SPI_MASTER

Data Structures

Here are the data structures with brief descriptions:

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

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

Home

Data Structures | Data Structure Index | Data Fields

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_pir**
const **SPI_MASTER_GPIO_CONFIG_t** *const	**mosi_0_pir**
const **SPI_MASTER_GPIO_t** *const	**mosi_1_pir**
const **SPI_MASTER_GPIO_CONFIG_t** *const	**mosi_1_pir**
const **SPI_MASTER_GPIO_t** *const	**mosi_2_pir**
const **SPI_MASTER_GPIO_CONFIG_t** *const	**mosi_2_pir**
const **SPI_MASTER_GPIO_t** *const	**mosi_3_pir**
const **SPI_MASTER_GPIO_CONFIG_t** *const	**mosi_3_pir**
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_sele**
const **SPI_MASTER_GPIO_CONFIG_t**	**slave_sele**

*const	[8]
SPI_MASTER_functionhandler	**tx_cbhandl**
SPI_MASTER_functionhandler	**rx_cbhandl**
SPI_MASTER_functionhandler	**parity_cbha**
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_s

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_o

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_MAST

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_

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

Home

Data Structures | Data Structure Index | Data Fields

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**

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

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

Data Structure Index

S

S	SPI_MASTER_CONFIG	SPI_MASTER_GPIO_CON
	SPI_MASTER_GPIO	SPI_MASTER_RUNTIM
SPI_MASTER		

S

SPI_MASTER

Home

Data Structures | Data Structure Index | Data Fields

All | Variables

c | d | f | h | l | m | p | r | s | t | w

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**

- 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**

File List

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

- SPI_MASTER.c
- SPI_MASTER.h

Home

File List Globals

Code

Functions

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

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.

Data Structures

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_MASTE** 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 nun and execute the callback de enabled. More...
SPI_MASTER_STATUS_t	**SPI_MASTER_Receive** (co *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 events

	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_S
	SPI_MASTER_STATUS_FA
	SPI_MASTER_STATUS_B
	SPI_MASTER_STATUS_B
	SPI_MASTER_STATUS_I
	}
	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_TRANSFE** **SPI_MASTER_TRANSFE** **SPI_MASTER_TRANSFE** **SPI_MASTER_TRANSFE**

	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

Home

File List Globals

All Functions Typedefs Enumerations Enumerator

s

- 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_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_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

Home

File List | Globals

Code

SPI_MASTER.h

Go to the documentation of this file.

```

#ifndef SPI_MASTER_H
#define SPI_MASTER_H
/*********************************************************************************
 * HEADER FILES
 
*********************************************************************************
#include <xmc_gpio.h>
#include <xmc_scu.h>
#include <xmc_spi.h>
#include <DAVE_Common.h>
#include "spi_master_conf.h"

#if((SPI_MASTER_DMA_TRANSMIT_MODE == 1U) ||
(SPI_MASTER_DMA_RECEIVE_MODE == 1U))
#include "./GLOBAL_DMA/global_dma.h"
#endif

/********************************************************************************
 * MACROS
 
*********************************************************************************
#if (!((XMC_LIB_MAJOR_VERSION == 2U) && \
 (XMC_LIB_MINOR_VERSION >= 1U) && \
 (XMC_LIB_PATCH_VERSION >= 6U)))
#error "SPI_MASTER requires XMC Peripheral Library v2.1.6 or
```

```
higher"
#endif


/*
 * @brief Represents the maximum data size for DMA transaction*/
#define SPI_MASTER_DMA_MAXCOUNT (4095U)
/****************************************************************************
 * ENUMS

**********************************************************************************
typedef enum SPI_MASTER_STATUS
{
 SPI_MASTER_STATUS_SUCCESS = 0U,
 SPI_MASTER_STATUS_FAILURE,
 SPI_MASTER_STATUS_BUSY,
 SPI_MASTER_STATUS_BUFFER_INVALID,
 SPI_MASTER_STATUS_MODE_MISMATCH
} SPI_MASTER_STATUS_t;

typedef enum SPI_MASTER_SR_ID
{
 SPI_MASTER_SR_ID_0 = 0U,
 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
} SPI_MASTER_SR_ID_t;

typedef enum SPI_MASTER_SS_SIGNAL
{
 SPI_MASTER_SS_SIGNAL_0 = 0U,
 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,
 SPI_MASTER_SS_SIGNAL_7
} SPI_MASTER_SS_SIGNAL_t;

typedef enum SPI_MASTER_INPUT
{
 SPI_MASTER_INPUT_A = 0U,
 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_INVALID
} SPI_MASTER_INPUT_t;

typedef enum SPI_MASTER_TRANSFER_MODE
{
 SPI_MASTER_TRANSFER_MODE_INTERRUPT,
 SPI_MASTER_TRANSFER_MODE_DMA,
 SPI_MASTER_TRANSFER_MODE_DIRECT
} SPI_MASTER_TRANSFER_MODE_t;
typedef void (*SPI_MASTER_functionhandler)(void);
typedef SPI_MASTER_STATUS_t (*SPI_MASTER_lInit_functionhandler)(void);

/******************************************************************************
* DATA STRUCTURES
******************************************************************************
typedef struct SPI_MASTER_GPIO
{
 XMC_GPIO_PORT_t* port;
 uint8_t pin;
} SPI_MASTER_GPIO_t;

typedef struct SPI_MASTER_GPIO_CONFIG
```

```
{
  XMC_GPIO_CONFIG_t port_config;
  XMC_GPIO_HWCTRL_t hw_control;
  XMC_SPI_CH_SLAVE_SELECT_t slave_select_ch;
} SPI_MASTER_GPIO_CONFIG_t;

  typedef struct SPI_MASTER_CONFIG
{
  XMC_SPI_CH_CONFIG_t * const channel_config;
  SPI_MASTER_lInit_functionhandler fptr_spi_master_config;
  /* Port configuration */
  const SPI_MASTER_GPIO_t* const mosi_0_pin;
  const SPI_MASTER_GPIO_CONFIG_t* const mosi_0_pin_config;
  const SPI_MASTER_GPIO_t* const mosi_1_pin;
  const SPI_MASTER_GPIO_CONFIG_t* const mosi_1_pin_config;
  const SPI_MASTER_GPIO_t* const mosi_2_pin;
  const SPI_MASTER_GPIO_CONFIG_t* const mosi_2_pin_config;
  const SPI_MASTER_GPIO_t* const mosi_3_pin;
  const SPI_MASTER_GPIO_CONFIG_t* const mosi_3_pin_config;
  const SPI_MASTER_GPIO_t* const sclk_out_pin;
  const SPI_MASTER_GPIO_CONFIG_t* const sclk_out_pin_config;
  const SPI_MASTER_GPIO_t* const slave_select_pin[8];
  const SPI_MASTER_GPIO_CONFIG_t* const slave_select_pin_config[8];
  SPI_MASTER_functionhandler tx_cbhandler;
  SPI_MASTER_functionhandler rx_cbhandler;
  SPI_MASTER_functionhandler parity_cbhandler;
  /* FIFO configuration */
  XMC_USIC_CH_FIFO_SIZE_t tx_fifo_size;
  XMC_USIC_CH_FIFO_SIZE_t rx_fifo_size;
  /* Clock Settings */
  XMC_SPI_CH_BRG_SHIFT_CLOCK_PASSIVE_LEVEL_t shift_clk_passive_level;
  SPI_MASTER_TRANSFER_MODE_t transmit_mode;
  SPI_MASTER_TRANSFER_MODE_t receive_mode;
  XMC_SPI_CH_MODE_t spi_master_config_mode;
```

```
 uint8_t slave_select_lines;
 uint8_t leading_trailing_delay;
 SPI_MASTER_SR_ID_t tx_sr;
 SPI_MASTER_SR_ID_t rx_sr;
 SPI_MASTER_SR_ID_t parity_sr;
} SPI_MASTER_CONFIG_t;

typedef struct SPI_MASTER_RUNTIME
{
 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;
 } SPI_MASTER_RUNTIME_t;


typedef struct SPI_MASTER
{
 XMC_USIC_CH_t* const channel;
 const SPI_MASTER_CONFIG_t * const config;
 SPI_MASTER_RUNTIME_t * const runtime;
#if ((SPI_MASTER_DMA_TRANSMIT_MODE == 1U) || (SPI_MASTER_DMA_RECEIVE_MODE == 1U))
 const GLOBAL_DMA_t * const global_dma;
#endif
#if (SPI_MASTER_DMA_TRANSMIT_MODE == 1U)
 const XMC_DMA_CH_CONFIG_t * const dma_ch_tx_config;
```

```
#endif
#if (SPI_MASTER_DMA_RECEIVE_MODE == 1U)
 const XMC_DMA_CH_CONFIG_t * const dma_ch_rx_config;
 const GLOBAL_DMA_t * const global_dma_rx;
 const uint8_t dma_ch_rx_number;
#endif
#if (SPI_MASTER_DMA_TRANSMIT_MODE == 1U)
 const uint8_t dma_ch_tx_number;
#endif
} SPI_MASTER_t;

/*******************************************************************************
* API Prototypes
********************************************************************************
#ifdef __cplusplus
extern "C" {
#endif

DAVE_APP_VERSION_t SPI_MASTER_GetAppVersion(void);

SPI_MASTER_STATUS_t SPI_MASTER_Init(SPI_MASTER_t* const handle);

SPI_MASTER_STATUS_t SPI_MASTER_SetMode(SPI_MASTER_t* const handle, const XMC_SPI_CH_MODE_t mode);

SPI_MASTER_STATUS_t SPI_MASTER_SetBaudRate(SPI_MASTER_t* const handle, const uint32_t baud_rate);

SPI_MASTER_STATUS_t SPI_MASTER_Transmit(const SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count);

SPI_MASTER_STATUS_t SPI_MASTER_Receive(const SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count);

SPI_MASTER_STATUS_t SPI_MASTER_Transfer(const SPI_MASTER_t *const handle,
 uint8_t* tx_dataptr,
 uint8_t* rx_dataptr,
 uint32_t count);

#if (SPI_MASTER_INTERRUPT_RECEIVE_MODE == 1U)

SPI_MASTER_STATUS_t SPI_MASTER_StartReceiveIRQ(const SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count);
#endif

#if(SPI_MASTER_INTERRUPT_TRANSMIT_MODE == 1U)

SPI_MASTER_STATUS_t SPI_MASTER_StartTransmitIRQ(const SPI_MASTER_t *const handle, uint8_t *addr, uint32_t count);
#endif


#if(SPI_MASTER_DMA_RECEIVE_MODE == 1U)

SPI_MASTER_STATUS_t SPI_MASTER_StartReceiveDMA(const SPI_MASTER_t *const handle, uint8_t *addr, uint32_t block_size);
#endif

#if(SPI_MASTER_DMA_TRANSMIT_MODE == 1U)

SPI_MASTER_STATUS_t SPI_MASTER_StartTransmitDMA(const SPI_MASTER_t *const handle, uint8_t *addr, uint32_t block_size);
#endif

__STATIC_INLINE uint32_t SPI_MASTER_GetFlagStatus(const SPI_MASTER_t* handle, const uint32_t flag)
{
 XMC_ASSERT("SPI_MASTER_GetFlagStatus:handle NULL" , (handle != NULL));
```

```
  return (XMC_SPI_CH_GetStatusFlag(handle->channel) & flag);
}

__STATIC_INLINE void SPI_MASTER_ClearFlag(const SPI_MASTER_t* handle, const uint32_t flag_mask)
{
  XMC_ASSERT("SPI_MASTER_ClearFlag:handle NULL" , (handle != NULL));
  XMC_SPI_CH_ClearStatusFlag(handle->channel, flag_mask);
}

__STATIC_INLINE bool SPI_MASTER_IsTxBusy(const SPI_MASTER_t* const handle)
{
  XMC_ASSERT("SPI_MASTER_IsTxBusy:handle NULL", (handle != NULL))
  return (handle->runtime->tx_busy);
}

__STATIC_INLINE bool SPI_MASTER_IsRxBusy(const SPI_MASTER_t* const handle)
{
  XMC_ASSERT("SPI_MASTER_IsTxBusy:handle NULL", (handle != NULL))
  return (handle->runtime->rx_busy);
}


__STATIC_INLINE void SPI_MASTER_EnableSlaveSelectSignal(const SPI_MASTER_t* handle, const SPI_MASTER_SS_SIGNAL_t slave)
{
  XMC_ASSERT("SPI_MASTER_EnableSlaveSelectSignal:handle NULL" , (handle != NULL));
  XMC_ASSERT("SPI_MASTER_EnableSlaveSelectSignal:Invalid Slave selection" , ((slave == SPI_MASTER_SS_SIGNAL_0) ||
  (slave == SPI_MASTER_SS_SIGNAL_1) ||
  (slave == SPI_MASTER_SS_SIGNAL_2) ||
  (slave == SPI_MASTER_SS_SIGNAL_3) ||
  (slave == SPI_MASTER_SS_SIGNAL_4) ||
  (slave == SPI_MASTER_SS_SIGNAL_5) ||
  (slave == SPI_MASTER_SS_SIGNAL_6) ||
  (slave == SPI_MASTER_SS_SIGNAL_7))
  );
  XMC_SPI_CH_EnableSlaveSelect(handle->channel, handle->config->slave_select_pin_config[slave]->slave_select_ch);
}

__STATIC_INLINE void SPI_MASTER_DisableSlaveSelectSignal(const SPI_MASTER_t* handle)
{
  XMC_ASSERT("SPI_MASTER_Transmit:handle NULL" , (handle != NULL));
  XMC_SPI_CH_DisableSlaveSelect(handle->channel);
}

__STATIC_INLINE uint16_t SPI_MASTER_GetReceivedWord(const SPI_MASTER_t *const handle)
{
  XMC_ASSERT("SPI_MASTER_GetReceivedWord:handle NULL" , (handle != NULL));
  return XMC_SPI_CH_GetReceivedData(handle->channel);
}

__STATIC_INLINE void SPI_MASTER_TransmitWord(const SPI_MASTER_t *const handle, const uint16_t data)
{
  XMC_ASSERT("SPI_MASTER_TransmitWord:handle NULL" , (handle != NULL));
  XMC_SPI_CH_Transmit(handle->channel, data, handle->runtime->spi_master_mode);
}
```

```

__STATIC_INLINE void SPI_MASTER_EnableEvent(const SPI_MASTER_t *const handle, const uint32_t event_mask)
{
  XMC_ASSERT("SPI_MASTER_EnableEvent:handle NULL" , (handle != NULL));
  XMC_SPI_CH_EnableEvent(handle->channel, event_mask);
}

__STATIC_INLINE void SPI_MASTER_DisableEvent(const SPI_MASTER_t *const handle, const uint32_t event_mask)
{
  XMC_ASSERT("SPI_MASTER_DisableEvent:handle NULL" , (handle != NULL));
  XMC_SPI_CH_DisableEvent(handle->channel, event_mask);
}

__STATIC_INLINE void SPI_MASTER_SetTXFIFOTriggerLimit(const SPI_MASTER_t *const handle, const uint32_t limit)
{
  XMC_ASSERT("SPI_MASTER_SetTXFIFOTriggerLimit:handle NULL" , (handle != NULL));
  XMC_USIC_CH_TXFIFO_SetSizeTriggerLimit(handle->channel, handle->config->tx_fifo_size, limit);
}

__STATIC_INLINE void SPI_MASTER_SetRXFIFOTriggerLimit(const SPI_MASTER_t *const handle, const uint32_t limit)
{
  XMC_ASSERT("SPI_MASTER_SetRXFIFOTriggerLimit:handle NULL" , (handle != NULL));
  XMC_USIC_CH_RXFIFO_SetSizeTriggerLimit(handle->channel, handle->config->rx_fifo_size, limit);
}

```

```
__STATIC_INLINE void SPI_MASTER_TXFIFO_EnableEvent(const SPI_MASTER_t *const handle, const uint32_t event)
{
  XMC_ASSERT("SPI_MASTER_TXFIFO_EnableEvent:handle NULL" , (handle != NULL));
  XMC_USIC_CH_TXFIFO_EnableEvent(handle->channel, event);
}

__STATIC_INLINE void SPI_MASTER_TXFIFO_DisableEvent(const SPI_MASTER_t *const handle, const uint32_t event)
{
  XMC_ASSERT("SPI_MASTER_TXFIFO_DisableEvent:handle NULL" , (handle != NULL));
  XMC_USIC_CH_TXFIFO_DisableEvent(handle->channel, event);
}

__STATIC_INLINE uint32_t SPI_MASTER_TXFIFO_GetEvent(const SPI_MASTER_t *const handle)
{
  XMC_ASSERT("SPI_MASTER_TXFIFO_GetEvent:handle NULL" , (handle != NULL));
  return XMC_USIC_CH_TXFIFO_GetEvent(handle->channel);
}

__STATIC_INLINE void SPI_MASTER_TXFIFO_ClearEvent(const SPI_MASTER_t *const handle, const uint32_t event)
{
  XMC_ASSERT("SPI_MASTER_TXFIFO_ClearEvent:handle NULL" , (handle != NULL));
  XMC_USIC_CH_TXFIFO_ClearEvent(handle->channel, event);
}

```

```
__STATIC_INLINE bool SPI_MASTER_IsTxFIFOFull(const SPI_MASTER_t* const handle)
{
  XMC_ASSERT("SPI_MASTER_IsTxFIFOFull:handle NULL", (handle != NULL))
  return XMC_USIC_CH_TXFIFO_IsFull(handle->channel);
}

__STATIC_INLINE void SPI_MASTER_RXFIFO_EnableEvent(const SPI_MASTER_t *const handle, const uint32_t event)
{
  XMC_ASSERT("SPI_MASTER_RXFIFO_EnableEvent:handle NULL" , (handle != NULL));
  XMC_USIC_CH_RXFIFO_EnableEvent(handle->channel, event);
}

__STATIC_INLINE void SPI_MASTER_RXFIFO_DisableEvent(const SPI_MASTER_t *const handle, const uint32_t event)
{
  XMC_ASSERT("SPI_MASTER_RXFIFO_DisableEvent:handle NULL" , (handle != NULL));
  XMC_USIC_CH_RXFIFO_DisableEvent(handle->channel, event);
}

__STATIC_INLINE uint32_t SPI_MASTER_RXFIFO_GetEvent(const SPI_MASTER_t *const handle)
{
  XMC_ASSERT("SPI_MASTER_RXFIFO_GetEvent:handle NULL" , (handle != NULL));
  return XMC_USIC_CH_RXFIFO_GetEvent(handle->channel);
}

```

```
__STATIC_INLINE void SPI_MASTER_RXFIFO_ClearEvent(const SPI_MASTER_t *const handle, const uint32_t event)
{
  XMC_ASSERT("SPI_MASTER_RXFIFO_ClearEvent:handle NULL" , (handle != NULL));
  XMC_USIC_CH_RXFIFO_ClearEvent(handle->channel, event);
}

__STATIC_INLINE bool SPI_MASTER_IsRxFIFOEmpty(const SPI_MASTER_t* const handle)
{
  XMC_ASSERT("SPI_MASTER_IsRxFIFOEmpty:handle NULL", (handle != NULL))
  return XMC_USIC_CH_RXFIFO_IsEmpty(handle->channel);
}

SPI_MASTER_STATUS_t SPI_MASTER_AbortTransmit(const SPI_MASTER_t *const handle);

SPI_MASTER_STATUS_t SPI_MASTER_AbortReceive(const SPI_MASTER_t *const handle);

#include "spi_master_extern.h"

#ifdef __cplusplus
}
#endif

#endif /* SPI_MASTER_H */
```

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_MASTE** 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.

```

/*******************************************************************************
 * HEADER FILES
 *******************************************************************************
#include "spi_master.h"


/*******************************************************************************
 * MACROS
 *******************************************************************************
#define SPI_MASTER_WORD_LENGTH_8_BIT (8U) /* This is used to check while incrementing the data index */
#define SPI_MASTER_2_BYTES_PER_WORD (2U) /* Word length is 16-bits */
#define SPI_MASTER_1_BYTE_PER_WORD (1U) /* Word length is 8-bits */

#define SPI_MASTER_RECEIVE_INDICATION_FLAG ((uint32_t)XMC_SPI_CH_STATUS_FLAG_RECEIVE_INDICATION | \
(uint32_t)XMC_SPI_CH_STATUS_FLAG_ALTERNATIVE_RECEIVE_INI

#define SPI_MASTER_FIFO_RECEIVE_INDICATION_FLAG ((uint32_t)XMC_USIC_CH_RXFIFO_EVENT_STANDARD | \
 (uint32_t)XMC_USIC_CH_RXFIFO_EVENT_ALTERNATE)

#define SPI_MASTER_RECEIVE_EVENT ((uint32_t)XMC_SPI_CH_EVENT_STANDARD_RECEIVE | \
 (uint32_t)XMC_SPI_CH_EVENT_ALTERNATIVE_RECEIVE)

#define SPI_MASTER_FIFO_RECEIVE_EVENT ((uint32_t)XMC_USIC_CH_RXFIFO_EVENT_CONF_STANDARD | \
(uint32_t)XMC_USIC_CH_RXFIFO_EVENT_CONF_ALTERNATE)
/*******************************************************************************
 * LOCAL DATA
 *******************************************************************************


/*******************************************************************************
 * LOCAL ROUTINES
 *******************************************************************************
#if (SPI_MASTER_INTERRUPT_TRANSMIT_MODE == 1U)
/* Transmit interrupt handler for the APP */
void SPI_MASTER_lTransmitHandler(const SPI_MASTER_t * const handle);
#endif

#if (SPI_MASTER_INTERRUPT_RECEIVE_MODE == 1U)
static SPI_MASTER_STATUS_t SPI_MASTER_lReceiveIRQ(const SPI_MASTER_t *const handle, uint32_t count);
/* This is used to reconfigure the FIFO settings dynamically */
static void SPI_MASTER_lReconfigureRxFIFO(const SPI_MASTER_t * const handle, uint32_t data_size);
/* Read data from FIFO */
static void SPI_MASTER_lFIFORead(const SPI_MASTER_t * const handle, const uint32_t bytes_per_word);
/* Receive interrupt handler for the APP */
void SPI_MASTER_lReceiveHandler(const SPI_MASTER_t * const handle);
```

```
#endif

#if(SPI_MASTER_DMA_RECEIVE_MODE == 1U)
static SPI_MASTER_STATUS_t SPI_MASTER_lReceiveDMA(const SPI_MASTER_t *const handle, uint32_t count);
#endif

#if(SPI_MASTER_DIRECT_TRANSMIT_MODE == 1U)
static SPI_MASTER_STATUS_t SPI_MASTER_lStartTransmitPolling(const SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count);
#endif

#if(SPI_MASTER_DIRECT_RECEIVE_MODE == 1U)
static SPI_MASTER_STATUS_t SPI_MASTER_lStartReceivePolling(const SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count);
static SPI_MASTER_STATUS_t SPI_MASTER_lReceivePolling(const SPI_MASTER_t *const handle, uint32_t count);
#endif

#ifdef SPI_MASTER_PARITY_ERROR
/* Protocol interrupt handler for the APP */
void SPI_MASTER_lProtocolHandler(const SPI_MASTER_t * const handle);
#endif

/* Flush RBUF0, RBUF1 */
static void SPI_MASTER_lStdRBUFFlush(XMC_USIC_CH_t *const channel);

/* This is used to reconfigure the registers while changing the SPI mode dynamically */
static void SPI_MASTER_lPortConfig(const SPI_MASTER_t* handle);
```

```
/* Set the mode of the port pin according to the configuration */
static void SPI_MASTER_lPortModeSet(const SPI_MASTER_t* handle);
/* Set the mode of the port pin as input */
static void SPI_MASTER_lPortModeReset(const SPI_MASTER_t* handle);
/* Returns whether mode change is valid or not */
static SPI_MASTER_STATUS_t SPI_MASTER_lValidateModeChange(const SPI_MASTER_t * handle, XMC_SPI_CH_MODE_t mode);
/*******************************************************************************
 * API IMPLEMENTATION
*******************************************************************************
/*
 * API to retrieve the version of the SPI_MASTER
 */
DAVE_APP_VERSION_t SPI_MASTER_GetAppVersion()
{
  DAVE_APP_VERSION_t version;

  version.major = SPI_MASTER_MAJOR_VERSION;
  version.minor = SPI_MASTER_MINOR_VERSION;
  version.patch = SPI_MASTER_PATCH_VERSION;

  return version;
}

/*
 * This function initializes the SPI channel, based on UI configuration.
 */
SPI_MASTER_STATUS_t SPI_MASTER_Init(SPI_MASTER_t* const handle)
{
  SPI_MASTER_STATUS_t status;

```

```
XMC_ASSERT("SPI_MASTER_Init:handle NULL" , (handle != NULL));

/* Configure the port registers and data input registers of SPI channel */
status = handle->config->fptr_spi_master_config();

return status;
}

/*
 * Change the SPI mode of communication.
 */
SPI_MASTER_STATUS_t SPI_MASTER_SetMode(SPI_MASTER_t* const handle,
 const XMC_SPI_CH_MODE_t mode)
{
 SPI_MASTER_STATUS_t status;

 XMC_ASSERT("SPI_MASTER_Configure:handle NULL" , (handle != NULL));

 status = SPI_MASTER_STATUS_SUCCESS;

 if ((false == handle->runtime->tx_busy) && (false == handle->runtime->rx_busy))
 {
 if (handle->runtime->spi_master_mode != mode)
 {
 status = SPI_MASTER_lValidateModeChange(handle, mode);

 if (SPI_MASTER_STATUS_SUCCESS == status)
 {
 handle->runtime->spi_master_mode = mode;

 /* This changes the operating mode and related settings */
 SPI_MASTER_lPortConfig(handle);
 }
 }
 }
 else
 {
 status = SPI_MASTER_STATUS_BUSY;
 }
 return status;
}

/*
 * Set the baud rate during runtime.
 */
SPI_MASTER_STATUS_t SPI_MASTER_SetBaudRate(SPI_MASTER_t* const handle, const uint32_t baud_rate)
{
 SPI_MASTER_STATUS_t status;

 if ((false == handle->runtime->tx_busy) && (false == handle->runtime->rx_busy))
 {
 /* Stops the SPI channel */
 status = (SPI_MASTER_STATUS_t)XMC_SPI_CH_Stop(handle->channel);

 if (SPI_MASTER_STATUS_SUCCESS == status)
 {
 /* Set all the pins as input */
 SPI_MASTER_lPortModeReset(handle);

 /* Update the new baud rate */
 status = (SPI_MASTER_STATUS_t)XMC_SPI_CH_SetBaudrate(handle->channel, baud_rate);

 if (SPI_MASTER_STATUS_SUCCESS == status)
```

```
{
/* Configure Leading/Trailing delay */
XMC_SPI_CH_SetSlaveSelectDelay(handle->channel, (uint32_t)handle->config->leading_trailing_delay);
}

/* Configure the clock polarity and clock delay */
XMC_SPI_CH_ConfigureShiftClockOutput(handle->channel,
handle->config->shift_clk_passive_level,
XMC_SPI_CH_BRG_SHIFT_CLOCK_OUTPUT_SCLK);
/* Start the SPI channel */
XMC_SPI_CH_Start(handle->channel);

/* Set the mode of the according the generated configuration */
SPI_MASTER_lPortModeSet(handle);
}
}
else
{
status = SPI_MASTER_STATUS_BUSY;
}

return status;
}

SPI_MASTER_STATUS_t SPI_MASTER_Transmit(const SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count)
{
SPI_MASTER_STATUS_t status;

status = SPI_MASTER_STATUS_FAILURE;

#if (SPI_MASTER_INTERRUPT_TRANSMIT_MODE == 1U)
if (handle->config->transmit_mode == SPI_MASTER_TRANSFER_MODE_INTERRUPT)
{
status = SPI_MASTER_StartTransmitIRQ(handle, dataptr, count);
```

```c
  }
#endif

#if (SPI_MASTER_DMA_TRANSMIT_MODE == 1U)
  if (handle->config->transmit_mode == SPI_MASTER_TRANSFER_MODE_DMA)
  {
    status = SPI_MASTER_StartTransmitDMA(handle, dataptr, count);
  }
#endif

#if (SPI_MASTER_DIRECT_TRANSMIT_MODE == 1U)
  if (handle->config->transmit_mode == SPI_MASTER_TRANSFER_MODE_DIRECT)
  {
    status = SPI_MASTER_lStartTransmitPolling(handle, dataptr, count);
  }
#endif

  return status;
}

SPI_MASTER_STATUS_t SPI_MASTER_Receive(const SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count)
{
  SPI_MASTER_STATUS_t status;

  status = SPI_MASTER_STATUS_FAILURE;

#if (SPI_MASTER_INTERRUPT_RECEIVE_MODE == 1U)
  if (handle->config->receive_mode == SPI_MASTER_TRANSFER_MODE_INTERRUPT)
  {
    status = SPI_MASTER_StartReceiveIRQ(handle, dataptr, count);
  }
#endif
```

```

#if (SPI_MASTER_DMA_RECEIVE_MODE == 1U)
  if (handle->config->receive_mode == SPI_MASTER_TRANSFER_MODE_DMA)
  {
  status = SPI_MASTER_StartReceiveDMA(handle, dataptr, count);
  }
#endif

#if (SPI_MASTER_DIRECT_RECEIVE_MODE == 1U)
  if (handle->config->receive_mode == SPI_MASTER_TRANSFER_MODE_DIRECT)
  {
  status = SPI_MASTER_lStartReceivePolling(handle, dataptr, count);
  }
#endif

  return status;
}

#if (SPI_MASTER_INTERRUPT_TRANSMIT_MODE == 1U)
/*
 * Transmit the number of data words specified.
 */
SPI_MASTER_STATUS_t SPI_MASTER_StartTransmitIRQ(const SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count)
{
  SPI_MASTER_STATUS_t status;
  uint32_t bytes_per_word = SPI_MASTER_1_BYTE_PER_WORD; /* This is to support the word length 8 and 16.
  Specify the number of bytes for the configured word length */
  SPI_MASTER_RUNTIME_t * runtime_handle;

  XMC_ASSERT("SPI_MASTER_StartTransmitIRQ:handle NULL" , (handle != NULL));

```

```
status = SPI_MASTER_STATUS_MODE_MISMATCH;
runtime_handle = handle->runtime;

if (handle->config->transmit_mode == SPI_MASTER_TRANSFER_MODE_INTERRUPT)
{
/* Check whether SPI channel is free or not */
if ((dataptr != NULL) && (count > 0U))
{
status = SPI_MASTER_STATUS_BUSY;
/*Check data pointer is valid or not*/
if (false == runtime_handle->tx_busy)
{
if (handle->runtime->word_length > SPI_MASTER_WORD_LENGTH_8_BIT)
{
bytes_per_word = SPI_MASTER_2_BYTES_PER_WORD; /* Word length is 16-bits */
}

/* Obtain the address of data, size of data */
runtime_handle->tx_data = dataptr;
runtime_handle->tx_data_count = (uint32_t)count << (bytes_per_word - 1U);
/* Initialize to first index and set the busy flag */
runtime_handle->tx_data_index = 0U;
runtime_handle->tx_busy = true;

/* Enable the transmit buffer event */
if ((uint32_t)handle->config->tx_fifo_size > 0U)
{
/* Flush the Transmit FIFO */
XMC_USIC_CH_TXFIFO_Flush(handle->channel);
XMC_USIC_CH_TXFIFO_EnableEvent(handle->channel, (uint32_t)XMC_USIC_CH_TXFIFO_EVENT_CONF_STANDARD);
}
else
{
XMC_USIC_CH_EnableEvent(handle->channel, (uint32_t)XMC_USIC_CH_EVENT_TRANSMIT_BUFFER);
}
XMC_SPI_CH_SetTransmitMode(handle->channel, runtime_handle->spi_master_mode);
status = SPI_MASTER_STATUS_SUCCESS;

/* Trigger the transmit buffer interrupt */
XMC_USIC_CH_TriggerServiceRequest(handle->channel, (uint32_t)handle->config->tx_sr);
}
}
else
{
status = SPI_MASTER_STATUS_BUFFER_INVALID;
}
}
return status;
}
#endif

#if(SPI_MASTER_DMA_TRANSMIT_MODE == 1U)
SPI_MASTER_STATUS_t SPI_MASTER_StartTransmitDMA(const SPI_MASTER_t *const handle, uint8_t *data_ptr, uint32_t block_size)
{
SPI_MASTER_STATUS_t status;
SPI_MASTER_RUNTIME_t * runtime_handle;
uint32_t dma_ctll;
uint32_t mode;

XMC_ASSERT("SPI_MASTER_StartTransmitDMA:handle NULL" , (handle != NULL));

status = SPI_MASTER_STATUS_MODE_MISMATCH;
runtime_handle = handle->runtime;

```

```
if (handle->config->transmit_mode == SPI_MASTER_TRANSFER_MODE_DMA)
{
/* Check whether SPI channel is free or not */
if (false == runtime_handle->tx_busy)
{
/* Check data pointer is valid or not */
if ((data_ptr != NULL) && (block_size > 0U) && (block_size <= SPI_MASTER_DMA_MAXCOUNT))
{
/* Obtain the address of data, size of data */
runtime_handle->tx_data_count = block_size;
/* Initialize to first index and set the busy flag */
runtime_handle->tx_data_index = 0U;
runtime_handle->tx_busy = true;

if (runtime_handle->tx_data_dummy == true)
{
dma_ctll = (uint32_t)handle->global_dma->dma->CH[handle->dma_ch_tx_number].CTLL;

dma_ctll = (uint32_t)(dma_ctll & (uint32_t)(~(GPDMA0_CH_CTLL_SINC_Msk))) |
((uint32_t)XMC_DMA_CH_ADDRESS_COUNT_MODE_NO_CHANGE << GPDMA0_CH_CTLL_SINC_Pos);

handle->global_dma->dma->CH[handle->dma_ch_tx_number].CTLL = dma_ctll;
mode = (uint32_t)((uint32_t)handle->runtime->spi_master_mode & 0xfffbU);
}
else
{
runtime_handle->tx_data = data_ptr;
dma_ctll = handle->global_dma->dma->CH[handle->dma_ch_tx_number].CTLL;

dma_ctll = (uint32_t)(dma_ctll & (~GPDMA0_CH_CTLL_SINC_Msk)) |
((uint32_t)XMC_DMA_CH_ADDRESS_COUNT_MODE_INCREMENT << GPDMA0_CH_CTLL_SINC_Pos);

handle->global_dma->dma->CH[handle->dma_ch_tx_number].CTLL = dma_ctll;
mode = (uint32_t)handle->runtime->spi_master_mode;
}

/* Enable transmit event generation */
XMC_SPI_CH_EnableEvent(handle->channel, (uint32_t)XMC_SPI_CH_EVENT_RECEIVE_START);

XMC_DMA_CH_SetBlockSize(handle->global_dma->dma, handle->dma_ch_tx_number, block_size);

XMC_DMA_CH_SetSourceAddress(handle->global_dma->dma, handle->dma_ch_tx_number, (uint32_t)runtime_handle->tx_data);

XMC_SPI_CH_SetTransmitMode(handle->channel, runtime_handle->spi_master_mode);

XMC_DMA_CH_SetDestinationAddress(handle->global_dma->dma,
handle->dma_ch_tx_number,
(uint32_t)&(handle->channel->TBUF[mode]));

status = SPI_MASTER_STATUS_SUCCESS;

XMC_DMA_CH_Enable(handle->global_dma->dma, handle->dma_ch_tx_number);
}
else
{
 status = SPI_MASTER_STATUS_BUFFER_INVALID;
 }
 }
 else
 {
 status = SPI_MASTER_STATUS_BUSY;
 }
 }

 return status;
}
#endif

#if (SPI_MASTER_DIRECT_TRANSMIT_MODE == 1U)
SPI_MASTER_STATUS_t SPI_MASTER_lStartTransmitPolling(const SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count)
{
 SPI_MASTER_STATUS_t status;
 uint16_t data;
 uint32_t bytes_per_word = SPI_MASTER_1_BYTE_PER_WORD;; /* This is to support the word length 8 and 16.
 Specify the number of bytes for the configured word length */
 SPI_MASTER_RUNTIME_t * runtime_handle;

 status = SPI_MASTER_STATUS_BUSY;
 runtime_handle = handle->runtime;
 data = 0U;

 XMC_ASSERT("SPI_MASTER_lStartTransmitPolling:handle NULL" , (handle != NULL));

 /* Check whether SPI channel is free or not */
 if ((dataptr != NULL) && (count > 0U))
 {
 /* Check data pointer is valid or not */
```

```
if (false == runtime_handle->tx_busy)
{
if (handle->runtime->word_length > SPI_MASTER_WORD_LENGTH_8_BIT)
{
bytes_per_word = SPI_MASTER_2_BYTES_PER_WORD; /* Word length is 16-bits */
}

runtime_handle->tx_busy = true;
/* Obtain the address of data, size of data */
runtime_handle->tx_data = dataptr;
runtime_handle->tx_data_count = (uint32_t)count << (bytes_per_word - 1U);
/* Initialize to first index and set the busy flag */
runtime_handle->tx_data_index = 0U;

XMC_SPI_CH_SetTransmitMode(handle->channel, runtime_handle->spi_master_mode);

if ((uint32_t)handle->config->tx_fifo_size > 0U)
{
/* Flush the Transmit FIFO */
XMC_USIC_CH_TXFIFO_Flush(handle->channel);

while (runtime_handle->tx_data_index < runtime_handle->tx_data_count)
{
while (XMC_USIC_CH_TXFIFO_IsFull(handle->channel) == true)
{
/* Wait until FIFO is having space for next entry */
}
if (runtime_handle->tx_data_dummy == true)
{
XMC_USIC_CH_TXFIFO_PutDataHPCMode(handle->channel, 0xFFFFU, (uint32_t)runtime_handle->spi_master_mode);
}
```

```
else
{
if(bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
{
data = *((uint16_t*)&runtime_handle->tx_data[runtime_handle->tx_data_index]);
}
else
{
data = runtime_handle->tx_data[runtime_handle->tx_data_index];
}
XMC_USIC_CH_TXFIFO_PutDataHPCMode(handle->channel, data, (uint32_t)runtime_handle->spi_master_mode);
}
(runtime_handle->tx_data_index)+= bytes_per_word;
}
}
else
{
do
{
while((uint32_t)XMC_USIC_CH_GetTransmitBufferStatus(handle->channel) == (uint32_t)XMC_USIC_CH_TBUF_STATUS_BUSY)
{
}

if (runtime_handle->tx_data_dummy == true)
{
XMC_USIC_CH_WriteToTBUFTCI(handle->channel, 0xFFFFU, runtime_handle->spi_master_mode);
}
else
{
if(bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
{
data = *((uint16_t*)&runtime_handle->tx_data[runtime_handle->tx_data_index]);
}
else
{
data = runtime_handle->tx_data[runtime_handle->tx_data_index];
}
XMC_USIC_CH_WriteToTBUFTCI(handle->channel, data, runtime_handle->spi_master_mode);
}
(runtime_handle->tx_data_index)+=bytes_per_word;

while ((XMC_SPI_CH_GetStatusFlag(handle->channel) & (uint32_t)XMC_SPI_CH_STATUS_FLAG_RECEIVER_START_INDICATI
== 0U)
{

}
XMC_SPI_CH_ClearStatusFlag(handle->channel, (uint32_t)XMC_SPI_CH_STATUS_FLAG_RECEIVER_START_INDICATI
} while(runtime_handle->tx_data_index < runtime_handle->tx_data_count);
}

while((uint32_t)XMC_USIC_CH_GetTransmitBufferStatus(handle->channel) == (uint32_t)XMC_USIC_CH_TBUF_STATUS_BUSY)
{
}

runtime_handle->tx_busy = false;
runtime_handle->tx_data_count = 0U;
runtime_handle->tx_data_index = 0U;
status = SPI_MASTER_STATUS_SUCCESS;
}
}
else
{
status = SPI_MASTER_STATUS_BUFFER_INVALID;
}
```

```
 runtime_handle->rx_data_dummy = true;
 return status;
}
#endif

#if (SPI_MASTER_DIRECT_RECEIVE_MODE == 1U)

SPI_MASTER_STATUS_t SPI_MASTER_lStartReceivePolling(const SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count)
{
 SPI_MASTER_STATUS_t status;

 SPI_MASTER_RUNTIME_t * runtime_handle;
 static uint8_t dummy_data[2] = {0xFFU, 0xFFU};

 XMC_ASSERT("SPI_MASTER_lStartReceivePolling:handle NULL" , (handle != NULL));

 status = SPI_MASTER_STATUS_BUSY;
 runtime_handle = handle->runtime;

 if ((dataptr != NULL) && (count > 0U))
 {
 /*Check data pointer is valid or not*/
 if ((false == runtime_handle->rx_busy) && (false == runtime_handle->tx_busy))
 {
 runtime_handle->rx_busy = true;
 runtime_handle->rx_data = dataptr;
 runtime_handle->tx_data = &dummy_data[0];
 runtime_handle->tx_data_dummy = true;
 runtime_handle->rx_data_dummy = false;

 status = SPI_MASTER_lReceivePolling(handle, count);

 runtime_handle->tx_data_dummy = false;
 runtime_handle->rx_busy = false;
 }
 }
 else
 {
 status = SPI_MASTER_STATUS_BUFFER_INVALID;
 }
 return status;
}


#endif

#if (SPI_MASTER_INTERRUPT_RECEIVE_MODE == 1U)
/*
 * Receive the specified the number of data words.
 */
SPI_MASTER_STATUS_t SPI_MASTER_StartReceiveIRQ(const SPI_MASTER_t *const handle, uint8_t* dataptr, uint32_t count)
{
 SPI_MASTER_STATUS_t status;
 SPI_MASTER_RUNTIME_t * runtime_handle;
 static uint8_t dummy_data[2] = {0xFFU, 0xFFU};

 XMC_ASSERT("SPI_MASTER_StartReceiveIRQ:handle NULL" , (handle != NULL));

 status = SPI_MASTER_STATUS_MODE_MISMATCH;
 runtime_handle = handle->runtime;

 if (handle->config->receive_mode == SPI_MASTER_TRANSFER_MODE_INTERRUPT)
 {
 status = SPI_MASTER_STATUS_BUSY;
 /* Check whether SPI channel is free or not */
 if ((dataptr != NULL) && (count > 0U))
 {
 /*Check data pointer is valid or not*/
 if ((false == runtime_handle->rx_busy) && (false == runtime_handle->tx_busy))
 {
 runtime_handle->rx_busy = true;
 runtime_handle->rx_data = dataptr;
 runtime_handle->tx_data = &dummy_data[0];
 runtime_handle->tx_data_count = count;
 runtime_handle->tx_data_dummy = true;
 runtime_handle->rx_data_dummy = false;

 status = SPI_MASTER_lReceiveIRQ(handle, count);

 }
 }
 else
 {
 status = SPI_MASTER_STATUS_BUFFER_INVALID;
 }
 }
 return status;
}
#endif

#if(SPI_MASTER_DMA_RECEIVE_MODE == 1U)
SPI_MASTER_STATUS_t SPI_MASTER_StartReceiveDMA(const SPI_MASTER_t *const handle, uint8_t *dataptr, uint32_t block_size)
{
 SPI_MASTER_STATUS_t status;
 SPI_MASTER_RUNTIME_t * runtime_handle;
 static uint8_t dummy_data[2] = {0xFFU, 0xFFU};

 XMC_ASSERT("SPI_MASTER_StartReceiveDMA:handle NULL" , (handle != NULL));

 status = SPI_MASTER_STATUS_MODE_MISMATCH;
 runtime_handle = handle->runtime;
```

```

if (handle->config->receive_mode == SPI_MASTER_TRANSFER_MODE_DMA)
{
status = SPI_MASTER_STATUS_BUSY;
/* Check whether SPI channel is free or not */
if ((false == runtime_handle->rx_busy) && (false == runtime_handle->tx_busy))
{
/* Check data pointer is valid or not */
if ((dataptr != NULL) && (block_size > 0U) && (block_size <= SPI_MASTER_DMA_MAXCOUNT))
{
runtime_handle->rx_busy = true;
runtime_handle->rx_data = dataptr;
runtime_handle->tx_data = &dummy_data[0];
runtime_handle->tx_data_count = block_size;
runtime_handle->tx_data_dummy = true;
runtime_handle->rx_data_dummy = false;

status = SPI_MASTER_lReceiveDMA(handle, block_size);
}
else
{
status = SPI_MASTER_STATUS_BUFFER_INVALID;
}
}
else
{
status = SPI_MASTER_STATUS_BUSY;
}
}
return status;
}
#endif

/*
```

```
 * Transmit and receive the data at the same time. This is supported for full duplex mode only.
 */
SPI_MASTER_STATUS_t SPI_MASTER_Transfer(const SPI_MASTER_t *const handle,
 uint8_t* tx_dataptr,
 uint8_t* rx_dataptr,
 uint32_t count)
{
 SPI_MASTER_STATUS_t status;
 SPI_MASTER_RUNTIME_t * runtime_handle;

 XMC_ASSERT("SPI_MASTER_Transfer:handle NULL" , (handle != NULL));

 status = SPI_MASTER_STATUS_BUSY;
 runtime_handle = handle->runtime;

 if (XMC_SPI_CH_MODE_STANDARD == runtime_handle->spi_master_mode)
 {
 /* Check whether SPI channel is free or not */
 if ((tx_dataptr != NULL) && (rx_dataptr != NULL) && (count > 0U))
 {
 /*Check data pointer is valid or not*/
 if ((false == runtime_handle->rx_busy) && (false == runtime_handle->tx_busy))
 {
 runtime_handle->rx_busy = true;
 runtime_handle->rx_data = rx_dataptr;
 runtime_handle->tx_data = tx_dataptr;
 runtime_handle->tx_data_count = count;
 runtime_handle->tx_data_dummy = false;
 runtime_handle->rx_data_dummy = false;

#if (SPI_MASTER_INTERRUPT_RECEIVE_MODE == 1U)
 if (handle->config->receive_mode ==
SPI_MASTER_TRANSFER_MODE_INTERRUPT)
 {
 status = SPI_MASTER_lReceiveIRQ(handle, count);
 }
#endif
#if (SPI_MASTER_DMA_RECEIVE_MODE == 1U)
 if (handle->config->receive_mode ==
SPI_MASTER_TRANSFER_MODE_DMA)
 {
 status = SPI_MASTER_lReceiveDMA(handle, count);
 }
#endif
#if (SPI_MASTER_DIRECT_RECEIVE_MODE == 1U)
 if (handle->config->receive_mode ==
SPI_MASTER_TRANSFER_MODE_DIRECT)
 {
 status = SPI_MASTER_lReceivePolling(handle, count);
 runtime_handle->rx_busy = false;
 }
#endif
 }
 }
 else
 {
 status = SPI_MASTER_STATUS_BUFFER_INVALID;
 }
 }
 else
 {
 status = SPI_MASTER_STATUS_FAILURE;
 }

 return status;
}


/*
```

```
 * Aborts the ongoing data reception.
 */
SPI_MASTER_STATUS_t SPI_MASTER_AbortReceive(const SPI_MASTER_t *const handle)
{
 SPI_MASTER_STATUS_t status;

 status = SPI_MASTER_STATUS_FAILURE;

 if ((handle->config->receive_mode != SPI_MASTER_TRANSFER_MODE_DIRECT) && (handle->runtime->rx_busy))
 {
 /* Abort if any ongoing transmission w.r.t reception. */
 status = SPI_MASTER_AbortTransmit(handle);

 if (status == SPI_MASTER_STATUS_SUCCESS)
 {
 /* Reset the user buffer pointer to null */
 handle->runtime->rx_busy = false;
 handle->runtime->rx_data = NULL;
 handle->runtime->tx_data_dummy = false;
 /* Disable the receive interrupts */
 if ((uint32_t)handle->config->rx_fifo_size > 0U)
 {
 XMC_USIC_CH_RXFIFO_DisableEvent(handle->channel, (uint32_t)SPI_MASTER_FIFO_RECEIVE_EVENT);
 }
 else
 {
#if (SPI_MASTER_DMA_RECEIVE_MODE == 1U)
 if (handle->config->receive_mode == SPI_MASTER_TRANSFER_MODE_DMA)
 {
 /* Disable the receive event */
 if(XMC_DMA_CH_IsEnabled(handle->global_dma->dma, handle->dma_ch_rx_number))
 {
 XMC_DMA_CH_Disable(handle->global_dma->dma, handle->dma_ch_rx_number);
 while(XMC_DMA_CH_IsEnabled(handle->global_dma->dma, handle->dma_ch_rx_number)==true)
 {
 }
 XMC_SPI_CH_DisableEvent(handle->channel,
 (uint32_t)((uint32_t)XMC_USIC_CH_EVENT_STANDARD_RECEIVE | (uint32_t)XMC_USIC_CH_EVENT_ALTERNATIVE_RECEIVE));
 }
 }
 else
#endif
 {
 XMC_SPI_CH_DisableEvent(handle->channel,
 (uint32_t)((uint32_t)XMC_USIC_CH_EVENT_STANDARD_RECEIVE | (uint32_t)XMC_USIC_CH_EVENT_ALTERNATIVE_RECEIVE));
 }
 }
 status = SPI_MASTER_STATUS_SUCCESS;
 }
 else
 {
 status = SPI_MASTER_STATUS_FAILURE;
 }
 }
 return status;
}

/*
 * Aborts the ongoing data transmission.
 */
SPI_MASTER_STATUS_t SPI_MASTER_AbortTransmit(const SPI_MASTER_t *const handle)
```

```
{
SPI_MASTER_STATUS_t status;

status = SPI_MASTER_STATUS_FAILURE;

if ((handle->config->transmit_mode != SPI_MASTER_TRANSFER_MODE_DIRECT) && (handle->runtime->tx_busy))
{
/*Reset the user buffer pointer to null*/
handle->runtime->tx_busy = false;
handle->runtime->tx_data = NULL;
handle->runtime->tx_data_dummy = false;
/*Disable the transmit interrupts*/
if ((uint32_t)handle->config->tx_fifo_size > 0U)
{
/*Disable the transmit FIFO event*/
XMC_USIC_CH_TXFIFO_DisableEvent(handle->channel, (uint32_t)XMC_USIC_CH_TXFIFO_EVENT_CONF_STANDARD);
XMC_USIC_CH_TXFIFO_Flush(handle->channel);
}
else
{
#if (SPI_MASTER_DMA_TRANSMIT_MODE == 1U)
if(handle->config->transmit_mode == SPI_MASTER_TRANSFER_MODE_DMA)
{
/*Disable the standard transmit event*/
if(XMC_DMA_CH_IsEnabled(handle->global_dma->dma, handle->dma_ch_tx_number))
{
XMC_DMA_CH_Disable(handle->global_dma->dma, handle->dma_ch_tx_number);
while(XMC_DMA_CH_IsEnabled(handle->global_dma->dma, handle->dma_ch_tx_number)==true)
{
}
```

```
 XMC_SPI_CH_DisableEvent(handle->channel, (uint32_t)XMC_USIC_CH_EVENT_TRANSMIT_BUFFER);
 }
 }
 else
#endif
 {
 /*Disable the standard transmit event*/
 XMC_SPI_CH_DisableEvent(handle->channel, (uint32_t)XMC_USIC_CH_EVENT_TRANSMIT_BUFFER);
 }
 }
 status = SPI_MASTER_STATUS_SUCCESS;
 }
 return status;
}
/*****************************************************************************
** Private API definitions **
******************************************************************************
#if(SPI_MASTER_INTERRUPT_TRANSMIT_MODE == 1U)
/*
 * Transmit interrupt handler for the APP.
 * This is a common interrupt handling function called for different instances of the APP.
 *
 */
void SPI_MASTER_lTransmitHandler(const SPI_MASTER_t * const handle)
{
 uint16_t data; /* Data to be loaded into the TBUF */
 uint32_t bytes_per_word = SPI_MASTER_1_BYTE_PER_WORD; /* This is to support the word length 8 and 16.*/
 SPI_MASTER_RUNTIME_t * runtime_handle = handle->runtime;

 if (handle->runtime->word_length > SPI_MASTER_WORD_LENGTH_8_BIT)
 {
```

```
bytes_per_word = SPI_MASTER_2_BYTES_PER_WORD; /* Word length is 16-bits */
}

if (runtime_handle->tx_data_index < runtime_handle->tx_data_count)
{
data = 0U;
/*When Transmit FIFO is enabled*/
if ((uint32_t)handle->config->tx_fifo_size > 0U)
{
/*Fill the transmit FIFO */
while (XMC_USIC_CH_TXFIFO_IsFull(handle->channel) == false)
{
if (runtime_handle->tx_data_index < runtime_handle->tx_data_count)
{
/*Load the FIFO byte by byte till either FIFO is full or all data is loaded*/
if (runtime_handle->tx_data_dummy == true)
{
XMC_USIC_CH_TXFIFO_PutDataHPCMode(handle->channel, 0xFFFFU, (uint32_t)runtime_handle->spi_master_mode);
}
else
{
if(bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
{
data = *((uint16_t*)&runtime_handle->tx_data[runtime_handle->tx_data_index]);
}
else
{
data = runtime_handle->tx_data[runtime_handle->tx_data_index];
}
XMC_USIC_CH_TXFIFO_PutDataHPCMode(handle->channel, data, (uint32_t)runtime_handle->spi_master_mode);
}
(runtime_handle->tx_data_index)+= bytes_per_word;
}
else
{
break;
}
}
}
else/*When Transmit FIFO is disabled*/
{
if (runtime_handle->tx_data_dummy == true)
{
XMC_USIC_CH_WriteToTBUFTCI(handle->channel, 0xFFFFU, (uint32_t)runtime_handle->spi_master_mode);
}
else
{
if(bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
{
data = *((uint16_t*)&runtime_handle->tx_data[runtime_handle->tx_data_index]);
}
else
{
data = runtime_handle->tx_data[runtime_handle->tx_data_index];
}
XMC_USIC_CH_WriteToTBUFTCI(handle->channel, data, (uint32_t)runtime_handle->spi_master_mode);
}
(runtime_handle->tx_data_index)+= bytes_per_word;
}
}
else
{
if (XMC_USIC_CH_TXFIFO_IsEmpty(handle->channel) == true)
{
```

```
/* Clear the flag */
if ((uint32_t)handle->config->tx_fifo_size > 0U)
{
/* Clear the transmit FIFO event */
XMC_USIC_CH_TXFIFO_DisableEvent(handle->channel, (uint32_t)XMC_USIC_CH_TXFIFO_EVENT_CONF_STANDARD);
}
else
{
/* Clear the standard transmit event */
XMC_USIC_CH_DisableEvent(handle->channel, (uint32_t)XMC_USIC_CH_EVENT_TRANSMIT_BUFFER);
}

/* Wait for the transmit buffer to be free to ensure that all data is transmitted */
while (XMC_USIC_CH_GetTransmitBufferStatus(handle->channel) == XMC_USIC_CH_TBUF_STATUS_BUSY)
{

}

/* All data is transmitted */
runtime_handle->tx_busy = false;
runtime_handle->tx_data = NULL;

if ((handle->config->tx_cbhandler != NULL) && (runtime_handle->rx_busy == false))
{
/* Execute the callback function provided in the SPI_MASTER APP UI */
handle->config->tx_cbhandler();
}
}
}
}
#endif
```

```

#if (SPI_MASTER_INTERRUPT_RECEIVE_MODE == 1U)

SPI_MASTER_STATUS_t SPI_MASTER_lReceiveIRQ(const SPI_MASTER_t *const handle, uint32_t count)
{

  SPI_MASTER_STATUS_t status;
  SPI_MASTER_RUNTIME_t * runtime_handle;
  uint32_t bytes_per_word = SPI_MASTER_1_BYTE_PER_WORD;; /* This is to support the word length 8 and 16.
  Specify the number of bytes for the configured word length*/

  runtime_handle = handle->runtime;
  runtime_handle->rx_data_index = 0U;

  if (handle->runtime->word_length > SPI_MASTER_WORD_LENGTH_8_BIT)
  {
  bytes_per_word = SPI_MASTER_2_BYTES_PER_WORD; /* Word length is 16-bits */
  }

  /* If no active reception in progress, obtain the address of data buffer and number of data bytes to be received */
  runtime_handle->rx_data_count = (uint32_t)count << (bytes_per_word - 1U);

  /* Check if FIFO is enabled */
  if ((uint32_t)handle->config->rx_fifo_size > 0U)
  {
  /* Clear the receive FIFO */
  XMC_USIC_CH_RXFIFO_Flush(handle->channel);
  SPI_MASTER_lStdRBUFFlush(handle->channel);

  /* Configure the FIFO trigger limit based on the required data
size */
  SPI_MASTER_lReconfigureRxFIFO(handle, runtime_handle->rx_data_count);

  /* Enable the receive FIFO events */
  XMC_USIC_CH_RXFIFO_EnableEvent(handle->channel, (uint32_t)SPI_MASTER_FIFO_RECEIVE_EVENT);
  }
  else
  {
  /* Flush the RBUF0 and RBUF1 */
  SPI_MASTER_lStdRBUFFlush(handle->channel);

  /* Enable the standard receive events */
  XMC_USIC_CH_EnableEvent(handle->channel, (uint32_t)SPI_MASTER_RECEIVE_EVENT);
  }
  /* Call the transmit, to receive the data synchronously */
  status = SPI_MASTER_Transmit(handle, runtime_handle->tx_data, runtime_handle->tx_data_count);

  return status;
}

/*
 * Receive interrupt handler for the APP.
 * This is a common interrupt handling function for different instances of the SPI_MASTER APP.
 */
void SPI_MASTER_lReceiveHandler(const SPI_MASTER_t * const handle)
{
  uint16_t data; /* Data to be loaded into the TBUF */
  uint32_t bytes_per_word = SPI_MASTER_1_BYTE_PER_WORD; /* This is to support the word length 8 and 16. */
  SPI_MASTER_RUNTIME_t * runtime_handle = handle->runtime;
```

```

data = 0U;

if (handle->runtime->word_length > SPI_MASTER_WORD_LENGTH_8_BIT)
{
bytes_per_word = SPI_MASTER_2_BYTES_PER_WORD; /* Word length is 16-bits */
}

if ((uint32_t)handle->config->rx_fifo_size > 0U)
{
/* read the FIFO */
SPI_MASTER_lFIFORead(handle, bytes_per_word);
/* Reconfigure the RXFIFO trigger limit based on pending receive bytes */
SPI_MASTER_lReconfigureRxFIFO(handle, (uint32_t)(runtime_handle->rx_data_count - runtime_handle->rx_data_index));
}
else
{
/* When RxFIFO is disabled */
if ((XMC_USIC_CH_GetReceiveBufferStatus(handle->channel) & (uint32_t)XMC_USIC_CH_RBUF_STATUS_DATA_VALID0) != 0U )
{
if (runtime_handle->rx_data_index < runtime_handle->rx_data_count)
{
data = XMC_SPI_CH_GetReceivedData(handle->channel);

runtime_handle->rx_data[runtime_handle->rx_data_index] = (uint8_t)data;

if (bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
{
runtime_handle->rx_data[runtime_handle->rx_data_index + 1U] = (uint8_t)((uint16_t)data >> 8);
}

(runtime_handle->rx_data_index)+= bytes_per_word;
}
}
if ((XMC_USIC_CH_GetReceiveBufferStatus(handle->channel) & (uint32_t)XMC_USIC_CH_RBUF_STATUS_DATA_VALID1) != 0U)
{
if (runtime_handle->rx_data_index < runtime_handle->rx_data_count)
{
data = XMC_SPI_CH_GetReceivedData(handle->channel);

runtime_handle->rx_data[runtime_handle->rx_data_index] = (uint8_t)data;

if (bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
{
runtime_handle->rx_data[runtime_handle->rx_data_index + 1U] = (uint8_t)((uint16_t)data >> 8);
}

(runtime_handle->rx_data_index)+= bytes_per_word;
}
}

if (runtime_handle->rx_data_index == runtime_handle->rx_data_count)
{
/* Disable both standard receive and alternative receive FIFO events */
if ((uint32_t)handle->config->rx_fifo_size > 0U)
{
/* Enable the receive FIFO events */
XMC_USIC_CH_RXFIFO_DisableEvent(handle->channel, (uint32_t)SPI_MASTER_FIFO_RECEIVE_EVENT);
}
```

```
 else
 {
 XMC_SPI_CH_DisableEvent(handle->channel, (uint32_t)SPI_MASTER_RECEIVE_EVENT);
 }
 /* Reception complete */
 runtime_handle->rx_busy = false;
 runtime_handle->tx_data_dummy = false;
 runtime_handle->rx_data_dummy = true;
 runtime_handle->rx_data = NULL;

 if (handle->config->rx_cbhandler != NULL)
 {
 /* Execute the 'End of reception' callback function */
 handle->config->rx_cbhandler();
 }
 }
 }
}

/*
 * Read the data from FIFO until it becomes empty.
 */
void SPI_MASTER_lFIFORead(const SPI_MASTER_t * const handle, const uint32_t bytes_per_word)
{
 SPI_MASTER_RUNTIME_t * runtime_handle;
 uint16_t data;

 runtime_handle = handle->runtime;
 data = 0U;

 /* When Receive FIFO is enabled*/
 while (XMC_USIC_CH_RXFIFO_IsEmpty(handle->channel) == false)
 {
 if (runtime_handle->rx_data_index < runtime_handle-
>rx_data_count)
 {
 data = XMC_SPI_CH_GetReceivedData(handle->channel);
 runtime_handle->rx_data[runtime_handle->rx_data_index] = (uint8_t)data;

 if (bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
 {
 runtime_handle->rx_data[runtime_handle->rx_data_index + 1U] = (uint8_t)((uint16_t)data >> 8);
 }
 (runtime_handle->rx_data_index)+= bytes_per_word;
 }

 if (runtime_handle->rx_data_index == runtime_handle->rx_data_count)
 {
 /*Reception complete*/
 runtime_handle->rx_busy = false;
 runtime_handle->tx_data_dummy = false;
 /*Disable both standard receive and alternative receive FIFO events*/
 XMC_USIC_CH_RXFIFO_DisableEvent(handle->channel, (uint32_t)SPI_MASTER_FIFO_RECEIVE_EVENT);
 if (handle->config->rx_cbhandler != NULL)
 {
 /* Execute the 'End of reception' callback function */
 handle->config->rx_cbhandler();
 }
 break;
 }
 }
}

/*
 * This function configures the FIFO settings
 */
```

```
static void SPI_MASTER_lReconfigureRxFIFO(const SPI_MASTER_t * const handle, uint32_t data_size)
{
 uint32_t fifo_size;
 uint32_t ret_limit_val;

 if (((uint32_t)handle->config->rx_fifo_size > 0U) && (data_size > 0U))
 {
 fifo_size = (uint32_t)0x01 << handle->config->rx_fifo_size;

 if (handle->runtime->word_length > SPI_MASTER_WORD_LENGTH_8_BIT)
 {
 /* Data size is divided by 2, to change the trigger limit according the word length */
 data_size = (uint32_t)data_size >> 1U;
 }

 /*If data size is more than FIFO size, configure the limit to the FIFO size*/
 if (data_size < (fifo_size >> 1))
 {
 ret_limit_val = data_size - 1U;
 }
 else
 {
 ret_limit_val = fifo_size >> 1;
 }

 /*Set the limit value*/
 XMC_USIC_CH_RXFIFO_SetSizeTriggerLimit(handle->channel, handle->config->rx_fifo_size, ret_limit_val);
 }
}
#endif

```

```
#if (SPI_MASTER_DIRECT_RECEIVE_MODE == 1U)
SPI_MASTER_STATUS_t SPI_MASTER_lReceivePolling(const SPI_MASTER_t *const handle, uint32_t count)
{
 SPI_MASTER_RUNTIME_t * runtime_handle;
 uint32_t bytes_per_word = SPI_MASTER_1_BYTE_PER_WORD; /* This is to support the word length 8 and 16.
 Specify the number of bytes for the configured word length */
 uint16_t data;

 runtime_handle = handle->runtime;
 data = 0U;
 runtime_handle->rx_data_index = 0U;
 runtime_handle->tx_data_index = 0U;

 if (handle->runtime->word_length > SPI_MASTER_WORD_LENGTH_8_BIT)
 {
 bytes_per_word = SPI_MASTER_2_BYTES_PER_WORD; /* Word length is 16-bits */
 }

 runtime_handle->rx_data_count = (uint32_t)count << (bytes_per_word - 1U);

 XMC_SPI_CH_SetTransmitMode(handle->channel, runtime_handle->spi_master_mode);

 /* Check if FIFO is enabled */
 if ((uint32_t)handle->config->rx_fifo_size > 0U)
 {
 /* Clear the receive FIFO */
 XMC_USIC_CH_RXFIFO_Flush(handle->channel);
 SPI_MASTER_lStdRBUFFlush(handle->channel);

 if (runtime_handle->tx_data_dummy == true)
```

```
{
XMC_USIC_CH_TXFIFO_PutDataHPCMode(handle->channel, 0xFFFFU, (uint32_t)runtime_handle->spi_master_mode);
}
else
{
if(bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
{
data = *((uint16_t*)&runtime_handle->tx_data[runtime_handle->tx_data_index]);
}
else
{
data = runtime_handle->tx_data[runtime_handle->tx_data_index];
}
XMC_USIC_CH_TXFIFO_PutDataHPCMode(handle->channel, data, (uint32_t)runtime_handle->spi_master_mode);
}

(runtime_handle->tx_data_index)+= bytes_per_word;


while (runtime_handle->tx_data_index < runtime_handle->rx_data_count)
{
if (runtime_handle->tx_data_dummy == true)
{
XMC_USIC_CH_TXFIFO_PutDataHPCMode(handle->channel, 0xFFFFU, (uint32_t)runtime_handle->spi_master_mode);
}
else
{
if(bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
{
data = *((uint16_t*)&runtime_handle->tx_data[runtime_handle->tx_data_index]);
}
else
{
data = runtime_handle->tx_data[runtime_handle->tx_data_index];
}
XMC_USIC_CH_TXFIFO_PutDataHPCMode(handle->channel, data, (uint32_t)runtime_handle->spi_master_mode);
}

while(XMC_USIC_CH_RXFIFO_IsEmpty(handle->channel) == true)
{

}

data = XMC_SPI_CH_GetReceivedData(handle->channel);

runtime_handle->rx_data[runtime_handle->rx_data_index] = (uint8_t)data;

if (bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
{
runtime_handle->rx_data[runtime_handle->rx_data_index + 1U] = (uint8_t)((uint16_t)data >> 8);
}

(runtime_handle->rx_data_index)+= bytes_per_word;
(runtime_handle->tx_data_index)+= bytes_per_word;
}

while(XMC_USIC_CH_RXFIFO_IsEmpty(handle->channel) == true)
{

}

```

```
data = XMC_SPI_CH_GetReceivedData(handle->channel);

runtime_handle->rx_data[runtime_handle->rx_data_index] = (uint8_t)data;

if (bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
{
runtime_handle->rx_data[runtime_handle->rx_data_index + 1U] = (uint8_t)((uint16_t)data >> 8);
}

XMC_USIC_CH_RXFIFO_ClearEvent(handle->channel, SPI_MASTER_FIFO_RECEIVE_INDICATION_FLAG);
}
else
{
/* Flush the RBUF0 and RBUF1 */
SPI_MASTER_lStdRBUFFlush(handle->channel);


while((uint32_t)XMC_USIC_CH_GetTransmitBufferStatus(handle->channel) == (uint32_t)XMC_USIC_CH_TBUF_STATUS_BUSY)
{
}

if (runtime_handle->tx_data_dummy == true)
{
XMC_USIC_CH_WriteToTBUFTCI(handle->channel, 0xFFFFU, (uint32_t)runtime_handle->spi_master_mode);
}
else
{
if(bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
{
data = *((uint16_t*)&runtime_handle->tx_data[runtime_handle->tx_data_index]);
}
```

```
else
{
data = runtime_handle->tx_data[runtime_handle->tx_data_index];
}
XMC_USIC_CH_WriteToTBUFTCI(handle->channel, data, (uint32_t)runtime_handle->spi_master_mode);
}

(runtime_handle->tx_data_index)+= bytes_per_word;

while (runtime_handle->tx_data_index < runtime_handle->rx_data_count)
{
while((uint32_t)XMC_USIC_CH_GetTransmitBufferStatus(handle->channel) == (uint32_t)XMC_USIC_CH_TBUF_STATUS_BUSY)
{

}

if (runtime_handle->tx_data_dummy == true)
{
XMC_USIC_CH_WriteToTBUFTCI(handle->channel, 0xFFFFU, (uint32_t)runtime_handle->spi_master_mode);
}
else
{
if(bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
{
data = *((uint16_t*)&runtime_handle->tx_data[runtime_handle->tx_data_index]);
}
else
{
data = runtime_handle->tx_data[runtime_handle->tx_data_index];
}
XMC_USIC_CH_WriteToTBUFTCI(handle->channel, data, (uint32_t)runtime_handle->spi_master_mode);
}

while (XMC_USIC_CH_GetReceiveBufferStatus(handle->channel) == 0U)
{

}

data = XMC_SPI_CH_GetReceivedData(handle->channel);

runtime_handle->rx_data[runtime_handle->rx_data_index] = (uint8_t)data;

if (bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
{
runtime_handle->rx_data[runtime_handle->rx_data_index + 1U] = (uint8_t)((uint16_t)data >> 8);
}

(runtime_handle->rx_data_index)+= bytes_per_word;
(runtime_handle->tx_data_index)+= bytes_per_word;

XMC_SPI_CH_ClearStatusFlag(handle->channel, SPI_MASTER_RECEIVE_INDICATION_FLAG);
}

while (XMC_USIC_CH_GetReceiveBufferStatus(handle->channel) == 0U)
{

}

data = XMC_SPI_CH_GetReceivedData(handle->channel);

```

```
  runtime_handle->rx_data[runtime_handle->rx_data_index] = (uint8_t)data;

  if (bytes_per_word == SPI_MASTER_2_BYTES_PER_WORD)
  {
  runtime_handle->rx_data[runtime_handle->rx_data_index + 1U] = (uint8_t)((uint16_t)data >> 8);
  }

  XMC_SPI_CH_ClearStatusFlag(handle->channel, SPI_MASTER_RECEIVE_INDICATION_FLAG);
  }
  runtime_handle->rx_data_count = 0U;
  runtime_handle->rx_data_index = 0U;
  runtime_handle->tx_data_index = 0U;

  return SPI_MASTER_STATUS_SUCCESS;
}
#endif

#if (SPI_MASTER_DMA_RECEIVE_MODE == 1U)
SPI_MASTER_STATUS_t SPI_MASTER_lReceiveDMA(const SPI_MASTER_t *const handle, uint32_t block_size)
{
  SPI_MASTER_STATUS_t status;
  SPI_MASTER_RUNTIME_t * runtime_handle;

  runtime_handle = handle->runtime;
  runtime_handle->rx_data_index = 0U;
  runtime_handle->rx_data_count = (uint32_t)block_size;

  SPI_MASTER_lStdRBUFFlush(handle->channel);

  XMC_SPI_CH_EnableEvent(handle->channel, (uint32_t)SPI_MASTER_RECEIVE_EVENT);

  XMC_DMA_CH_SetBlockSize(handle->global_dma->dma,
handle->dma_ch_rx_number, runtime_handle->rx_data_count);

  XMC_DMA_CH_SetSourceAddress(handle->global_dma->dma,
  handle->dma_ch_rx_number,
  (uint32_t)&(handle->channel->RBUF));

  XMC_DMA_CH_SetDestinationAddress(handle->global_dma->dma, handle->dma_ch_rx_number, (uint32_t)runtime_handle->rx_data);

  status = SPI_MASTER_STATUS_SUCCESS;

  XMC_DMA_CH_Enable(handle->global_dma->dma, handle->dma_ch_rx_number);

  /* Call the transmit, to receive the data synchronously */
  status = SPI_MASTER_Transmit(handle, runtime_handle->tx_data, runtime_handle->tx_data_count);

  return status;
}
#endif

/*
  * Clears the receive buffers
  */
static void SPI_MASTER_lStdRBUFFlush(XMC_USIC_CH_t *const channel)
{
  /* Clear RBF0 */
  (void)XMC_SPI_CH_GetReceivedData(channel);
  /* Clear RBF1 */
  (void)XMC_SPI_CH_GetReceivedData(channel);
}

#if (SPI_MASTER_PARITY_ERROR == 1U)
/*
```

```
 * Protocol interrupt handling function.
 * The function is common for different instances of the SPI_MASTER APP.
 */
void SPI_MASTER_lProtocolHandler(const SPI_MASTER_t * const handle)
{
 uint32_t psr_status;

 psr_status = XMC_SPI_CH_GetStatusFlag(handle->channel);

 /*Check for Parity detection error */
 if ((handle->config->parity_cbhandler != NULL) && \
 (psr_status & (uint32_t)XMC_SPI_CH_STATUS_FLAG_PARITY_ERROR_EVENT_DE
 {
 handle->config->parity_cbhandler();
 }
}
#endif

/*
 * This is used to reconfigure the registers while changing the SPI mode dynamically
 */
static void SPI_MASTER_lPortConfig(const SPI_MASTER_t* handle)
{
 switch (handle->runtime->spi_master_mode)
 {
 case XMC_SPI_CH_MODE_STANDARD:
 /* Configure the data input line selected */
 XMC_SPI_CH_SetInputSource(handle->channel, XMC_SPI_CH_INPUT_DIN0, (uint8_t)(handle->runtime->dx0_input));
 /* Configure the pin as input */
 XMC_GPIO_SetMode(handle->config->mosi_1_pin->port, handle->config->mosi_1_pin->pin,
 XMC_GPIO_MODE_INPUT_TRISTATE);
 /* Disable the HW control of the PINs */
 XMC_GPIO_SetHardwareControl(handle->config->mosi_0_pin->port,
 handle->config->mosi_0_pin->pin,
 XMC_GPIO_HWCTRL_DISABLED);
 XMC_GPIO_SetHardwareControl(handle->config->mosi_1_pin->port,
 handle->config->mosi_1_pin->pin,
 XMC_GPIO_HWCTRL_DISABLED);

 break;

 case XMC_SPI_CH_MODE_STANDARD_HALFDUPLEX:
 /* Configure the data input line selected */
 XMC_SPI_CH_SetInputSource(handle->channel, XMC_SPI_CH_INPUT_DIN0, (uint8_t)(handle->runtime->dx0_input_half_duplex));
 /* Disable the HW control of the PINs */
 XMC_GPIO_SetHardwareControl(handle->config->mosi_0_pin->port,
 handle->config->mosi_0_pin->pin,
 XMC_GPIO_HWCTRL_DISABLED);
 break;

 case XMC_SPI_CH_MODE_DUAL:
 case XMC_SPI_CH_MODE_QUAD:
 /* Configure the data input line for loopback mode */
 XMC_SPI_CH_SetInputSource(handle->channel, XMC_SPI_CH_INPUT_DIN0, (uint8_t)SPI_MASTER_INPUT_G);
 /* Configure the pin as input */
 XMC_GPIO_SetMode(handle->config->mosi_1_pin->port,
 handle->config->mosi_1_pin->pin,
 handle->config->mosi_1_pin_config->port_config.mode);

 /* Configure the Hardware control mode selected for the pin */
 XMC_GPIO_SetHardwareControl(handle->config->mosi_0_pin-
>port,
 handle->config->mosi_0_pin->pin,
 handle->config->mosi_0_pin_config->hw_control);
 XMC_GPIO_SetHardwareControl(handle->config->mosi_1_pin->port,
 handle->config->mosi_1_pin->pin,
 handle->config->mosi_1_pin_config->hw_control);
 break;

 default:
 break;
 }
}

/*
 * This is used to reassign the mode for ports after updating the baud rate
 */
static void SPI_MASTER_lPortModeSet(const SPI_MASTER_t* handle)
{
 uint32_t ss_line;

 /* Configure the ports with actual mode */
 for (ss_line = 0U; ss_line < handle->config->slave_select_lines; ss_line++)
 {
 XMC_GPIO_SetMode(handle->config->slave_select_pin[ss_line]->port,
 handle->config->slave_select_pin[ss_line]->pin,
 handle->config->slave_select_pin_config[ss_line]->port_config.mode);
 }

 XMC_GPIO_SetMode(handle->config->sclk_out_pin->port,
 handle->config->sclk_out_pin->pin,
 handle->config->sclk_out_pin_config->port_config.mode);
```

```

switch (handle->runtime->spi_master_mode)
{
case XMC_SPI_CH_MODE_STANDARD:
case XMC_SPI_CH_MODE_STANDARD_HALFDUPLEX:
XMC_GPIO_SetMode(handle->config->mosi_0_pin->port,
handle->config->mosi_0_pin->pin,
handle->config->mosi_0_pin_config->port_config.mode);
break;

case XMC_SPI_CH_MODE_DUAL:
XMC_GPIO_SetMode(handle->config->mosi_0_pin->port,
handle->config->mosi_0_pin->pin,
handle->config->mosi_0_pin_config->port_config.mode);
XMC_GPIO_SetMode(handle->config->mosi_1_pin->port,
handle->config->mosi_1_pin->pin,
handle->config->mosi_1_pin_config->port_config.mode);
break;

case XMC_SPI_CH_MODE_QUAD:
XMC_GPIO_SetMode(handle->config->mosi_0_pin->port,
handle->config->mosi_0_pin->pin,
handle->config->mosi_0_pin_config->port_config.mode);
XMC_GPIO_SetMode(handle->config->mosi_1_pin->port,
handle->config->mosi_1_pin->pin,
handle->config->mosi_1_pin_config->port_config.mode);
XMC_GPIO_SetMode(handle->config->mosi_2_pin->port,
handle->config->mosi_2_pin->pin,
handle->config->mosi_2_pin_config->port_config.mode);
XMC_GPIO_SetMode(handle->config->mosi_3_pin->port,
handle->config->mosi_3_pin->pin,
handle->config->mosi_3_pin_config->port_config.mode);
break;

default:
break;
}
```

```
}

/*
 * This is used to make the ports as input during update of the baud rate, to avoid the noise in output ports
 */
static void SPI_MASTER_lPortModeReset(const SPI_MASTER_t* handle)
{
 uint32_t ss_line;

 /* Configure the ports as input */
 for (ss_line = 0U; ss_line < handle->config->slave_select_lines; ss_line++)
 {
 XMC_GPIO_SetMode(handle->config->slave_select_pin[ss_line]->port,
 handle->config->slave_select_pin[ss_line]->pin,
 XMC_GPIO_MODE_INPUT_TRISTATE);
 }

 XMC_GPIO_SetMode(handle->config->sclk_out_pin->port, handle->config->sclk_out_pin->pin, XMC_GPIO_MODE_INPUT_TRISTATE);

 switch (handle->runtime->spi_master_mode)
 {
 case XMC_SPI_CH_MODE_STANDARD:
 case XMC_SPI_CH_MODE_STANDARD_HALFDUPLEX:
 XMC_GPIO_SetMode(handle->config->mosi_0_pin->port, handle->config->mosi_0_pin->pin, XMC_GPIO_MODE_INPUT_TRISTATE);
 break;

 case XMC_SPI_CH_MODE_DUAL:
 XMC_GPIO_SetMode(handle->config->mosi_0_pin->port, handle->config->mosi_0_pin->pin,
XMC_GPIO_MODE_INPUT_TRISTATE);
  XMC_GPIO_SetMode(handle->config->mosi_1_pin->port,
handle->config->mosi_1_pin->pin,
XMC_GPIO_MODE_INPUT_TRISTATE);
  break;

  case XMC_SPI_CH_MODE_QUAD:
  XMC_GPIO_SetMode(handle->config->mosi_0_pin->port,
handle->config->mosi_0_pin->pin,
XMC_GPIO_MODE_INPUT_TRISTATE);
  XMC_GPIO_SetMode(handle->config->mosi_1_pin->port,
handle->config->mosi_1_pin->pin,
XMC_GPIO_MODE_INPUT_TRISTATE);
  XMC_GPIO_SetMode(handle->config->mosi_2_pin->port,
handle->config->mosi_2_pin->pin,
XMC_GPIO_MODE_INPUT_TRISTATE);
  XMC_GPIO_SetMode(handle->config->mosi_3_pin->port,
handle->config->mosi_3_pin->pin,
XMC_GPIO_MODE_INPUT_TRISTATE);
  break;

  default:
  break;
  }
}

/*
  * This is used check whether the mode change is valid or not
  */
static SPI_MASTER_STATUS_t
SPI_MASTER_lValidateModeChange(const SPI_MASTER_t * handle,
XMC_SPI_CH_MODE_t mode)
{
  SPI_MASTER_STATUS_t status;

  status = SPI_MASTER_STATUS_SUCCESS;

```

```
if ((handle->config->spi_master_config_mode == XMC_SPI_CH_MODE_STANDARD_HALFDUPLEX) ||
(handle->config->spi_master_config_mode < mode))
{
status = SPI_MASTER_STATUS_FAILURE;
}
else if (handle->config->spi_master_config_mode == XMC_SPI_CH_MODE_STANDARD)
{
if (XMC_SPI_CH_MODE_DUAL <= mode)
{
status = SPI_MASTER_STATUS_FAILURE;
}
}
else
{
if ((mode == XMC_SPI_CH_MODE_STANDARD) && (handle->runtime->dx0_input == SPI_MASTER_INPUT_INVALID))
{
status = SPI_MASTER_STATUS_FAILURE;
}

else if ((mode == XMC_SPI_CH_MODE_STANDARD_HALFDUPLEX) && (handle->runtime->dx0_input_half_duplex == SPI_MASTER_INPUT_INVALID))
{
status = SPI_MASTER_STATUS_FAILURE;
}
else
{
/* added to abide MISRA */
}
}
return status;
}
```

SPI_MASTER

Enumerations

enum **SPI_MASTER_STATUS** {
SPI_MASTER_STATUS_S
SPI_MASTER_STATUS_FA
SPI_MASTER_STATUS_B
SPI_MASTER_STATUS_B
SPI_MASTER_STATUS_I
}
Return status of the **SPI_MA**

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_SIGNA
SPI_MASTER_SS_SIGNA
SPI_MASTER_SS_SIGNA
SPI_MASTER_SS_SIGNA
SPI_MASTER_SS_SIGNA
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_TRANSFER** **SPI_MASTER_TRANSFER** **SPI_MASTER_TRANSFER** **SPI_MASTER_TRANSFER** 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_TRANSFER** 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**.

Code

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_INDICATI
SPI_MASTER_ClearFlag(&SPI_MASTER_0,
(uint32_t)XMC_SPI_CH_STATUS_FLAG_TRANSMIT_SHIFT_INDICATI
}
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_MAS
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_

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 (cons
cons
)

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 (con
con
)

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**.