytes are transmitted. If a callback function is not configured, user has to poll for the value of *tx_busy* flag of the APP handle structure(*handle->runtime->tx_busy*) to check for the completion of data transmission or use **SPI_MASTER_IsTxBusy()** API.

DMA:

DMA mode is available only in XMC4x family of microcontrollers. A DMA channel is configured to provide data to the SPI channel transmit buffer. This removes the load off the CPU. This API will only configure and enable the DMA channel by specifying the data buffer and count of bytes to transmit. Rest is taken care without the CPU's intervention. User can configure a callback function in the APP UI. When the transmission is complete, the callback function will be executed. FIFO will not be used in DMA mode. Receive start interrupt is configured for triggering the DMA channel. So each byte is transmitted in the background through the DMA channel. If the callback function is not configured, *handle->runtime->tx_busy* flag can be checked to verify if the transmission is complete. **Direct:**

Data will be transmitted using polling method. Status flags are used to check if data can be transmitted. ***Note:*** *In Direct mode, the API blocks the CPU until the count of bytes requested is transmitted.*

Example Usage:

```
#include <DAVE.h>
//Description:
```

```
//Transmits "Infineon" to the slave device.
int main(void)
{
DAVE_STATUS_t status;
uint8_t Send_Data[] = "Infineon";
status = DAVE_Init(); // SPI_MASTER_Init() is called from DAVE_Init()
if(status == DAVE_STATUS_SUCCESS)
{
if(SPI_MASTER_Transmit(&SPI_MASTER_0, Send_Data,
sizeof(Send_Data)) == SPI_MASTER_STATUS_SUCCESS)
{
while(SPI_MASTER_0.runtime->tx_busy)
{
}
}
}
else
{
XMC_DEBUG("main: Application initialization failed");
while(1U)
{
}
}
return 1U;
}
```

Definition at line **292** of file **SPI_MASTER.c**.

References **SPI_MASTER::config**,
SPI_MASTER_STATUS_FAILURE,
SPI_MASTER_TRANSFER_MODE_DIRECT,
SPI_MASTER_TRANSFER_MODE_DMA,
SPI_MASTER_TRANSFER_MODE_INTERRUPT, and
SPI_MASTER_CONFIG::transmit_mode.

__STATIC_INLINE void SPI_MASTER_TransmitWord (const SPI_MA const uint16_)

Transmits a word of data.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
data	Data to be transmitted

Description:

Transmits a word of data through the SPI channel as a master device. If transmit FIFO is configured, the data is placed in the IN[0] register of the USIC channel. If transmit FIFO is not configured, API waits for the TBUF to be free and then places the data in the TBUF register. User can poll for receive event or configure interrupt by connecting an external INTERRUPT APP. This API can be used inside the ISR to read the received data.

Definition at line **1384** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**, **SPI_MASTER::runtime**, and **SPI_MASTER_RUNTIME::spi_master_mode**.

__STATIC_INLINE void SPI_MASTER_TXFIFO_ClearEvent (const S const u)

Clears the transmit FIFO event flags in the status register.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
event	Transmit FIFO events to be cleared.

Range:
XMC_USIC_CH_TXFIFO_EVENT_STANDARD,
XMC_USIC_CH_TXFIFO_EVENT_ERROR.

Returns

None

Description

USIC channel peripheral does not clear the event flags after they are read. This API clears the events provided in the *mask* value. XMC_USIC_CH_TXFIFO_EVENT enumeration can be used as input. Multiple events can be cleared by providing a mask value obtained by bitwise OR operation of multiple event enumerations.

Definition at line **1540** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**.

__STATIC_INLINE void SPI_MASTER_TXFIFO_DisableEvent (const const)

Disables the interrupt events related to transmit FIFO.

Parameters

handle Pointer to static and dynamic content of APP configuration.

event Events to be disabled.
Range:
XMC_USIC_CH_TXFIFO_EVENT_CONF_STANDARD,
XMC_USIC_CH_TXFIFO_EVENT_CONF_ERROR.

Description

By disabling the interrupt events, generation of interrupt is stopped. User can poll the event flags from the status register

using the API **SPI_MASTER_TXFIFO_GetEvent()**. Event bitmasks can be constructed using the enumeration XMC_USIC_CH_TXFIFO_EVENT_CONF. For providing multiple events, combine the events using bitwise OR operation.

Definition at line **1500** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**.

__STATIC_INLINE void SPI_MASTER_TXFIFO_EnableEvent (const const)

Enables the interrupt events related to transmit FIFO.

Parameters

handle	Pointer to static and dynamic content of APP configuration.
event	Events to be enabled. Multiple events can be bitwise OR combined. XMC_USIC_CH_TXFIFO_EVENT_CONF_STANDARD, XMC_USIC_CH_TXFIFO_EVENT_CONF_ERROR.

Description

Event bitmasks can be constructed using the enumeration XMC_USIC_CH_TXFIFO_EVENT_CONF.For providing multiple events, combine the events using bitwise OR operation. Events are configured in the TBCTR register.

Definition at line **1481** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**.

__STATIC_INLINE uint32_t SPI_MASTER_TXFIFO_GetEvent (const

Gets the transmit FIFO event status.

Parameters

handle Pointer to static and dynamic content of APP configuration.

Returns

Status of standard transmit and transmit buffer error events. **Range:** XMC_USIC_CH_TXFIFO_EVENT_STANDARD, XMC_USIC_CH_TXFIFO_EVENT_ERROR.

Description

Gives the status of transmit FIFO standard transmit buffer event and transmit buffer error event. The status bits are located at their bit positions in the TRBSR register in the returned value. User can make use of the XMC_USIC_CH_TXFIFO_EVENT enumeration for checking the status of return value. The status can be found by using the bitwise AND operation on the returned value with the enumerated value.

Definition at line **1519** of file **SPI_MASTER.h**.

References **SPI_MASTER::channel**.