/\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file stm32f4xx\_spi.c

\* @author MCD Application Team

\* @version V1.3.0

\* @date 08-November-2013

\* @brief This file provides firmware functions to manage the following

\* functionalities of the Serial peripheral interface (SPI):

\* + Initialization and Configuration

\* + Data transfers functions

\* + Hardware CRC Calculation

\* + DMA transfers management

\* + Interrupts and flags management

\*

@verbatim

===================================================================

##### How to use this driver #####

===================================================================

[..]

(#) Enable peripheral clock using the following functions

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_SPI1, ENABLE) for SPI1

RCC\_APB1PeriphClockCmd(RCC\_APB1Periph\_SPI2, ENABLE) for SPI2

RCC\_APB1PeriphResetCmd(RCC\_APB1Periph\_SPI3, ENABLE) for SPI3

RCC\_APB1PeriphResetCmd(RCC\_APB1Periph\_SPI3, ENABLE) for SPI4

RCC\_APB1PeriphResetCmd(RCC\_APB1Periph\_SPI3, ENABLE) for SPI5

RCC\_APB1PeriphResetCmd(RCC\_APB1Periph\_SPI3, ENABLE) for SPI6.

(#) Enable SCK, MOSI, MISO and NSS GPIO clocks using RCC\_AHB1PeriphClockCmd()

function. In I2S mode, if an external clock source is used then the I2S

CKIN pin GPIO clock should also be enabled.

(#) Peripherals alternate function:

(++) Connect the pin to the desired peripherals' Alternate Function (AF)

using GPIO\_PinAFConfig() function

(++) Configure the desired pin in alternate function by:

GPIO\_InitStruct->GPIO\_Mode = GPIO\_Mode\_AF

(++) Select the type, pull-up/pull-down and output speed via GPIO\_PuPd,

GPIO\_OType and GPIO\_Speed members

(++) Call GPIO\_Init() function In I2S mode, if an external clock source is

used then the I2S CKIN pin should be also configured in Alternate

function Push-pull pull-up mode.

(#) Program the Polarity, Phase, First Data, Baud Rate Prescaler, Slave

Management, Peripheral Mode and CRC Polynomial values using the SPI\_Init()

function.

In I2S mode, program the Mode, Standard, Data Format, MCLK Output, Audio

frequency and Polarity using I2S\_Init() function. For I2S mode, make sure

that either:

(++) I2S PLL is configured using the functions

RCC\_I2SCLKConfig(RCC\_I2S2CLKSource\_PLLI2S), RCC\_PLLI2SCmd(ENABLE) and

RCC\_GetFlagStatus(RCC\_FLAG\_PLLI2SRDY); or

(++) External clock source is configured using the function

RCC\_I2SCLKConfig(RCC\_I2S2CLKSource\_Ext) and after setting correctly

the define constant I2S\_EXTERNAL\_CLOCK\_VAL in the stm32f4xx\_conf.h file.

(#) Enable the NVIC and the corresponding interrupt using the function

SPI\_ITConfig() if you need to use interrupt mode.

(#) When using the DMA mode

(++) Configure the DMA using DMA\_Init() function

(++) Active the needed channel Request using SPI\_I2S\_DMACmd() function

(#) Enable the SPI using the SPI\_Cmd() function or enable the I2S using

I2S\_Cmd().

(#) Enable the DMA using the DMA\_Cmd() function when using DMA mode.

(#) Optionally, you can enable/configure the following parameters without

re-initialization (i.e there is no need to call again SPI\_Init() function):

(++) When bidirectional mode (SPI\_Direction\_1Line\_Rx or SPI\_Direction\_1Line\_Tx)

is programmed as Data direction parameter using the SPI\_Init() function

it can be possible to switch between SPI\_Direction\_Tx or SPI\_Direction\_Rx

using the SPI\_BiDirectionalLineConfig() function.

(++) When SPI\_NSS\_Soft is selected as Slave Select Management parameter

using the SPI\_Init() function it can be possible to manage the

NSS internal signal using the SPI\_NSSInternalSoftwareConfig() function.

(++) Reconfigure the data size using the SPI\_DataSizeConfig() function

(++) Enable or disable the SS output using the SPI\_SSOutputCmd() function

(#) To use the CRC Hardware calculation feature refer to the Peripheral

CRC hardware Calculation subsection.

[..] It is possible to use SPI in I2S full duplex mode, in this case, each SPI

peripheral is able to manage sending and receiving data simultaneously

using two data lines. Each SPI peripheral has an extended block called I2Sxext

(ie. I2S2ext for SPI2 and I2S3ext for SPI3).

The extension block is not a full SPI IP, it is used only as I2S slave to

implement full duplex mode. The extension block uses the same clock sources

as its master.

To configure I2S full duplex you have to:

(#) Configure SPIx in I2S mode (I2S\_Init() function) as described above.

(#) Call the I2S\_FullDuplexConfig() function using the same strucutre passed to

I2S\_Init() function.

(#) Call I2S\_Cmd() for SPIx then for its extended block.

(#) To configure interrupts or DMA requests and to get/clear flag status,

use I2Sxext instance for the extension block.

[..] Functions that can be called with I2Sxext instances are: I2S\_Cmd(),

I2S\_FullDuplexConfig(), SPI\_I2S\_ReceiveData(), SPI\_I2S\_SendData(),

SPI\_I2S\_DMACmd(), SPI\_I2S\_ITConfig(), SPI\_I2S\_GetFlagStatus(),

SPI\_I2S\_ClearFlag(), SPI\_I2S\_GetITStatus() and SPI\_I2S\_ClearITPendingBit().

Example: To use SPI3 in Full duplex mode (SPI3 is Master Tx, I2S3ext is Slave Rx):

RCC\_APB1PeriphClockCmd(RCC\_APB1Periph\_SPI3, ENABLE);

I2S\_StructInit(&I2SInitStruct);

I2SInitStruct.Mode = I2S\_Mode\_MasterTx;

I2S\_Init(SPI3, &I2SInitStruct);

I2S\_FullDuplexConfig(SPI3ext, &I2SInitStruct)

I2S\_Cmd(SPI3, ENABLE);

I2S\_Cmd(SPI3ext, ENABLE);

...

while (SPI\_I2S\_GetFlagStatus(SPI2, SPI\_FLAG\_TXE) == RESET)

{}

SPI\_I2S\_SendData(SPI3, txdata[i]);

...

while (SPI\_I2S\_GetFlagStatus(I2S3ext, SPI\_FLAG\_RXNE) == RESET)

{}

rxdata[i] = SPI\_I2S\_ReceiveData(I2S3ext);

...

[..]

(@) In I2S mode: if an external clock is used as source clock for the I2S,

then the define I2S\_EXTERNAL\_CLOCK\_VAL in file stm32f4xx\_conf.h should

be enabled and set to the value of the source clock frequency (in Hz).

(@) In SPI mode: To use the SPI TI mode, call the function SPI\_TIModeCmd()

just after calling the function SPI\_Init().

@endverbatim

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @attention

\*

\* <h2><center>&copy; COPYRIGHT 2013 STMicroelectronics</center></h2>

\*

\* Licensed under MCD-ST Liberty SW License Agreement V2, (the "License");

\* You may not use this file except in compliance with the License.

\* You may obtain a copy of the License at:

\*

\* http://www.st.com/software\_license\_agreement\_liberty\_v2

\*

\* Unless required by applicable law or agreed to in writing, software

\* distributed under the License is distributed on an "AS IS" BASIS,

\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

\* See the License for the specific language governing permissions and

\* limitations under the License.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

/\* Includes ------------------------------------------------------------------\*/

#include "stm32f4xx\_spi.h"

#include "stm32f4xx\_rcc.h"

/\*\* @addtogroup STM32F4xx\_StdPeriph\_Driver

\* @{

\*/

/\*\* @defgroup SPI

\* @brief SPI driver modules

\* @{

\*/

/\* Private typedef -----------------------------------------------------------\*/

/\* Private define ------------------------------------------------------------\*/

/\* SPI registers Masks \*/

#define CR1\_CLEAR\_MASK ((uint16\_t)0x3040)

#define I2SCFGR\_CLEAR\_MASK ((uint16\_t)0xF040)

/\* RCC PLLs masks \*/

#define PLLCFGR\_PPLR\_MASK ((uint32\_t)0x70000000)

#define PLLCFGR\_PPLN\_MASK ((uint32\_t)0x00007FC0)

#define SPI\_CR2\_FRF ((uint16\_t)0x0010)

#define SPI\_SR\_TIFRFE ((uint16\_t)0x0100)

/\* Private macro -------------------------------------------------------------\*/

/\* Private variables ---------------------------------------------------------\*/

/\* Private function prototypes -----------------------------------------------\*/

/\* Private functions ---------------------------------------------------------\*/

/\*\* @defgroup SPI\_Private\_Functions

\* @{

\*/

/\*\* @defgroup SPI\_Group1 Initialization and Configuration functions

\* @brief Initialization and Configuration functions

\*

@verbatim

===============================================================================

##### Initialization and Configuration functions #####

===============================================================================

[..] This section provides a set of functions allowing to initialize the SPI

Direction, SPI Mode, SPI Data Size, SPI Polarity, SPI Phase, SPI NSS

Management, SPI Baud Rate Prescaler, SPI First Bit and SPI CRC Polynomial.

[..] The SPI\_Init() function follows the SPI configuration procedures for Master

mode and Slave mode (details for these procedures are available in reference

manual (RM0090)).

@endverbatim

\* @{

\*/

/\*\*

\* @brief De-initialize the SPIx peripheral registers to their default reset values.

\* @param SPIx: To select the SPIx/I2Sx peripheral, where x can be: 1, 2, 3, 4, 5 or 6

\* in SPI mode or 2 or 3 in I2S mode.

\*

\* @note The extended I2S blocks (ie. I2S2ext and I2S3ext blocks) are de-initialized

\* when the relative I2S peripheral is de-initialized (the extended block's clock

\* is managed by the I2S peripheral clock).

\*

\* @retval None

\*/

void SPI\_I2S\_DeInit(SPI\_TypeDef\* SPIx)

{

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH(SPIx));

if (SPIx == SPI1)

{

/\* Enable SPI1 reset state \*/

RCC\_APB2PeriphResetCmd(RCC\_APB2Periph\_SPI1, ENABLE);

/\* Release SPI1 from reset state \*/

RCC\_APB2PeriphResetCmd(RCC\_APB2Periph\_SPI1, DISABLE);

}

else if (SPIx == SPI2)

{

/\* Enable SPI2 reset state \*/

RCC\_APB1PeriphResetCmd(RCC\_APB1Periph\_SPI2, ENABLE);

/\* Release SPI2 from reset state \*/

RCC\_APB1PeriphResetCmd(RCC\_APB1Periph\_SPI2, DISABLE);

}

else if (SPIx == SPI3)

{

/\* Enable SPI3 reset state \*/

RCC\_APB1PeriphResetCmd(RCC\_APB1Periph\_SPI3, ENABLE);

/\* Release SPI3 from reset state \*/

RCC\_APB1PeriphResetCmd(RCC\_APB1Periph\_SPI3, DISABLE);

}

else if (SPIx == SPI4)

{

/\* Enable SPI4 reset state \*/

RCC\_APB2PeriphResetCmd(RCC\_APB2Periph\_SPI4, ENABLE);

/\* Release SPI4 from reset state \*/

RCC\_APB2PeriphResetCmd(RCC\_APB2Periph\_SPI4, DISABLE);

}

else if (SPIx == SPI5)

{

/\* Enable SPI5 reset state \*/

RCC\_APB2PeriphResetCmd(RCC\_APB2Periph\_SPI5, ENABLE);

/\* Release SPI5 from reset state \*/

RCC\_APB2PeriphResetCmd(RCC\_APB2Periph\_SPI5, DISABLE);

}

else

{

if (SPIx == SPI6)

{

/\* Enable SPI6 reset state \*/

RCC\_APB2PeriphResetCmd(RCC\_APB2Periph\_SPI6, ENABLE);

/\* Release SPI6 from reset state \*/

RCC\_APB2PeriphResetCmd(RCC\_APB2Periph\_SPI6, DISABLE);

}

}

}

/\*\*

\* @brief Initializes the SPIx peripheral according to the specified

\* parameters in the SPI\_InitStruct.

\* @param SPIx: where x can be 1, 2, 3, 4, 5 or 6 to select the SPI peripheral.

\* @param SPI\_InitStruct: pointer to a SPI\_InitTypeDef structure that

\* contains the configuration information for the specified SPI peripheral.

\* @retval None

\*/

void SPI\_Init(SPI\_TypeDef\* SPIx, SPI\_InitTypeDef\* SPI\_InitStruct)

{

uint16\_t tmpreg = 0;

/\* check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH(SPIx));

/\* Check the SPI parameters \*/

assert\_param(IS\_SPI\_DIRECTION\_MODE(SPI\_InitStruct->SPI\_Direction));

assert\_param(IS\_SPI\_MODE(SPI\_InitStruct->SPI\_Mode));

assert\_param(IS\_SPI\_DATASIZE(SPI\_InitStruct->SPI\_DataSize));

assert\_param(IS\_SPI\_CPOL(SPI\_InitStruct->SPI\_CPOL));

assert\_param(IS\_SPI\_CPHA(SPI\_InitStruct->SPI\_CPHA));

assert\_param(IS\_SPI\_NSS(SPI\_InitStruct->SPI\_NSS));

assert\_param(IS\_SPI\_BAUDRATE\_PRESCALER(SPI\_InitStruct->SPI\_BaudRatePrescaler));

assert\_param(IS\_SPI\_FIRST\_BIT(SPI\_InitStruct->SPI\_FirstBit));

assert\_param(IS\_SPI\_CRC\_POLYNOMIAL(SPI\_InitStruct->SPI\_CRCPolynomial));

/\*---------------------------- SPIx CR1 Configuration ------------------------\*/

/\* Get the SPIx CR1 value \*/

tmpreg = SPIx->CR1;

/\* Clear BIDIMode, BIDIOE, RxONLY, SSM, SSI, LSBFirst, BR, MSTR, CPOL and CPHA bits \*/

tmpreg &= CR1\_CLEAR\_MASK;

/\* Configure SPIx: direction, NSS management, first transmitted bit, BaudRate prescaler

master/salve mode, CPOL and CPHA \*/

/\* Set BIDImode, BIDIOE and RxONLY bits according to SPI\_Direction value \*/

/\* Set SSM, SSI and MSTR bits according to SPI\_Mode and SPI\_NSS values \*/

/\* Set LSBFirst bit according to SPI\_FirstBit value \*/

/\* Set BR bits according to SPI\_BaudRatePrescaler value \*/

/\* Set CPOL bit according to SPI\_CPOL value \*/

/\* Set CPHA bit according to SPI\_CPHA value \*/

tmpreg |= (uint16\_t)((uint32\_t)SPI\_InitStruct->SPI\_Direction | SPI\_InitStruct->SPI\_Mode |

SPI\_InitStruct->SPI\_DataSize | SPI\_InitStruct->SPI\_CPOL |

SPI\_InitStruct->SPI\_CPHA | SPI\_InitStruct->SPI\_NSS |

SPI\_InitStruct->SPI\_BaudRatePrescaler | SPI\_InitStruct->SPI\_FirstBit);

/\* Write to SPIx CR1 \*/

SPIx->CR1 = tmpreg;

/\* Activate the SPI mode (Reset I2SMOD bit in I2SCFGR register) \*/

SPIx->I2SCFGR &= (uint16\_t)~((uint16\_t)SPI\_I2SCFGR\_I2SMOD);

/\*---------------------------- SPIx CRCPOLY Configuration --------------------\*/

/\* Write to SPIx CRCPOLY \*/

SPIx->CRCPR = SPI\_InitStruct->SPI\_CRCPolynomial;

}

/\*\*

\* @brief Initializes the SPIx peripheral according to the specified

\* parameters in the I2S\_InitStruct.

\* @param SPIx: where x can be 2 or 3 to select the SPI peripheral (configured in I2S mode).

\* @param I2S\_InitStruct: pointer to an I2S\_InitTypeDef structure that

\* contains the configuration information for the specified SPI peripheral

\* configured in I2S mode.

\*

\* @note The function calculates the optimal prescaler needed to obtain the most

\* accurate audio frequency (depending on the I2S clock source, the PLL values

\* and the product configuration). But in case the prescaler value is greater

\* than 511, the default value (0x02) will be configured instead.

\*

\* @note if an external clock is used as source clock for the I2S, then the define

\* I2S\_EXTERNAL\_CLOCK\_VAL in file stm32f4xx\_conf.h should be enabled and set

\* to the value of the the source clock frequency (in Hz).

\*

\* @retval None

\*/

void I2S\_Init(SPI\_TypeDef\* SPIx, I2S\_InitTypeDef\* I2S\_InitStruct)

{

uint16\_t tmpreg = 0, i2sdiv = 2, i2sodd = 0, packetlength = 1;

uint32\_t tmp = 0, i2sclk = 0;

#ifndef I2S\_EXTERNAL\_CLOCK\_VAL

uint32\_t pllm = 0, plln = 0, pllr = 0;

#endif /\* I2S\_EXTERNAL\_CLOCK\_VAL \*/

/\* Check the I2S parameters \*/

assert\_param(IS\_SPI\_23\_PERIPH(SPIx));

assert\_param(IS\_I2S\_MODE(I2S\_InitStruct->I2S\_Mode));

assert\_param(IS\_I2S\_STANDARD(I2S\_InitStruct->I2S\_Standard));

assert\_param(IS\_I2S\_DATA\_FORMAT(I2S\_InitStruct->I2S\_DataFormat));

assert\_param(IS\_I2S\_MCLK\_OUTPUT(I2S\_InitStruct->I2S\_MCLKOutput));

assert\_param(IS\_I2S\_AUDIO\_FREQ(I2S\_InitStruct->I2S\_AudioFreq));

assert\_param(IS\_I2S\_CPOL(I2S\_InitStruct->I2S\_CPOL));

/\*----------------------- SPIx I2SCFGR & I2SPR Configuration -----------------\*/

/\* Clear I2SMOD, I2SE, I2SCFG, PCMSYNC, I2SSTD, CKPOL, DATLEN and CHLEN bits \*/

SPIx->I2SCFGR &= I2SCFGR\_CLEAR\_MASK;

SPIx->I2SPR = 0x0002;

/\* Get the I2SCFGR register value \*/

tmpreg = SPIx->I2SCFGR;

/\* If the default value has to be written, reinitialize i2sdiv and i2sodd\*/

if(I2S\_InitStruct->I2S\_AudioFreq == I2S\_AudioFreq\_Default)

{

i2sodd = (uint16\_t)0;

i2sdiv = (uint16\_t)2;

}

/\* If the requested audio frequency is not the default, compute the prescaler \*/

else

{

/\* Check the frame length (For the Prescaler computing) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if(I2S\_InitStruct->I2S\_DataFormat == I2S\_DataFormat\_16b)

{

/\* Packet length is 16 bits \*/

packetlength = 1;

}

else

{

/\* Packet length is 32 bits \*/

packetlength = 2;

}

/\* Get I2S source Clock frequency \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* If an external I2S clock has to be used, this define should be set

in the project configuration or in the stm32f4xx\_conf.h file \*/

#ifdef I2S\_EXTERNAL\_CLOCK\_VAL

/\* Set external clock as I2S clock source \*/

if ((RCC->CFGR & RCC\_CFGR\_I2SSRC) == 0)

{

RCC->CFGR |= (uint32\_t)RCC\_CFGR\_I2SSRC;

}

/\* Set the I2S clock to the external clock value \*/

i2sclk = I2S\_EXTERNAL\_CLOCK\_VAL;

#else /\* There is no define for External I2S clock source \*/

/\* Set PLLI2S as I2S clock source \*/

if ((RCC->CFGR & RCC\_CFGR\_I2SSRC) != 0)

{

RCC->CFGR &= ~(uint32\_t)RCC\_CFGR\_I2SSRC;

}

/\* Get the PLLI2SN value \*/

plln = (uint32\_t)(((RCC->PLLI2SCFGR & RCC\_PLLI2SCFGR\_PLLI2SN) >> 6) & \

(RCC\_PLLI2SCFGR\_PLLI2SN >> 6));

/\* Get the PLLI2SR value \*/

pllr = (uint32\_t)(((RCC->PLLI2SCFGR & RCC\_PLLI2SCFGR\_PLLI2SR) >> 28) & \

(RCC\_PLLI2SCFGR\_PLLI2SR >> 28));

/\* Get the PLLM value \*/

pllm = (uint32\_t)(RCC->PLLCFGR & RCC\_PLLCFGR\_PLLM);

/\* Get the I2S source clock value \*/

i2sclk = (uint32\_t)(((HSE\_VALUE / pllm) \* plln) / pllr);

#endif /\* I2S\_EXTERNAL\_CLOCK\_VAL \*/

/\* Compute the Real divider depending on the MCLK output state, with a floating point \*/

if(I2S\_InitStruct->I2S\_MCLKOutput == I2S\_MCLKOutput\_Enable)

{

/\* MCLK output is enabled \*/

tmp = (uint16\_t)(((((i2sclk / 256) \* 10) / I2S\_InitStruct->I2S\_AudioFreq)) + 5);

}

else

{

/\* MCLK output is disabled \*/

tmp = (uint16\_t)(((((i2sclk / (32 \* packetlength)) \*10 ) / I2S\_InitStruct->I2S\_AudioFreq)) + 5);

}

/\* Remove the flatting point \*/

tmp = tmp / 10;

/\* Check the parity of the divider \*/

i2sodd = (uint16\_t)(tmp & (uint16\_t)0x0001);

/\* Compute the i2sdiv prescaler \*/

i2sdiv = (uint16\_t)((tmp - i2sodd) / 2);

/\* Get the Mask for the Odd bit (SPI\_I2SPR[8]) register \*/

i2sodd = (uint16\_t) (i2sodd << 8);

}

/\* Test if the divider is 1 or 0 or greater than 0xFF \*/

if ((i2sdiv < 2) || (i2sdiv > 0xFF))

{

/\* Set the default values \*/

i2sdiv = 2;

i2sodd = 0;

}

/\* Write to SPIx I2SPR register the computed value \*/

SPIx->I2SPR = (uint16\_t)((uint16\_t)i2sdiv | (uint16\_t)(i2sodd | (uint16\_t)I2S\_InitStruct->I2S\_MCLKOutput));

/\* Configure the I2S with the SPI\_InitStruct values \*/

tmpreg |= (uint16\_t)((uint16\_t)SPI\_I2SCFGR\_I2SMOD | (uint16\_t)(I2S\_InitStruct->I2S\_Mode | \

(uint16\_t)(I2S\_InitStruct->I2S\_Standard | (uint16\_t)(I2S\_InitStruct->I2S\_DataFormat | \

(uint16\_t)I2S\_InitStruct->I2S\_CPOL))));

/\* Write to SPIx I2SCFGR \*/

SPIx->I2SCFGR = tmpreg;

}

/\*\*

\* @brief Fills each SPI\_InitStruct member with its default value.

\* @param SPI\_InitStruct: pointer to a SPI\_InitTypeDef structure which will be initialized.

\* @retval None

\*/

void SPI\_StructInit(SPI\_InitTypeDef\* SPI\_InitStruct)

{

/\*--------------- Reset SPI init structure parameters values -----------------\*/

/\* Initialize the SPI\_Direction member \*/

SPI\_InitStruct->SPI\_Direction = SPI\_Direction\_2Lines\_FullDuplex;

/\* initialize the SPI\_Mode member \*/

SPI\_InitStruct->SPI\_Mode = SPI\_Mode\_Slave;

/\* initialize the SPI\_DataSize member \*/

SPI\_InitStruct->SPI\_DataSize = SPI\_DataSize\_8b;

/\* Initialize the SPI\_CPOL member \*/

SPI\_InitStruct->SPI\_CPOL = SPI\_CPOL\_Low;

/\* Initialize the SPI\_CPHA member \*/

SPI\_InitStruct->SPI\_CPHA = SPI\_CPHA\_1Edge;

/\* Initialize the SPI\_NSS member \*/

SPI\_InitStruct->SPI\_NSS = SPI\_NSS\_Hard;

/\* Initialize the SPI\_BaudRatePrescaler member \*/

SPI\_InitStruct->SPI\_BaudRatePrescaler = SPI\_BaudRatePrescaler\_2;

/\* Initialize the SPI\_FirstBit member \*/

SPI\_InitStruct->SPI\_FirstBit = SPI\_FirstBit\_MSB;

/\* Initialize the SPI\_CRCPolynomial member \*/

SPI\_InitStruct->SPI\_CRCPolynomial = 7;

}

/\*\*

\* @brief Fills each I2S\_InitStruct member with its default value.

\* @param I2S\_InitStruct: pointer to a I2S\_InitTypeDef structure which will be initialized.

\* @retval None

\*/

void I2S\_StructInit(I2S\_InitTypeDef\* I2S\_InitStruct)

{

/\*--------------- Reset I2S init structure parameters values -----------------\*/

/\* Initialize the I2S\_Mode member \*/

I2S\_InitStruct->I2S\_Mode = I2S\_Mode\_SlaveTx;

/\* Initialize the I2S\_Standard member \*/

I2S\_InitStruct->I2S\_Standard = I2S\_Standard\_Phillips;

/\* Initialize the I2S\_DataFormat member \*/

I2S\_InitStruct->I2S\_DataFormat = I2S\_DataFormat\_16b;

/\* Initialize the I2S\_MCLKOutput member \*/

I2S\_InitStruct->I2S\_MCLKOutput = I2S\_MCLKOutput\_Disable;

/\* Initialize the I2S\_AudioFreq member \*/

I2S\_InitStruct->I2S\_AudioFreq = I2S\_AudioFreq\_Default;

/\* Initialize the I2S\_CPOL member \*/

I2S\_InitStruct->I2S\_CPOL = I2S\_CPOL\_Low;

}

/\*\*

\* @brief Enables or disables the specified SPI peripheral.

\* @param SPIx: where x can be 1, 2, 3, 4, 5 or 6 to select the SPI peripheral.

\* @param NewState: new state of the SPIx peripheral.

\* This parameter can be: ENABLE or DISABLE.

\* @retval None

\*/

void SPI\_Cmd(SPI\_TypeDef\* SPIx, FunctionalState NewState)

{

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH(SPIx));

assert\_param(IS\_FUNCTIONAL\_STATE(NewState));

if (NewState != DISABLE)

{

/\* Enable the selected SPI peripheral \*/

SPIx->CR1 |= SPI\_CR1\_SPE;

}

else

{

/\* Disable the selected SPI peripheral \*/

SPIx->CR1 &= (uint16\_t)~((uint16\_t)SPI\_CR1\_SPE);

}

}

/\*\*

\* @brief Enables or disables the specified SPI peripheral (in I2S mode).

\* @param SPIx: where x can be 2 or 3 to select the SPI peripheral (or I2Sxext

\* for full duplex mode).

\* @param NewState: new state of the SPIx peripheral.

\* This parameter can be: ENABLE or DISABLE.

\* @retval None

\*/

void I2S\_Cmd(SPI\_TypeDef\* SPIx, FunctionalState NewState)

{

/\* Check the parameters \*/

assert\_param(IS\_SPI\_23\_PERIPH\_EXT(SPIx));

assert\_param(IS\_FUNCTIONAL\_STATE(NewState));

if (NewState != DISABLE)

{

/\* Enable the selected SPI peripheral (in I2S mode) \*/

SPIx->I2SCFGR |= SPI\_I2SCFGR\_I2SE;

}

else

{

/\* Disable the selected SPI peripheral in I2S mode \*/

SPIx->I2SCFGR &= (uint16\_t)~((uint16\_t)SPI\_I2SCFGR\_I2SE);

}

}

/\*\*

\* @brief Configures the data size for the selected SPI.

\* @param SPIx: where x can be 1, 2, 3, 4, 5 or 6 to select the SPI peripheral.

\* @param SPI\_DataSize: specifies the SPI data size.

\* This parameter can be one of the following values:

\* @arg SPI\_DataSize\_16b: Set data frame format to 16bit

\* @arg SPI\_DataSize\_8b: Set data frame format to 8bit

\* @retval None

\*/

void SPI\_DataSizeConfig(SPI\_TypeDef\* SPIx, uint16\_t SPI\_DataSize)

{

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH(SPIx));

assert\_param(IS\_SPI\_DATASIZE(SPI\_DataSize));

/\* Clear DFF bit \*/

SPIx->CR1 &= (uint16\_t)~SPI\_DataSize\_16b;

/\* Set new DFF bit value \*/

SPIx->CR1 |= SPI\_DataSize;

}

/\*\*

\* @brief Selects the data transfer direction in bidirectional mode for the specified SPI.

\* @param SPIx: where x can be 1, 2, 3, 4, 5 or 6 to select the SPI peripheral.

\* @param SPI\_Direction: specifies the data transfer direction in bidirectional mode.

\* This parameter can be one of the following values:

\* @arg SPI\_Direction\_Tx: Selects Tx transmission direction

\* @arg SPI\_Direction\_Rx: Selects Rx receive direction

\* @retval None

\*/

void SPI\_BiDirectionalLineConfig(SPI\_TypeDef\* SPIx, uint16\_t SPI\_Direction)

{

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH(SPIx));

assert\_param(IS\_SPI\_DIRECTION(SPI\_Direction));

if (SPI\_Direction == SPI\_Direction\_Tx)

{

/\* Set the Tx only mode \*/

SPIx->CR1 |= SPI\_Direction\_Tx;

}

else

{

/\* Set the Rx only mode \*/

SPIx->CR1 &= SPI\_Direction\_Rx;

}

}

/\*\*

\* @brief Configures internally by software the NSS pin for the selected SPI.

\* @param SPIx: where x can be 1, 2, 3, 4, 5 or 6 to select the SPI peripheral.

\* @param SPI\_NSSInternalSoft: specifies the SPI NSS internal state.

\* This parameter can be one of the following values:

\* @arg SPI\_NSSInternalSoft\_Set: Set NSS pin internally

\* @arg SPI\_NSSInternalSoft\_Reset: Reset NSS pin internally

\* @retval None

\*/

void SPI\_NSSInternalSoftwareConfig(SPI\_TypeDef\* SPIx, uint16\_t SPI\_NSSInternalSoft)

{

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH(SPIx));

assert\_param(IS\_SPI\_NSS\_INTERNAL(SPI\_NSSInternalSoft));

if (SPI\_NSSInternalSoft != SPI\_NSSInternalSoft\_Reset)

{

/\* Set NSS pin internally by software \*/

SPIx->CR1 |= SPI\_NSSInternalSoft\_Set;

}

else

{

/\* Reset NSS pin internally by software \*/

SPIx->CR1 &= SPI\_NSSInternalSoft\_Reset;

}

}

/\*\*

\* @brief Enables or disables the SS output for the selected SPI.

\* @param SPIx: where x can be 1, 2, 3, 4, 5 or 6 to select the SPI peripheral.

\* @param NewState: new state of the SPIx SS output.

\* This parameter can be: ENABLE or DISABLE.

\* @retval None

\*/

void SPI\_SSOutputCmd(SPI\_TypeDef\* SPIx, FunctionalState NewState)

{

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH(SPIx));

assert\_param(IS\_FUNCTIONAL\_STATE(NewState));

if (NewState != DISABLE)

{

/\* Enable the selected SPI SS output \*/

SPIx->CR2 |= (uint16\_t)SPI\_CR2\_SSOE;

}

else

{

/\* Disable the selected SPI SS output \*/

SPIx->CR2 &= (uint16\_t)~((uint16\_t)SPI\_CR2\_SSOE);

}

}

/\*\*

\* @brief Enables or disables the SPIx/I2Sx DMA interface.

\*

\* @note This function can be called only after the SPI\_Init() function has

\* been called.

\* @note When TI mode is selected, the control bits SSM, SSI, CPOL and CPHA

\* are not taken into consideration and are configured by hardware

\* respectively to the TI mode requirements.

\*

\* @param SPIx: where x can be 1, 2, 3, 4, 5 or 6

\* @param NewState: new state of the selected SPI TI communication mode.

\* This parameter can be: ENABLE or DISABLE.

\* @retval None

\*/

void SPI\_TIModeCmd(SPI\_TypeDef\* SPIx, FunctionalState NewState)

{

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH(SPIx));

assert\_param(IS\_FUNCTIONAL\_STATE(NewState));

if (NewState != DISABLE)

{

/\* Enable the TI mode for the selected SPI peripheral \*/

SPIx->CR2 |= SPI\_CR2\_FRF;

}

else

{

/\* Disable the TI mode for the selected SPI peripheral \*/

SPIx->CR2 &= (uint16\_t)~SPI\_CR2\_FRF;

}

}

/\*\*

\* @brief Configures the full duplex mode for the I2Sx peripheral using its

\* extension I2Sxext according to the specified parameters in the

\* I2S\_InitStruct.

\* @param I2Sxext: where x can be 2 or 3 to select the I2S peripheral extension block.

\* @param I2S\_InitStruct: pointer to an I2S\_InitTypeDef structure that

\* contains the configuration information for the specified I2S peripheral

\* extension.

\*

\* @note The structure pointed by I2S\_InitStruct parameter should be the same

\* used for the master I2S peripheral. In this case, if the master is

\* configured as transmitter, the slave will be receiver and vice versa.

\* Or you can force a different mode by modifying the field I2S\_Mode to the

\* value I2S\_SlaveRx or I2S\_SlaveTx indepedently of the master configuration.

\*

\* @note The I2S full duplex extension can be configured in slave mode only.

\*

\* @retval None

\*/

void I2S\_FullDuplexConfig(SPI\_TypeDef\* I2Sxext, I2S\_InitTypeDef\* I2S\_InitStruct)

{

uint16\_t tmpreg = 0, tmp = 0;

/\* Check the I2S parameters \*/

assert\_param(IS\_I2S\_EXT\_PERIPH(I2Sxext));

assert\_param(IS\_I2S\_MODE(I2S\_InitStruct->I2S\_Mode));

assert\_param(IS\_I2S\_STANDARD(I2S\_InitStruct->I2S\_Standard));

assert\_param(IS\_I2S\_DATA\_FORMAT(I2S\_InitStruct->I2S\_DataFormat));

assert\_param(IS\_I2S\_CPOL(I2S\_InitStruct->I2S\_CPOL));

/\*----------------------- SPIx I2SCFGR & I2SPR Configuration -----------------\*/

/\* Clear I2SMOD, I2SE, I2SCFG, PCMSYNC, I2SSTD, CKPOL, DATLEN and CHLEN bits \*/

I2Sxext->I2SCFGR &= I2SCFGR\_CLEAR\_MASK;

I2Sxext->I2SPR = 0x0002;

/\* Get the I2SCFGR register value \*/

tmpreg = I2Sxext->I2SCFGR;

/\* Get the mode to be configured for the extended I2S \*/

if ((I2S\_InitStruct->I2S\_Mode == I2S\_Mode\_MasterTx) || (I2S\_InitStruct->I2S\_Mode == I2S\_Mode\_SlaveTx))

{

tmp = I2S\_Mode\_SlaveRx;

}

else

{

if ((I2S\_InitStruct->I2S\_Mode == I2S\_Mode\_MasterRx) || (I2S\_InitStruct->I2S\_Mode == I2S\_Mode\_SlaveRx))

{

tmp = I2S\_Mode\_SlaveTx;

}

}

/\* Configure the I2S with the SPI\_InitStruct values \*/

tmpreg |= (uint16\_t)((uint16\_t)SPI\_I2SCFGR\_I2SMOD | (uint16\_t)(tmp | \

(uint16\_t)(I2S\_InitStruct->I2S\_Standard | (uint16\_t)(I2S\_InitStruct->I2S\_DataFormat | \

(uint16\_t)I2S\_InitStruct->I2S\_CPOL))));

/\* Write to SPIx I2SCFGR \*/

I2Sxext->I2SCFGR = tmpreg;

}

/\*\*

\* @}

\*/

/\*\* @defgroup SPI\_Group2 Data transfers functions

\* @brief Data transfers functions

\*

@verbatim

===============================================================================

##### Data transfers functions #####

===============================================================================

[..] This section provides a set of functions allowing to manage the SPI data

transfers. In reception, data are received and then stored into an internal

Rx buffer while. In transmission, data are first stored into an internal Tx

buffer before being transmitted.

[..] The read access of the SPI\_DR register can be done using the SPI\_I2S\_ReceiveData()

function and returns the Rx buffered value. Whereas a write access to the SPI\_DR

can be done using SPI\_I2S\_SendData() function and stores the written data into

Tx buffer.

@endverbatim

\* @{

\*/

/\*\*

\* @brief Returns the most recent received data by the SPIx/I2Sx peripheral.

\* @param SPIx: To select the SPIx/I2Sx peripheral, where x can be: 1, 2, 3, 4, 5 or 6

\* in SPI mode or 2 or 3 in I2S mode or I2Sxext for I2S full duplex mode.

\* @retval The value of the received data.

\*/

uint16\_t SPI\_I2S\_ReceiveData(SPI\_TypeDef\* SPIx)

{

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH\_EXT(SPIx));

/\* Return the data in the DR register \*/

return SPIx->DR;

}

/\*\*

\* @brief Transmits a Data through the SPIx/I2Sx peripheral.

\* @param SPIx: To select the SPIx/I2Sx peripheral, where x can be: 1, 2, 3, 4, 5 or 6

\* in SPI mode or 2 or 3 in I2S mode or I2Sxext for I2S full duplex mode.

\* @param Data: Data to be transmitted.

\* @retval None

\*/

void SPI\_I2S\_SendData(SPI\_TypeDef\* SPIx, uint16\_t Data)

{

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH\_EXT(SPIx));

/\* Write in the DR register the data to be sent \*/

SPIx->DR = Data;

}

/\*\*

\* @}

\*/

/\*\* @defgroup SPI\_Group3 Hardware CRC Calculation functions

\* @brief Hardware CRC Calculation functions

\*

@verbatim

===============================================================================

##### Hardware CRC Calculation functions #####

===============================================================================

[..] This section provides a set of functions allowing to manage the SPI CRC hardware

calculation

[..] SPI communication using CRC is possible through the following procedure:

(#) Program the Data direction, Polarity, Phase, First Data, Baud Rate Prescaler,

Slave Management, Peripheral Mode and CRC Polynomial values using the SPI\_Init()

function.

(#) Enable the CRC calculation using the SPI\_CalculateCRC() function.

(#) Enable the SPI using the SPI\_Cmd() function

(#) Before writing the last data to the TX buffer, set the CRCNext bit using the

SPI\_TransmitCRC() function to indicate that after transmission of the last

data, the CRC should be transmitted.

(#) After transmitting the last data, the SPI transmits the CRC. The SPI\_CR1\_CRCNEXT

bit is reset. The CRC is also received and compared against the SPI\_RXCRCR

value.

If the value does not match, the SPI\_FLAG\_CRCERR flag is set and an interrupt

can be generated when the SPI\_I2S\_IT\_ERR interrupt is enabled.

[..]

(@) It is advised not to read the calculated CRC values during the communication.

(@) When the SPI is in slave mode, be careful to enable CRC calculation only

when the clock is stable, that is, when the clock is in the steady state.

If not, a wrong CRC calculation may be done. In fact, the CRC is sensitive

to the SCK slave input clock as soon as CRCEN is set, and this, whatever

the value of the SPE bit.

(@) With high bitrate frequencies, be careful when transmitting the CRC.

As the number of used CPU cycles has to be as low as possible in the CRC

transfer phase, it is forbidden to call software functions in the CRC

transmission sequence to avoid errors in the last data and CRC reception.

In fact, CRCNEXT bit has to be written before the end of the transmission/reception

of the last data.

(@) For high bit rate frequencies, it is advised to use the DMA mode to avoid the

degradation of the SPI speed performance due to CPU accesses impacting the

SPI bandwidth.

(@) When the STM32F4xx is configured as slave and the NSS hardware mode is

used, the NSS pin needs to be kept low between the data phase and the CRC

phase.

(@) When the SPI is configured in slave mode with the CRC feature enabled, CRC

calculation takes place even if a high level is applied on the NSS pin.

This may happen for example in case of a multi-slave environment where the

communication master addresses slaves alternately.

(@) Between a slave de-selection (high level on NSS) and a new slave selection

(low level on NSS), the CRC value should be cleared on both master and slave

sides in order to resynchronize the master and slave for their respective

CRC calculation.

(@) To clear the CRC, follow the procedure below:

(#@) Disable SPI using the SPI\_Cmd() function

(#@) Disable the CRC calculation using the SPI\_CalculateCRC() function.

(#@) Enable the CRC calculation using the SPI\_CalculateCRC() function.

(#@) Enable SPI using the SPI\_Cmd() function.

@endverbatim

\* @{

\*/

/\*\*

\* @brief Enables or disables the CRC value calculation of the transferred bytes.

\* @param SPIx: where x can be 1, 2, 3, 4, 5 or 6 to select the SPI peripheral.

\* @param NewState: new state of the SPIx CRC value calculation.

\* This parameter can be: ENABLE or DISABLE.

\* @retval None

\*/

void SPI\_CalculateCRC(SPI\_TypeDef\* SPIx, FunctionalState NewState)

{

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH(SPIx));

assert\_param(IS\_FUNCTIONAL\_STATE(NewState));

if (NewState != DISABLE)

{

/\* Enable the selected SPI CRC calculation \*/

SPIx->CR1 |= SPI\_CR1\_CRCEN;

}

else

{

/\* Disable the selected SPI CRC calculation \*/

SPIx->CR1 &= (uint16\_t)~((uint16\_t)SPI\_CR1\_CRCEN);

}

}

/\*\*

\* @brief Transmit the SPIx CRC value.

\* @param SPIx: where x can be 1, 2, 3, 4, 5 or 6 to select the SPI peripheral.

\* @retval None

\*/

void SPI\_TransmitCRC(SPI\_TypeDef\* SPIx)

{

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH(SPIx));

/\* Enable the selected SPI CRC transmission \*/

SPIx->CR1 |= SPI\_CR1\_CRCNEXT;

}

/\*\*

\* @brief Returns the transmit or the receive CRC register value for the specified SPI.

\* @param SPIx: where x can be 1, 2, 3, 4, 5 or 6 to select the SPI peripheral.

\* @param SPI\_CRC: specifies the CRC register to be read.

\* This parameter can be one of the following values:

\* @arg SPI\_CRC\_Tx: Selects Tx CRC register

\* @arg SPI\_CRC\_Rx: Selects Rx CRC register

\* @retval The selected CRC register value..

\*/

uint16\_t SPI\_GetCRC(SPI\_TypeDef\* SPIx, uint8\_t SPI\_CRC)

{

uint16\_t crcreg = 0;

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH(SPIx));

assert\_param(IS\_SPI\_CRC(SPI\_CRC));

if (SPI\_CRC != SPI\_CRC\_Rx)

{

/\* Get the Tx CRC register \*/

crcreg = SPIx->TXCRCR;

}

else

{

/\* Get the Rx CRC register \*/

crcreg = SPIx->RXCRCR;

}

/\* Return the selected CRC register \*/

return crcreg;

}

/\*\*

\* @brief Returns the CRC Polynomial register value for the specified SPI.

\* @param SPIx: where x can be 1, 2, 3, 4, 5 or 6 to select the SPI peripheral.

\* @retval The CRC Polynomial register value.

\*/

uint16\_t SPI\_GetCRCPolynomial(SPI\_TypeDef\* SPIx)

{

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH(SPIx));

/\* Return the CRC polynomial register \*/

return SPIx->CRCPR;

}

/\*\*

\* @}

\*/

/\*\* @defgroup SPI\_Group4 DMA transfers management functions

\* @brief DMA transfers management functions

\*

@verbatim

===============================================================================

##### DMA transfers management functions #####

===============================================================================

@endverbatim

\* @{

\*/

/\*\*

\* @brief Enables or disables the SPIx/I2Sx DMA interface.

\* @param SPIx: To select the SPIx/I2Sx peripheral, where x can be: 1, 2, 3, 4, 5 or 6

\* in SPI mode or 2 or 3 in I2S mode or I2Sxext for I2S full duplex mode.

\* @param SPI\_I2S\_DMAReq: specifies the SPI DMA transfer request to be enabled or disabled.

\* This parameter can be any combination of the following values:

\* @arg SPI\_I2S\_DMAReq\_Tx: Tx buffer DMA transfer request

\* @arg SPI\_I2S\_DMAReq\_Rx: Rx buffer DMA transfer request

\* @param NewState: new state of the selected SPI DMA transfer request.

\* This parameter can be: ENABLE or DISABLE.

\* @retval None

\*/

void SPI\_I2S\_DMACmd(SPI\_TypeDef\* SPIx, uint16\_t SPI\_I2S\_DMAReq, FunctionalState NewState)

{

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH\_EXT(SPIx));

assert\_param(IS\_FUNCTIONAL\_STATE(NewState));

assert\_param(IS\_SPI\_I2S\_DMAREQ(SPI\_I2S\_DMAReq));

if (NewState != DISABLE)

{

/\* Enable the selected SPI DMA requests \*/

SPIx->CR2 |= SPI\_I2S\_DMAReq;

}

else

{

/\* Disable the selected SPI DMA requests \*/

SPIx->CR2 &= (uint16\_t)~SPI\_I2S\_DMAReq;

}

}

/\*\*

\* @}

\*/

/\*\* @defgroup SPI\_Group5 Interrupts and flags management functions

\* @brief Interrupts and flags management functions

\*

@verbatim

===============================================================================

##### Interrupts and flags management functions #####

===============================================================================

[..] This section provides a set of functions allowing to configure the SPI Interrupts

sources and check or clear the flags or pending bits status.

The user should identify which mode will be used in his application to manage

the communication: Polling mode, Interrupt mode or DMA mode.

\*\*\* Polling Mode \*\*\*

====================

[..] In Polling Mode, the SPI/I2S communication can be managed by 9 flags:

(#) SPI\_I2S\_FLAG\_TXE : to indicate the status of the transmit buffer register

(#) SPI\_I2S\_FLAG\_RXNE : to indicate the status of the receive buffer register

(#) SPI\_I2S\_FLAG\_BSY : to indicate the state of the communication layer of the SPI.

(#) SPI\_FLAG\_CRCERR : to indicate if a CRC Calculation error occur

(#) SPI\_FLAG\_MODF : to indicate if a Mode Fault error occur

(#) SPI\_I2S\_FLAG\_OVR : to indicate if an Overrun error occur

(#) I2S\_FLAG\_TIFRFE: to indicate a Frame Format error occurs.

(#) I2S\_FLAG\_UDR: to indicate an Underrun error occurs.

(#) I2S\_FLAG\_CHSIDE: to indicate Channel Side.

(@) Do not use the BSY flag to handle each data transmission or reception. It is

better to use the TXE and RXNE flags instead.

[..] In this Mode it is advised to use the following functions:

(+) FlagStatus SPI\_I2S\_GetFlagStatus(SPI\_TypeDef\* SPIx, uint16\_t SPI\_I2S\_FLAG);

(+) void SPI\_I2S\_ClearFlag(SPI\_TypeDef\* SPIx, uint16\_t SPI\_I2S\_FLAG);

\*\*\* Interrupt Mode \*\*\*

======================

[..] In Interrupt Mode, the SPI communication can be managed by 3 interrupt sources

and 7 pending bits:

(+) Pending Bits:

(##) SPI\_I2S\_IT\_TXE : to indicate the status of the transmit buffer register

(##) SPI\_I2S\_IT\_RXNE : to indicate the status of the receive buffer register

(##) SPI\_IT\_CRCERR : to indicate if a CRC Calculation error occur (available in SPI mode only)

(##) SPI\_IT\_MODF : to indicate if a Mode Fault error occur (available in SPI mode only)

(##) SPI\_I2S\_IT\_OVR : to indicate if an Overrun error occur

(##) I2S\_IT\_UDR : to indicate an Underrun Error occurs (available in I2S mode only).

(##) I2S\_FLAG\_TIFRFE : to indicate a Frame Format error occurs (available in TI mode only).

(+) Interrupt Source:

(##) SPI\_I2S\_IT\_TXE: specifies the interrupt source for the Tx buffer empty

interrupt.

(##) SPI\_I2S\_IT\_RXNE : specifies the interrupt source for the Rx buffer not

empty interrupt.

(##) SPI\_I2S\_IT\_ERR : specifies the interrupt source for the errors interrupt.

[..] In this Mode it is advised to use the following functions:

(+) void SPI\_I2S\_ITConfig(SPI\_TypeDef\* SPIx, uint8\_t SPI\_I2S\_IT, FunctionalState NewState);

(+) ITStatus SPI\_I2S\_GetITStatus(SPI\_TypeDef\* SPIx, uint8\_t SPI\_I2S\_IT);

(+) void SPI\_I2S\_ClearITPendingBit(SPI\_TypeDef\* SPIx, uint8\_t SPI\_I2S\_IT);

\*\*\* DMA Mode \*\*\*

================

[..] In DMA Mode, the SPI communication can be managed by 2 DMA Channel requests:

(#) SPI\_I2S\_DMAReq\_Tx: specifies the Tx buffer DMA transfer request

(#) SPI\_I2S\_DMAReq\_Rx: specifies the Rx buffer DMA transfer request

[..] In this Mode it is advised to use the following function:

(+) void SPI\_I2S\_DMACmd(SPI\_TypeDef\* SPIx, uint16\_t SPI\_I2S\_DMAReq, FunctionalState

NewState);

@endverbatim

\* @{

\*/

/\*\*

\* @brief Enables or disables the specified SPI/I2S interrupts.

\* @param SPIx: To select the SPIx/I2Sx peripheral, where x can be: 1, 2, 3, 4, 5 or 6

\* in SPI mode or 2 or 3 in I2S mode or I2Sxext for I2S full duplex mode.

\* @param SPI\_I2S\_IT: specifies the SPI interrupt source to be enabled or disabled.

\* This parameter can be one of the following values:

\* @arg SPI\_I2S\_IT\_TXE: Tx buffer empty interrupt mask

\* @arg SPI\_I2S\_IT\_RXNE: Rx buffer not empty interrupt mask

\* @arg SPI\_I2S\_IT\_ERR: Error interrupt mask

\* @param NewState: new state of the specified SPI interrupt.

\* This parameter can be: ENABLE or DISABLE.

\* @retval None

\*/

void SPI\_I2S\_ITConfig(SPI\_TypeDef\* SPIx, uint8\_t SPI\_I2S\_IT, FunctionalState NewState)

{

uint16\_t itpos = 0, itmask = 0 ;

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH\_EXT(SPIx));

assert\_param(IS\_FUNCTIONAL\_STATE(NewState));

assert\_param(IS\_SPI\_I2S\_CONFIG\_IT(SPI\_I2S\_IT));

/\* Get the SPI IT index \*/

itpos = SPI\_I2S\_IT >> 4;

/\* Set the IT mask \*/

itmask = (uint16\_t)1 << (uint16\_t)itpos;

if (NewState != DISABLE)

{

/\* Enable the selected SPI interrupt \*/

SPIx->CR2 |= itmask;

}

else

{

/\* Disable the selected SPI interrupt \*/

SPIx->CR2 &= (uint16\_t)~itmask;

}

}

/\*\*

\* @brief Checks whether the specified SPIx/I2Sx flag is set or not.

\* @param SPIx: To select the SPIx/I2Sx peripheral, where x can be: 1, 2, 3, 4, 5 or 6

\* in SPI mode or 2 or 3 in I2S mode or I2Sxext for I2S full duplex mode.

\* @param SPI\_I2S\_FLAG: specifies the SPI flag to check.

\* This parameter can be one of the following values:

\* @arg SPI\_I2S\_FLAG\_TXE: Transmit buffer empty flag.

\* @arg SPI\_I2S\_FLAG\_RXNE: Receive buffer not empty flag.

\* @arg SPI\_I2S\_FLAG\_BSY: Busy flag.

\* @arg SPI\_I2S\_FLAG\_OVR: Overrun flag.

\* @arg SPI\_FLAG\_MODF: Mode Fault flag.

\* @arg SPI\_FLAG\_CRCERR: CRC Error flag.

\* @arg SPI\_I2S\_FLAG\_TIFRFE: Format Error.

\* @arg I2S\_FLAG\_UDR: Underrun Error flag.

\* @arg I2S\_FLAG\_CHSIDE: Channel Side flag.

\* @retval The new state of SPI\_I2S\_FLAG (SET or RESET).

\*/

FlagStatus SPI\_I2S\_GetFlagStatus(SPI\_TypeDef\* SPIx, uint16\_t SPI\_I2S\_FLAG)

{

FlagStatus bitstatus = RESET;

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH\_EXT(SPIx));

assert\_param(IS\_SPI\_I2S\_GET\_FLAG(SPI\_I2S\_FLAG));

/\* Check the status of the specified SPI flag \*/

if ((SPIx->SR & SPI\_I2S\_FLAG) != (uint16\_t)RESET)

{

/\* SPI\_I2S\_FLAG is set \*/

bitstatus = SET;

}

else

{

/\* SPI\_I2S\_FLAG is reset \*/

bitstatus = RESET;

}

/\* Return the SPI\_I2S\_FLAG status \*/

return bitstatus;

}

/\*\*

\* @brief Clears the SPIx CRC Error (CRCERR) flag.

\* @param SPIx: To select the SPIx/I2Sx peripheral, where x can be: 1, 2, 3, 4, 5 or 6

\* in SPI mode or 2 or 3 in I2S mode or I2Sxext for I2S full duplex mode.

\* @param SPI\_I2S\_FLAG: specifies the SPI flag to clear.

\* This function clears only CRCERR flag.

\* @arg SPI\_FLAG\_CRCERR: CRC Error flag.

\*

\* @note OVR (OverRun error) flag is cleared by software sequence: a read

\* operation to SPI\_DR register (SPI\_I2S\_ReceiveData()) followed by a read

\* operation to SPI\_SR register (SPI\_I2S\_GetFlagStatus()).

\* @note UDR (UnderRun error) flag is cleared by a read operation to

\* SPI\_SR register (SPI\_I2S\_GetFlagStatus()).

\* @note MODF (Mode Fault) flag is cleared by software sequence: a read/write

\* operation to SPI\_SR register (SPI\_I2S\_GetFlagStatus()) followed by a

\* write operation to SPI\_CR1 register (SPI\_Cmd() to enable the SPI).

\*

\* @retval None

\*/

void SPI\_I2S\_ClearFlag(SPI\_TypeDef\* SPIx, uint16\_t SPI\_I2S\_FLAG)

{

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH\_EXT(SPIx));

assert\_param(IS\_SPI\_I2S\_CLEAR\_FLAG(SPI\_I2S\_FLAG));

/\* Clear the selected SPI CRC Error (CRCERR) flag \*/

SPIx->SR = (uint16\_t)~SPI\_I2S\_FLAG;

}

/\*\*

\* @brief Checks whether the specified SPIx/I2Sx interrupt has occurred or not.

\* @param SPIx: To select the SPIx/I2Sx peripheral, where x can be: 1, 2, 3, 4, 5 or 6

\* in SPI mode or 2 or 3 in I2S mode or I2Sxext for I2S full duplex mode.

\* @param SPI\_I2S\_IT: specifies the SPI interrupt source to check.

\* This parameter can be one of the following values:

\* @arg SPI\_I2S\_IT\_TXE: Transmit buffer empty interrupt.

\* @arg SPI\_I2S\_IT\_RXNE: Receive buffer not empty interrupt.

\* @arg SPI\_I2S\_IT\_OVR: Overrun interrupt.

\* @arg SPI\_IT\_MODF: Mode Fault interrupt.

\* @arg SPI\_IT\_CRCERR: CRC Error interrupt.

\* @arg I2S\_IT\_UDR: Underrun interrupt.

\* @arg SPI\_I2S\_IT\_TIFRFE: Format Error interrupt.

\* @retval The new state of SPI\_I2S\_IT (SET or RESET).

\*/

ITStatus SPI\_I2S\_GetITStatus(SPI\_TypeDef\* SPIx, uint8\_t SPI\_I2S\_IT)

{

ITStatus bitstatus = RESET;

uint16\_t itpos = 0, itmask = 0, enablestatus = 0;

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH\_EXT(SPIx));

assert\_param(IS\_SPI\_I2S\_GET\_IT(SPI\_I2S\_IT));

/\* Get the SPI\_I2S\_IT index \*/

itpos = 0x01 << (SPI\_I2S\_IT & 0x0F);

/\* Get the SPI\_I2S\_IT IT mask \*/

itmask = SPI\_I2S\_IT >> 4;

/\* Set the IT mask \*/

itmask = 0x01 << itmask;

/\* Get the SPI\_I2S\_IT enable bit status \*/

enablestatus = (SPIx->CR2 & itmask) ;

/\* Check the status of the specified SPI interrupt \*/

if (((SPIx->SR & itpos) != (uint16\_t)RESET) && enablestatus)

{

/\* SPI\_I2S\_IT is set \*/

bitstatus = SET;

}

else

{

/\* SPI\_I2S\_IT is reset \*/

bitstatus = RESET;

}

/\* Return the SPI\_I2S\_IT status \*/

return bitstatus;

}

/\*\*

\* @brief Clears the SPIx CRC Error (CRCERR) interrupt pending bit.

\* @param SPIx: To select the SPIx/I2Sx peripheral, where x can be: 1, 2, 3, 4, 5 or 6

\* in SPI mode or 2 or 3 in I2S mode or I2Sxext for I2S full duplex mode.

\* @param SPI\_I2S\_IT: specifies the SPI interrupt pending bit to clear.

\* This function clears only CRCERR interrupt pending bit.

\* @arg SPI\_IT\_CRCERR: CRC Error interrupt.

\*

\* @note OVR (OverRun Error) interrupt pending bit is cleared by software

\* sequence: a read operation to SPI\_DR register (SPI\_I2S\_ReceiveData())

\* followed by a read operation to SPI\_SR register (SPI\_I2S\_GetITStatus()).

\* @note UDR (UnderRun Error) interrupt pending bit is cleared by a read

\* operation to SPI\_SR register (SPI\_I2S\_GetITStatus()).

\* @note MODF (Mode Fault) interrupt pending bit is cleared by software sequence:

\* a read/write operation to SPI\_SR register (SPI\_I2S\_GetITStatus())

\* followed by a write operation to SPI\_CR1 register (SPI\_Cmd() to enable

\* the SPI).

\* @retval None

\*/

void SPI\_I2S\_ClearITPendingBit(SPI\_TypeDef\* SPIx, uint8\_t SPI\_I2S\_IT)

{

uint16\_t itpos = 0;

/\* Check the parameters \*/

assert\_param(IS\_SPI\_ALL\_PERIPH\_EXT(SPIx));

assert\_param(IS\_SPI\_I2S\_CLEAR\_IT(SPI\_I2S\_IT));

/\* Get the SPI\_I2S IT index \*/

itpos = 0x01 << (SPI\_I2S\_IT & 0x0F);

/\* Clear the selected SPI CRC Error (CRCERR) interrupt pending bit \*/

SPIx->SR = (uint16\_t)~itpos;

}

/\*\*

\* @}

\*/

/\*\*

\* @}

\*/

/\*\*

\* @}

\*/

/\*\*

\* @}

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* (C) COPYRIGHT STMicroelectronics \*\*\*\*\*END OF FILE\*\*\*\*/