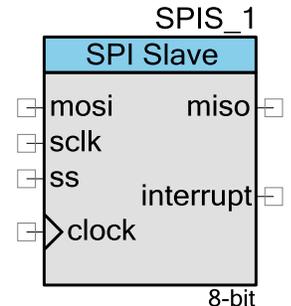


# Serial Peripheral Interface (SPI) Slave

1.20

## Features

- 2 to 16-bit Data Width
- 4 SPI Modes
- Data Rates to 33Mb/s



## General Description

The SPI Slave provides an industry-standard 4-wire Slave SPI interface. The interface supports all 4 SPI operating modes allowing interface with any SPI Master device. In addition to the standard 8-bit interface the SPI Slave supports a configurable 2 to 16-bit interface for interfacing to nonstandard SPI word lengths. SPI signals include the standard SCLK, MISO and MOSI pins and SS signal.

## When to use the SPI Slave

The SPI Slave component should be used any time the PSoC device is required to interface with a SPI Master device. In addition to 'SPI Master' labeled devices the SPI Slave can be used with many devices implementing a shift register type interface.

The SPI Master component should be used in instances requiring the PSoC device to interface with a SPI Slave device. The Shift Register component should be used in situations where its low level flexibility provides hardware capabilities not available in the SPI Slave component.

## Input/Output Connections

This section describes the various input and output connections for the SPI. An asterisk (\*) in the list of I/O's states that the I/O may be hidden on the symbol under the conditions listed in the description of that I/O.

### clock – Input

The clock input defines the sampling rate of the status register. All data clocking happens on sclk so the clock input DOES NOT handle the bit-rate of the SPI Slave. This input is always visible and must be connected.

**PRELIMINARY**

## **miso – Output**

The miso output carries the slave output – master input serial data to the master device on the bus. This output is always visible and must be connected for TX operations.

## **mosi – Input**

The mosi input carries the master output – slave input serial data from the master device on the bus. This input is always visible and must be connected.

## **sclk– Input**

The sclk input provides the slave synchronization clock input to this device. This input is always visible and must be connected.

## **ss – Input**

The ss input carries the slave select signal to this device. This input is always visible and must be connected.

## **interrupt – Output**

The interrupt output is the logical OR of the group of possible interrupt sources. This signal will go high while any of the enabled interrupt sources are true.

# **Parameters and Setup**

Drag an SPI Master component onto your design. Double-click component symbol to open the Configure dialog.

If the component will be used to communicate with an external SPI Master device, then connect the appropriate digital input and output Pins components.

**Note** Configure the Pins components connected to the MISO, SCLK and SS inputs/outputs to unselect the **Input Synchronized** parameter (under the Pins component **Input** tab) to prevent incorrect data sampling.

The following sections describe the SPI Slave parameters, and how they are configured using the dialog. They also indicate whether the options are hardware or software.

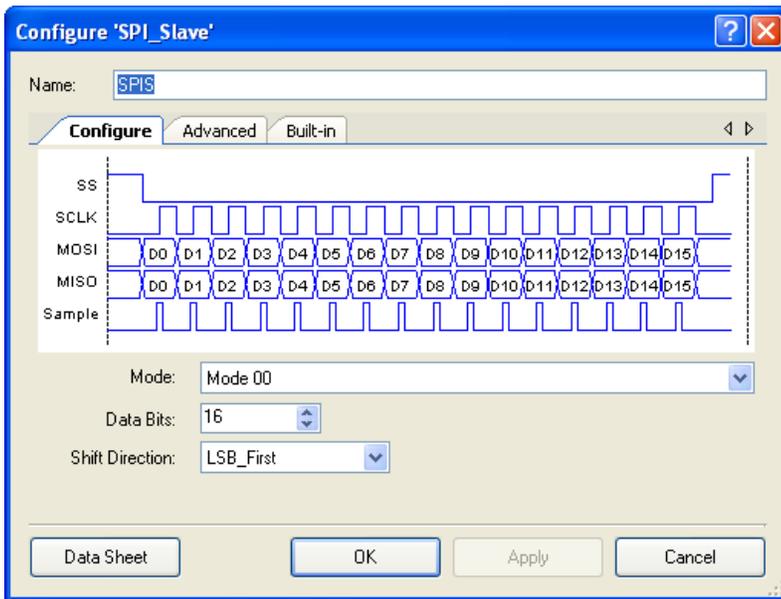
## **Hardware vs. Software Options**

Hardware configuration options change the way the project is synthesized and placed in the hardware. You must rebuild the hardware if you make changes to any of these options. Software configuration options do not affect synthesis or placement. When setting these parameters before build time you are setting their initial value which may be modified at any time with the API provided. Hardware only parameters are marked with an asterisk (\*).

**PRELIMINARY**



## Configure Tab



These are basic parameters expected for every SPI component and are therefore the first parameters visible to configure.

### Mode \*

The **Mode** parameter defines the desired clock phase and clock polarity mode used in the communication. The options are “Mode 00” (default), “Mode 01”, “Mode 10” and “Mode 11” which are defined in the implementation details below.

### Data Bits \*

The number of data bits defines the bit-width of a single transfer as transferred with the WriteByte() and ReadByte() API. The default number of bits is a single byte (8-bits). Any integer from 2 to 16 may be selected

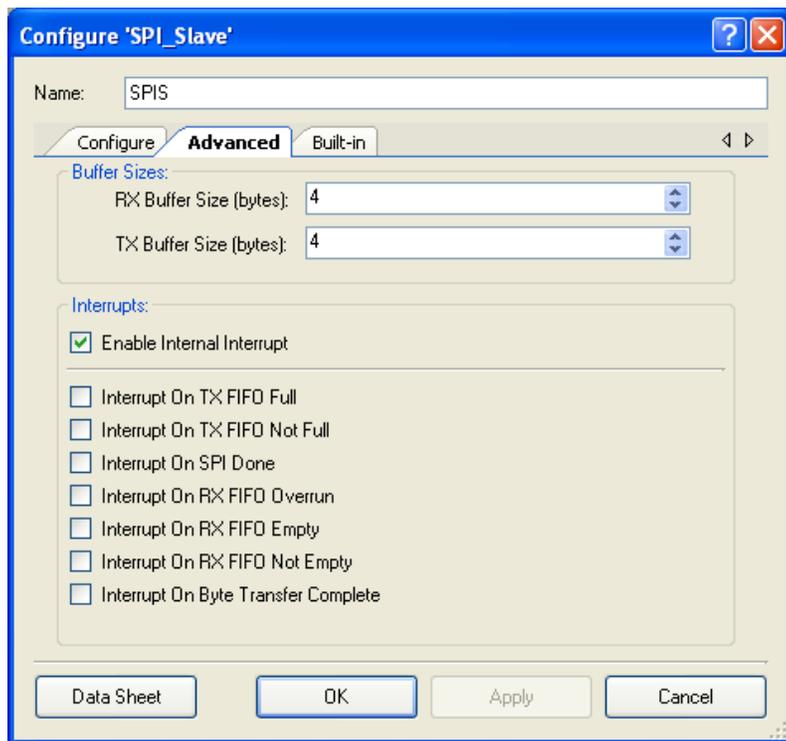
### Shift Direction \*

The **Shift Direction** parameter defines the direction the serial data is transmitted. When set to MSB\_First (default) the Most Significant bit is transmitted first through to the Least Significant bit. This is implemented by shifting the data left. LSB\_First is the exact opposite.



PRELIMINARY

## Advanced Tab



### RxBufferSize \*

The RX Buffer Size parameter defines the size (in bytes) of memory allocated for a circular data buffer. If this parameter is set to 1 a single byte FIFO is implemented in the hardware. If the parameter is set to 2-4 then the 4-byte FIFO is implemented in hardware. All other values up to 255 (8-bit Processor) or 64535 (32-bit Processor) will use the 4-byte FIFO and a memory array controlled by the supplied API. The default value is 8.

### TxBufferSize \*

The TX Buffer Size parameter defines the size (in bytes) of memory allocated for a circular data buffer. If this parameter is set to 1 a single byte FIFO is implemented in the hardware. If the parameter is set to 2-4 then the 4-byte FIFO is implemented in hardware. All other values up to 255 (8-bit Processor) or 64535 (32-bit Processor) will use the 4-byte FIFO and a memory array controlled by the supplied API. The default value is 8.

### Enable Internal Interrupt

The Enable Internal Interrupt option allows the user to use the predefined ISR of the SPI Slave component. The user may add to this ISR if selected or deselect the internal interrupt and handle the ISR with an external interrupt component connected to the interrupt output of the SPI Slave.

**PRELIMINARY**



If the user selects a RX or TX buffer size greater than 4 this parameter is set automatically as the internal ISR is needed to handle transferring data from the FIFO to the RX and/or TX buffer. At all times the interrupt output pin of the SPI Slave is visible and useable, outputting the same signal that goes to the internal interrupt based on the selected status interrupts. This output may then be used as a DMA request source to DMA from the RX or TX buffer independent of the interrupt or as another interrupt dependant upon the desired functionality.

### Interrupts

The interrupts selection parameters allow the user to configure the internal events that are allowed to cause an interrupt. Interrupt generation is a masked OR of all of the status register bits. The bit's chosen with these parameters defines the mask implemented at the initial configuration of this component.

## Clock Selection

The external clock input to the SPI Slave is only fed to the status register. The Bit-Rate is defined from the sclk input from the master device.

## Placement

The SPI Slave component is placed throughout the UDB array and all placement information is provided to the API through the cyfitter.h file.

## Resources

Resolution	Digital Blocks					API Memory (Bytes)		Pins (per External I/O)
	Datapaths	Macro cells	Status Registers	Control Registers	Counter7	Flash	RAM	
SPI Slave 8-bit	2	*	1	0	1			*
SPI Slave 16-bit	4	*	1	0	1			*

\* Unknown



**PRELIMINARY**

## Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent sections cover each function in more detail.

By default, PSoC Creator assigns the instance name "SPIS\_1" to the first instance of a component in a given design. You can rename the instance to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is "SPIS."

Function	Description
void SPIS_Start(void)	Enable the SPIS operation.
void SPIS_Stop(void)	Disable the SPIS operation.
void SPIS_EnableInt (void)	Enables the internal interrupt irq.
void SPIS_DisableInt (void)	Disables the internal interrupt irq
void SPIS_SetInterruptMode (uint8 interrupt)	Configures the interrupt sources enabled
uint8 SPIS_ReadStatus (void)	Returns the current state of the status register
void SPIS_WriteByte (uint8/16 byte)	Places a byte in the transmit buffer which will be sent at the next available bus time
void SPIS_WriteByteZero(uint8/16 byte)	Places a byte in the shift register directly. This is required for SPI Modes 00 and 01.
uint8/16 SPIS_ReadByte (void)	Returns the next byte of received data
uint8/uint16 SPIS_GetRxBufferSize (void)	Returns the size (in bytes) of the RX memory buffer
uint8/uint16 SPIS_GetTxBufferSize (void)	Returns the size (in bytes) of the TX memory buffer
void SPIS_ClearRxBuffer (void)	Clears the memory array of all received data
void SPIS_ClearTxBuffer (void)	Clears the memory array of all transmit data
void SPIS_TxEnable (void)	Enables the TX portion of the SPI Slave (MOSI)
void SPIS_TxDisable (void)	Disables the TX portion of the SPI Slave (MOSI)
void SPIS_PutArray (uint16* RamString, uint8 ByteCount)	Places an array of data into the transmit buffer
void SPIS_ClearFIFO(void)	Clears any received data from the RX FIFO

**PRELIMINARY**



## void SPIS\_Start(void)

<b>Description:</b>	Only necessary for initial configuration.
<b>Parameters:</b>	void
<b>Return Value:</b>	void
<b>Side Effects:</b>	The first time this function is called it initializes all of the necessary parameters for execution. i.e. setting the initial interrupt mask, configuring the interrupt service routine, configuring the bit-counter parameters and clearing the RX FIFO

## void SPIS\_Stop(void)

<b>Description:</b>	Has no affect on the SPIS operation
<b>Parameters:</b>	void
<b>Return Value:</b>	void
<b>Side Effects:</b>	None

## void SPIS\_EnableInt (void)

<b>Description:</b>	Enables the internal interrupt irq
<b>Parameters:</b>	void
<b>Return Value:</b>	void
<b>Side Effects:</b>	None

## void SPIS\_DisableInt (void)

<b>Description:</b>	Disables the internal interrupt irq
<b>Parameters:</b>	void
<b>Return Value:</b>	void
<b>Side Effects:</b>	None

## void SPIS\_SetInterruptMode (uint8 interrupt)

<b>Description:</b>	Configures the interrupt sources enabled
<b>Parameters:</b>	uint8: Bit-Field containing the interrupts to enable. Based on the bit-field arrangement of the status register. This value must be a combination of status register bit-masks defined in the header file.
<b>Return Value:</b>	void
<b>Side Effects:</b>	None



**PRELIMINARY**

## uint8 SPIS\_ReadStatus (void)

**Description:** Returns the current state of the status register  
**Parameters:** void  
**Return Value:** uint8: Current status register value  
**Side Effects:** Status register bits are clear on read.

## void SPIS\_WriteByte (uint8/16 byte)

**Description:** Places a byte in the transmit buffer which will be sent at the next available bus time  
**Parameters:** uint8/16: data byte  
**Return Value:** void  
**Side Effects:** Data may be placed in the memory buffer and will not be transmitted until all other data has been transmitted. This function blocks until there is space in the output memory buffer.

## void SPIS\_WriteByteZero (uint8/16 byte)

**Description:** Places a byte directly into the shift register for transmit which will be sent during the next clock phase from the master device  
**Parameters:** uint8/16: data byte  
**Return Value:** void  
**Side Effects:** Required for Modes 00 and 01 where data must be in the shift register before the first clock edge. Firmware must control this if there is already data being shifted out and if there is more data in the FIFO.

## uint8/16 SPIS\_ReadByte (void)

**Description:** Returns the next byte of received data  
**Parameters:** void  
**Return Value:** uint8/16: data byte  
**Side Effects:** This function blocks until there is data in the input memory buffer.

## uint8/uint16 SPIS\_GetRxBufferSize (void)

**Description:** Returns the number of bytes/words of data currently held in the RX buffer  
**Parameters:** void  
**Return Value:** uint8/uint16: Integer count of the number of bytes/words in the RX buffer  
**Side Effects:** None

## uint8/uint16 SPIS\_GetTxBufferSize (void)

**Description:** Returns the number of bytes/words of data currently held in the TX buffer  
**Parameters:** void  
**Return Value:** uint8/uint16: Integer count of the number of bytes/words in the RX buffer  
**Side Effects:** None

PRELIMINARY



**void SPIS\_ClearRxBuffer (void)**

**Description:** Clears the memory array of all received data  
**Parameters:** void  
**Return Value:** void  
**Side Effects:** None

**void SPIS\_ClearTxBuffer (void)**

**Description:** Clears the memory array of all transmit data  
**Parameters:** void  
**Return Value:** void  
**Side Effects:** Will not clear data already placed in the TX FIFO.

**void SPIS\_TxEnable (void)**

**Description:** Enables the TX portion of the SPI Slave (MISO)  
**Parameters:** void  
**Return Value:** void  
**Side Effects:**

**void SPIS\_TxDisable (void)**

**Description:** Disables the TX portion of the SPI Slave (MISO)  
**Parameters:** void  
**Return Value:** void  
**Side Effects:** None

**void SPIS\_PutArray (uint16\* RamString, uint8 ByteCount)**

**Description:** Places an array of data into the transmit buffer  
**Parameters:** uint16\*: RamString – Location of the first byte of the data to move to the transmit buffer  
uint8: Byte Count – Number of bytes in the array.  
**Return Value:** void  
**Side Effects:** None

**void SPIS\_ClearFIFO (void)**

**Description:** Clears any received data from the RX FIFO  
**Parameters:** void  
**Return Value:** void  
**Side Effects:** None

**PRELIMINARY**

## Defines

### SPIS\_INIT\_INTERRUPTS\_MASK

Defines the initial configuration of the interrupt sources chosen in the configuration GUI. This is a mask of the bits in the status register that have been enabled at configuration as sources for the interrupt.

## Status Register Bits

**Table 1 SPIS\_STATUS**

Bits	7	6	5	4	3	2	1	0
Value	Unused	Byte Complete	RX FIFO Overrun	RX FIFO Empty	RX FIFO Not Empty	TX FIFO Full	TX FIFO Not Full	SPI Done

- Byte Complete: Set when a Byte has been transmitted.
- RX FIFO Overrun: Set when RX Data has overrun the 4 byte FIFO or 1 Byte FIFO without being moved to the Memory array (if one exists)
- RX FIFO Empty: Set when the RX Data FIFO is empty (Does not indicate the RAM array conditions)
- RX FIFO Not Empty: Set when the RX Data FIFO is full (Does not indicate the RAM array conditions)
- TX FIFO Full: Set when the TX Data FIFO is full (Does not indicate the RAM array conditions):
- TX FIFO Not Full: Set when the TX Data FIFO is empty (Does not indicate the RAM array conditions):
- SPI Done: Set when all of the data in the transmit FIFO has been sent. This may be used to signal a transfer complete instead of using the byte complete status.

### SPIS\_TXBUFFERSIZE

Defines the amount of memory to allocate for the TX memory array buffer. This does not include the 4 bytes included in the FIFO. If this value is greater than 4, interrupts are implemented which move data to the FIFO from the circular memory buffer automatically.

### SPIS\_RXBUFFERSIZE

Defines the amount of memory to allocate for the RX memory array buffer. This does not include the 4 bytes included in the FIFO. If this value is greater than 4, interrupts are implemented which move data from the FIFO to the circular memory buffer automatically.

**PRELIMINARY**



## SPIS\_DATAWIDTH

Defines the number of bits per data transfer chosen by the user.

## Sample Firmware Source Code

The following is a C language example demonstrating the basic functionality of the SPI Slave component. This example assumes the component has been placed in a design with the default name "SPIS\_1."

**Note** If you rename your component you must also edit the example code as appropriate to match the component name you specify.

### Mode 00 or Mode 01

```

#include <device.h>
void main()
{
    uint8 i = 0;
    uint8 val[4];

    SPIS_1_Start();
    /* Preload the FIFO with TX data up to 4 bytes */
    SPIS_1_WriteByteZero(0xA3);
    SPIS_1_WriteByte(0xE7);
    SPIS_1_WriteByte(0x96);
    SPIS_1_WriteByte(0x28);

    /* Read the four bytes transmitted from the master */
    for(i=0;i<4;i++)
        val[i] = SPIS_1_ReadByte();
}

```

### Mode 10 or Mode 11

```

#include <device.h>
void main()
{
    uint8 i = 0;
    uint8 val[4];

    SPIS_1_Start();
    /* Preload the FIFO with TX data up to 4 bytes */
    SPIS_1_WriteByte(0xA3);
    SPIS_1_WriteByte(0xE7);
    SPIS_1_WriteByte(0x96);
    SPIS_1_WriteByte(0x28);

    /* Read the four bytes transmitted from the master */
    for(i=0;i<4;i++)
        val[i] = SPIS_1_ReadByte();
}

```



**PRELIMINARY**

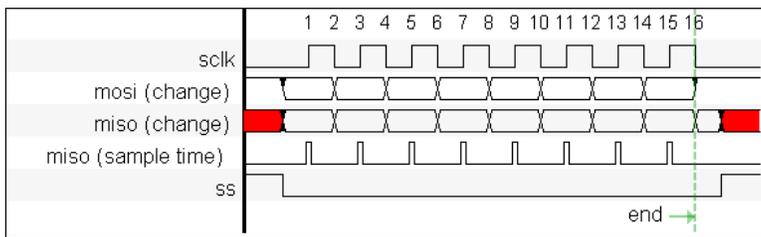
# Functional Description

## Default Configuration

The default configuration for the SPIS is as an 8-bit SPIS with Mode 00 configuration.

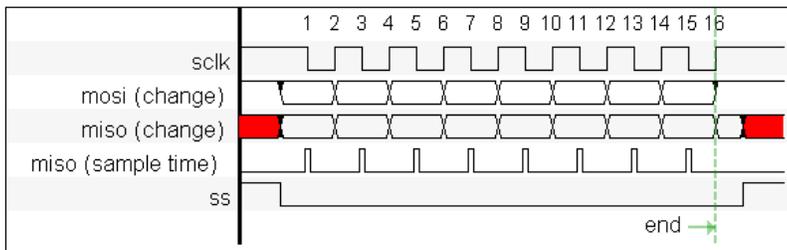
### SPIS Mode: 00

Mode 00 defines the Clock Phase of 0 and the Clock Polarity of 0 which has the following characteristics:



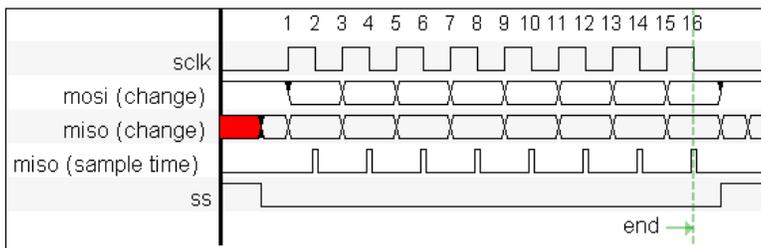
### SPIS Mode: 01

Mode 01 defines the Clock Phase of 0 and the Clock Polarity of 1 which has the following characteristics:



### SPIS Mode: 10

Mode 10 defines the Clock Phase of 1 and a Clock Polarity of 0 which has the following characteristics:

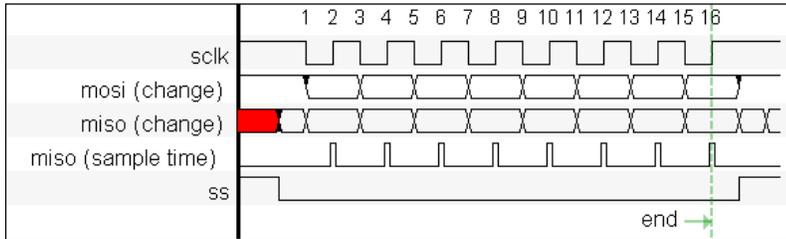


**PRELIMINARY**



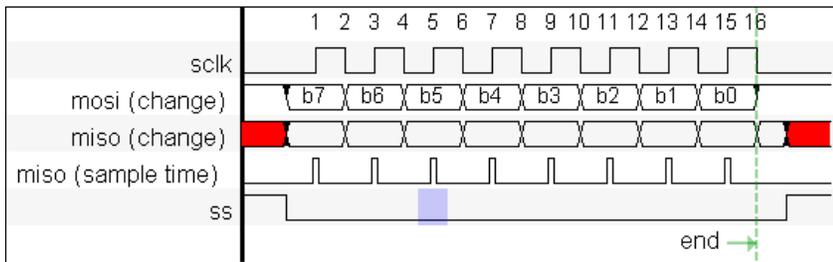
### SPIS Mode: 11

Mode 11 defines the Clock Phase of 1 and a Clock Polarity of 1 which has the following characteristics:



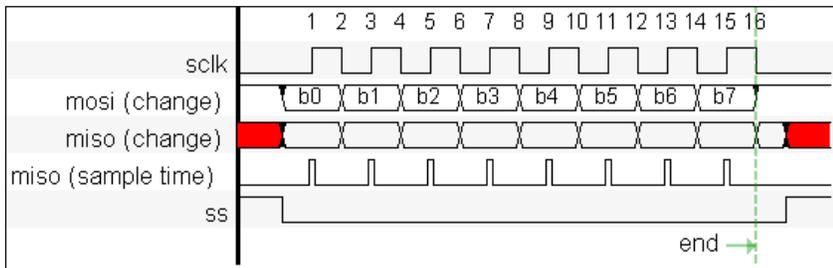
### SPIS ShiftDir: MSB\_First

When setting the Shift Direction parameter to MSB\_First the data is shifted out Most Significant bit first. For an 8-bit Transfer with Mode 00 the transfer looks like this:



### SPIS ShiftDir: LSB\_First

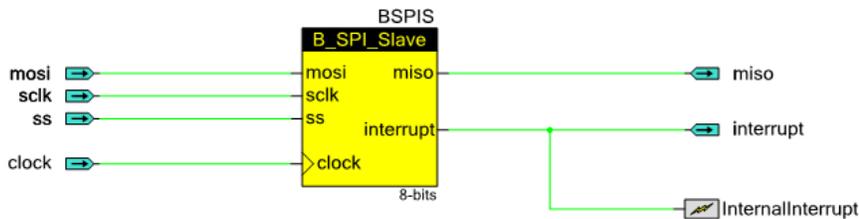
When setting the Shift Direction parameter to LSB\_First the data is shifted out Least Significant bit first. For an 8-bit Transfer with Mode 00 the transfer looks like this:



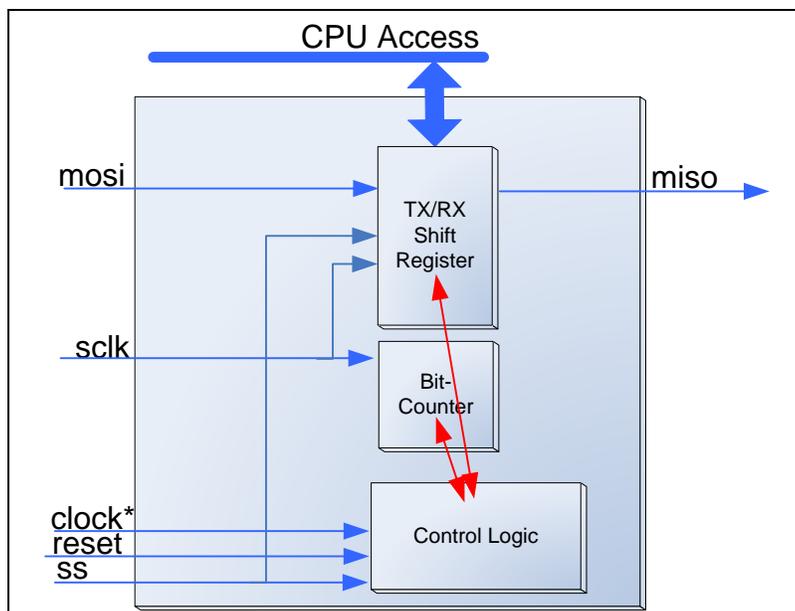
PRELIMINARY

## Block Diagram and Configuration

The SPIS is only available as a UDB configuration of blocks. The API is described above and the registers are described here to define the overall implementation of the SPIS.



The implementation is described in the following block diagram.



## Registers

### Status

The status register is a read only register which contains the various status bits defined for the SPIS. The value of this registers is available with the SPIS\_ReadStatus() and function call. The interrupt output signal is generated from an ORing of the masked bit-fields within the status register. You can set the mask using the SPIS\_SetInterruptMode() function call and upon receiving an interrupt you can retrieve the interrupt source by reading the Status register with the SPIS\_ReadStatus() function call.

**PRELIMINARY**



The Status register is clear on read so the interrupt source is held until the SPIS\_ReadStatus() function is called. All operations on the status register must use the following defines for the bit-fields as these bit-fields may be moved around within the status register at build time.

There are several bit-fields masks defined for the status registers. Any of these bit-fields may be included as an interrupt source. The bit-fields indicated with an \* are configured as sticky bits in the status register, all other bits are configured as real-time indicators of status. The #defines are available in the generated header file (.h) as follows:

**SPIS\_STS\_SPI\_DONE \***

Defined as the bit-mask of the Status register bit "SPI Done."

**SPIS\_STS\_TX\_FIFO\_NOT\_FULL**

Defined as the bit-mask of the Status register bit "Transmit FIFO Empty."

**SPIS\_STS\_TX\_FIFO\_FULL**

Defined as the bit-mask of the Status register bit "Transmit FIFO Full."

**SPIS\_STS\_RX\_FIFO\_NOT\_EMPTY**

Defined as the bit-mask of the Status register bit "Receive FIFO Full."

**SPIS\_STS\_RX\_FIFO\_EMPTY**

Defined as the bit-mask of the Status register bit "Receive FIFO Empty."

**SPIS\_STS\_RX\_FIFO\_OVERRUN \***

Defined as the bit-mask of the Status register bit "Receive FIFO Overrun."

**SPIS\_STS\_BYTE\_COMPLETE \***

Defined as the bit-mask of the Status register bit "Byte Complete."

**TX Data**

The TX data register contains the transmit data value to send. This is implemented as a FIFO in the SPIS. There is a software state machine to control data from the transmit memory buffer to handle much larger portions of data to be sent. All API dealing with the transmitting of data must go through this register to place the data onto the bus. If there is data in this register and flow control indicates that data can be sent, then the data will be transmitted on the bus. As soon as this register (FIFO) is empty no more data will be transmitted on the bus until it is added to the FIFO. DMA may be setup to fill this FIFO when empty using the TX\_DATA\_ADDR address defined in the header file.

**PRELIMINARY**

## RX Data

The RX data register contains the received data. This is implemented as a FIFO in the SPIS. There is a software state machine to control data movement from this receive FIFO into the memory buffer. Typically the RX interrupt will indicate that data has been received at which time that data has several routes to the firmware. DMA may be setup from this register to the memory array or the firmware may simply poll for the data at will. This will use the RX\_DATA\_ADDR address defined in the header file.

## Conditional Compilation Information

The SPIS requires only one conditional compile definition to handle the 8 or 16 bit Datapath configuration necessary to implement the expected NumberOfDataBits configuration it must support. It is required that the API conditionally compile Data Width defined in the parameter chosen. The API should never use these parameters directly but should use the define listed below.

### SPIS\_DATAWIDTH

This defines how many data bits will make up a single “byte” transfer.

## References

Not applicable

## DC and AC Electrical Characteristics

The following values are indicative of expected performance and based on initial characterization data.

### 5.0V/3.3V DC and AC Electrical Characteristics

Parameter	Typical	Min	Max	Units	Conditions and Notes
Input					
Input Voltage Range	---		Vss to Vdd	V	
Input Capacitance	---		---	pF	
Input Impedance	---		---	Ω	
Maximum Clock Rate	---		67	MHz	

**PRELIMINARY**



## Component Changes

This section lists the major changes in the component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
1.20.d	Minor datasheet edits.	
1.20.c	Minor datasheet edits.	
1.20.b	Moved component into subfolders of the component catalog.	
1.20.a	Data sheet change to note that the #define ANT_SPI_STS_TX_FIFO_EMPTY_SHIFT 0x04u is incorrect in the component.	This bit in the status register is actually RX_FIFO_EMPTY. The version 1.20 SPIS component will not be changed but version 1.50 will. The data sheet is being changed to help users avoid using the bit-field for the incorrect use.
1.20	Updated the component symbol; updated descriptions for I/Os and parameters	Symbol was updated to comply with corporate standard

© Cypress Semiconductor Corporation, 2009-2016. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit [cypress.com](http://cypress.com). Other names and brands may be claimed as property of their respective owners.



**PRELIMINARY**