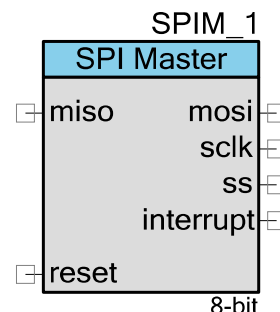


# Serial Peripheral Interface (SPI) Master

1.20

## Features

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



## General Description

The SPI master provides an industry-standard 4-wire master SPI interface. The interface supports all 4 SPI operating modes, allowing interface with any SPI slave device. In addition to the standard 8-bit interface, the SPI Master 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 multiple Slave Select (SS) signal generation.

## When to use the SPI Master

The SPI master component should be used any time the PSoC device is required to interface with one or more SPI slave devices. In addition to 'SPI slave' labeled devices, the SPI master can be used with many devices implementing a shift register type interface.

The SPI slave component should be used in instances requiring the PSoC device to interface with a SPI master device. The Shift Register component should be used in situations where its low level flexibility provides hardware capabilities not available in the SPI Master 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 bit-rate of the serial communication. The bit-rate is 1/2 the input clock frequency. This input is visible if the "External Clock" parameter is selected. If "Internal Clock" is used, then you define the desired data bit-rate and the clock needed is solved by PSoC Creator.

**PRELIMINARY**

## reset – Input

Resets the SPI state machine to the idle state. This will throw out any data that was currently being transmitted or received but will not clear data from the FIFO that has already been received or is ready to be transmitted.

## miso – Input

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

## mosi – Output

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

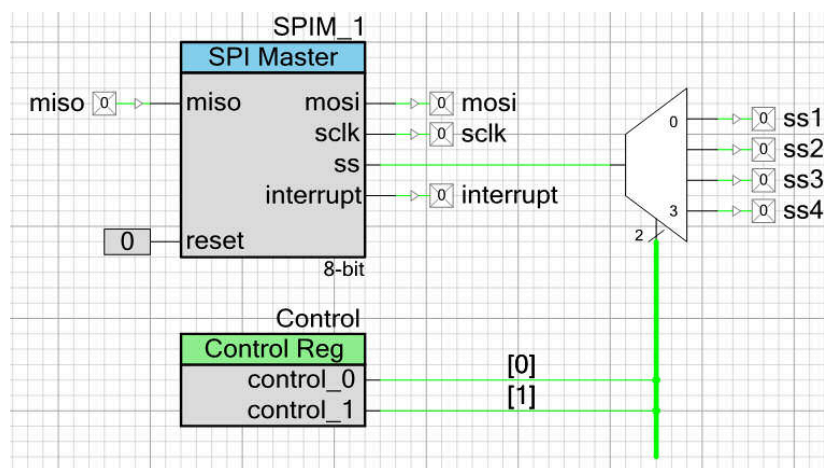
## sclk– Output

The sclk output carries the master synchronization clock output to the slave device(s) on the bus. This output is always visible and must be connected.

## ss – Output

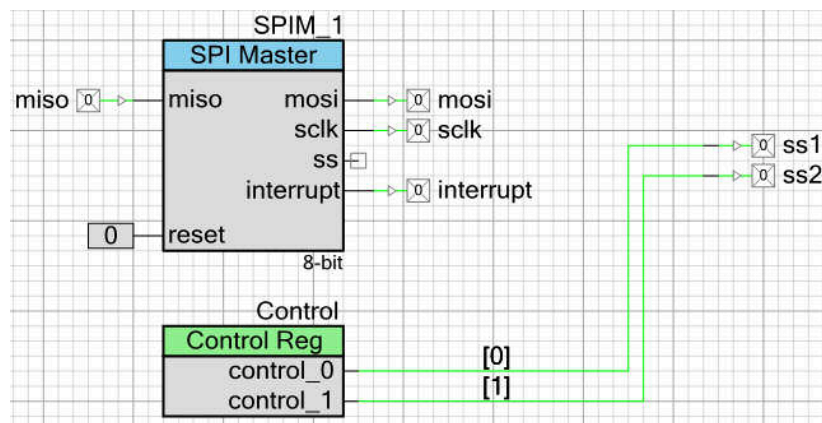
The ss output carries the slave select signal to a slave device(s) on the bus. It is also possible to connect a digital De-Multiplexer to handle multiple slave devices, or to have a completely firmware controlled Slave select.

**Figure 1: Slave Select Output to De-Multiplexer**



PRELIMINARY



**Figure 2: Firmware Controlled Slave Select(s)**

## 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 for communication with one or more external SPI Slave devices, then connect the appropriate digital input and output pins.

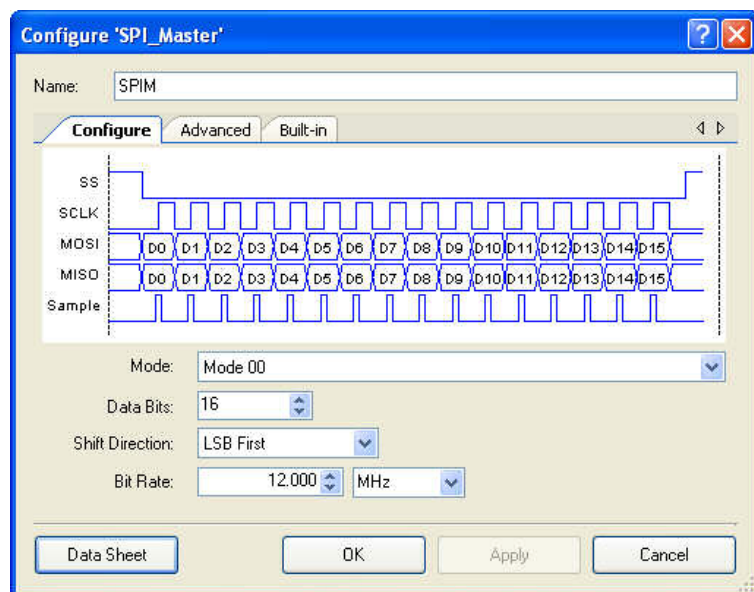
**Note** Configure the Pins component connected to the MISO input to unselect the **Input Synchronized** parameter (under the Pins component **Input** tab) to prevent incorrect data sampling.

The following sections describe the SPI Master 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 (\*).

## 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”, “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 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.

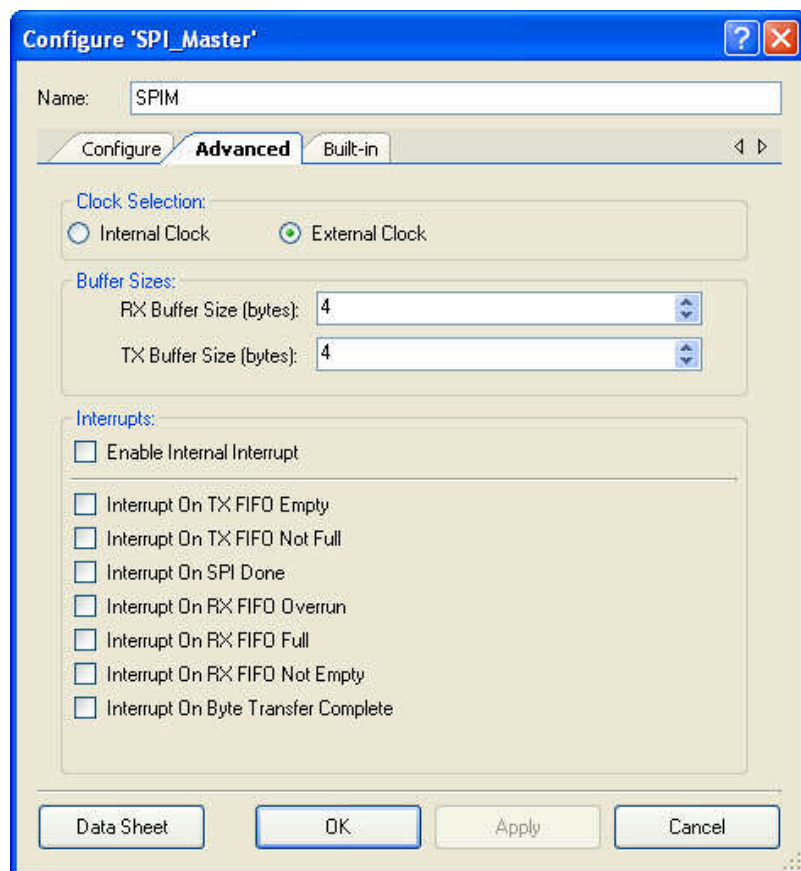
### Bit Rate \*

The Bit-Rate parameter defines the communication speed in Hertz. If the internal clock is selected this parameter will define the clock frequency of the internal clock as 2X the bit rate. This parameter has no affect if the external clock option is set.

**PRELIMINARY**



## Advanced Tab



### Clock Selection \*

The Clock Selection parameter allows the user to choose between an internally configured clock or an externally configured clock or I/O for the data-rate generation. When set to “Internal Clock” the required clock frequency is calculated and configured by PSoC Creator based on the “Bit Rate” parameter. When set to “External Clock” the component does not control the data-rate but can calculate the expected bit-rate.

If this parameter is “Internal Clock” then the clock input is not visible on the symbol.

### Rx Buffer Size \*

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.



**PRELIMINARY**

### **Tx Buffer Size \***

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.

### **Enable Internal Interrupt**

The Enable Internal Interrupt option allows the user to use the predefined ISR of the SPI Master 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 Master.

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 master 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**

When the internal clock configuration is selected PSoC Creator will calculate the needed frequency and clock source and will generate the resource needed for implementation. Otherwise, you must supply the clock and calculate the bit-rate at 1/2 the input clock frequency.

## **Placement**

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

**PRELIMINARY**



## Resources

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

\* Unknown

## 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 "SPIM\_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 "SPIM".

Function	Description
void SPIM_Start(void)	Enable the SPIM operation.
void SPIM_Stop(void)	Disable the SPIM operation.
void SPIM_EnableInt (void)	Enables the internal interrupt irq.
void SPIM_DisableInt (void)	Disables the internal interrupt irq
void SPIM_SetInterruptMode (uint8 interrupt)	Configures the interrupt sources enabled
uint8 SPIM_ReadStatus (void)	Returns the current state of the status register
void SPIM_WriteByte (uint8/16 byte)	Places a byte in the transmit buffer which will be sent at the next available bus time
uint8/16 SPIM_ReadByte (void)	Returns the next byte of received data
uint8/uint16 SPIM_GetRxBufferSize (void)	Returns the size (in bytes) of the RX memory buffer
uint8/uint16 SPIM_GetTxBufferSize (void)	Returns the size (in bytes) of the TX memory buffer
void SPIM_ClearRxBuffer (void)	Clears the memory array of all received data



**PRELIMINARY**



Function	Description
void SPIM_ClearTxBuffer (void)	Clears the memory array of all transmit data
void SPIM_TxEnable (void)	Enables the TX portion of the SPI Master (MOSI)
void SPIM_TxDisable (void)	Disables the TX portion of the SPI Master (MOSI)
void SPIM_PutArray (uint16* RamString, uint8 ByteCount)	Places an array of data into the transmit buffer
void SPIM_ClearFIFO(void)	Clears any received data from the RX FIFO

## void SPIM\_Start(void)

**Description:** Enable the SPIM operation by enabling the internal clock. If external clock is selected then this function is only necessary for initial configuration.

**Parameters:** void

**Return Value:** void

**Side Effects:** 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 SPIM\_Stop(void)

**Description:** Disable the SPIM operation by disabling the internal clock. If external clock is selected then this function has no affect on the SPIM operation

**Parameters:** void

**Return Value:** void

**Side Effects:** None

## void SPIM\_EnableInt (void)

**Description:** Enables the internal interrupt irq

**Parameters:** void

**Return Value:** void

**Side Effects:** None

**PRELIMINARY**





**void SPIM\_DisableInt (void)**

**Description:** Disables the internal interrupt irq

**Parameters:** void

**Return Value:** void

**Side Effects:** None

**void SPIM\_SetInterruptMode (uint8 interrupt)**

**Description:** Configures the interrupt sources enabled

**Parameters:** 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.

**Return Value:** void

**Side Effects:** None

**uint8 SPIM\_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 SPIM\_WriteByte (uint8/16 byte)**

**Description:** Places a byte in the transmit buffer which will be sent at the next available SPI 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.

**PRELIMINARY**

**uint8/16 SPIM\_ReadByte (void)**

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

**uint8/uint16 SPIM\_GetRxBufferSize (void)**

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

**uint8/uint16 SPIM\_GetTxBufferSize (void)**

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

**void SPIM\_ClearRxBuffer (void)**

<b>Description:</b>	Clears the memory array and RX FIFO of all received data
<b>Parameters:</b>	void
<b>Return Value:</b>	void
<b>Side Effects:</b>	None

**void SPIM\_ClearTxBuffer (void)**

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

**PRELIMINARY**

**void SPIM\_TxEnable (void)**

<b>Description:</b>	Enables the TX portion of the SPI Master (MOSI)
<b>Parameters:</b>	void
<b>Return Value:</b>	void
<b>Side Effects:</b>	None

**void SPIM\_TxDisable (void)**

<b>Description:</b>	Disables the TX portion of the SPI Master (MOSI)
<b>Parameters:</b>	void
<b>Return Value:</b>	void
<b>Side Effects:</b>	None

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

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

**void SPIM\_ClearFIFO (void)**

<b>Description:</b>	Clears any received data from the RX FIFO
<b>Parameters:</b>	void
<b>Return Value:</b>	void
<b>Side Effects:</b>	None

**Defines****SPIM\_INIT\_INTERRUPTS\_MASK**

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

**PRELIMINARY**

## Status Register Bits

**Table 1 SPIM\_STATUS**

Bits	7	6	5	4	3	2	1	0
Value	Unused	Byte Complete	RX Buf. Overrun	RX FIFO Not Empty	RX FIFO Full	TX FIFO Not Full	TX FIFO Empty	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 Not Empty: Set when the RX Data FIFO is not empty (Does not indicate the RAM array conditions)
- RX FIFO Full: Set when the RX Data FIFO is full (Does not indicate the RAM array conditions)
- TX FIFO Not Full: Set when the TX Data FIFO is full (Does not indicate the RAM array conditions):
- TX FIFO Empty: 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.

### SPIM\_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.

### SPIM\_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.

### SPIM\_DATAWIDTH

Defines the number of bits per data transfer chosen in the Configure dialog.

**PRELIMINARY**



## Sample Firmware Source Code

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

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

```
#include <device.h>

void main()
{
    uint8 i = 0;
    uint8 val = 0;
    SPIM_1_Start();

    while(1)
    {
        SPIM_1_WriteByte(i++);
        val = SPIM_1_ReadByte();
    }
}
```

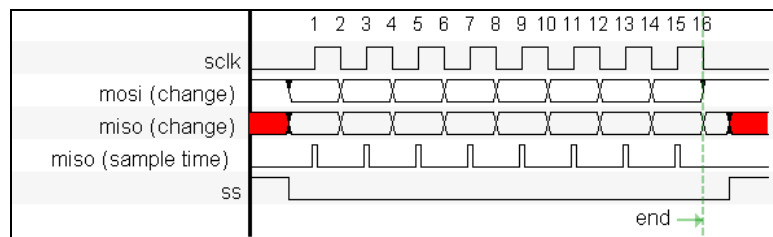
## Functional Description

### Default Configuration

The default configuration for the SPIM is as an 8-bit SPIM with Mode 00 configuration. By Default the Internal clock is selected with a bit-rate of 12Mb/s.

### SPIM Mode: 00

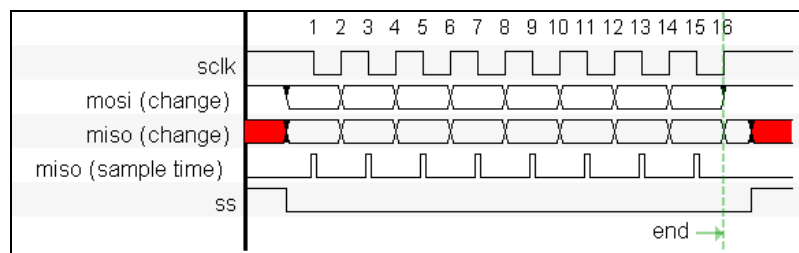
Mode 00 defines the Clock Phase of 0 and the Clock Polarity of 0 which has the following characteristics:



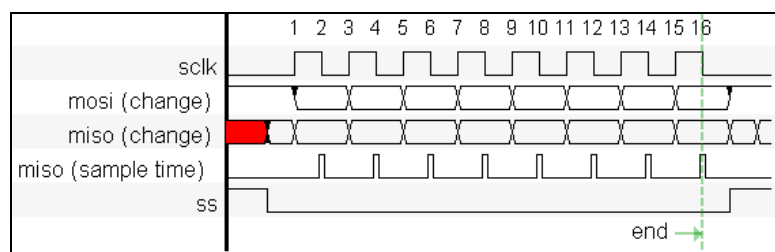
PRELIMINARY

**SPIM Mode: 01**

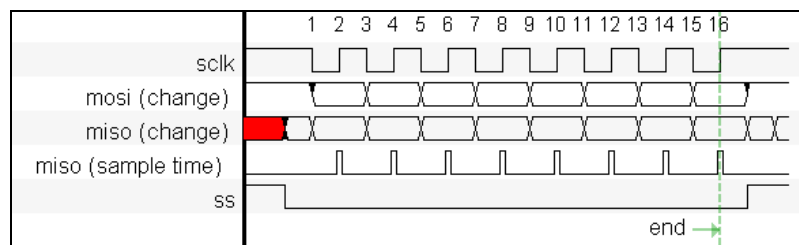
Mode 01 defines the Clock Phase of 0 and the Clock Polarity of 1 which has the following characteristics:

**SPIM Mode: 10**

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

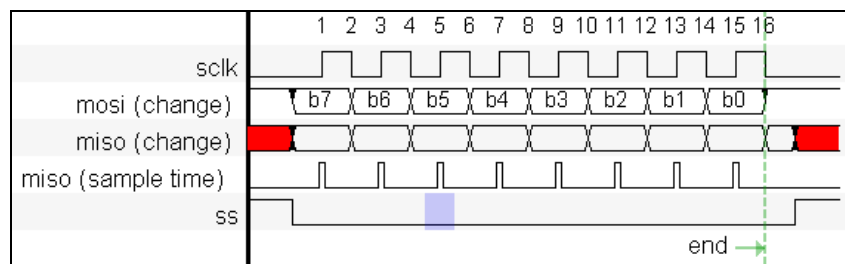
**SPIM Mode: 11**

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

**PRELIMINARY**

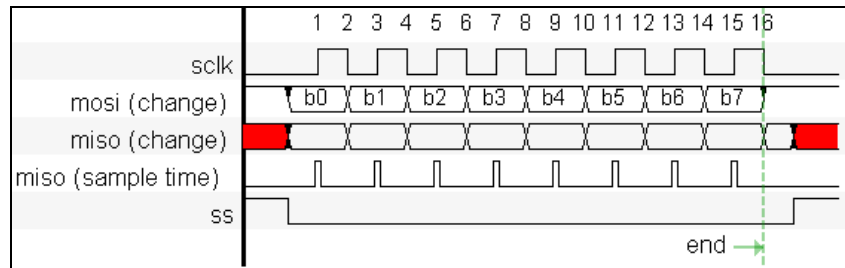
## SPIM ShiftDir: MSB\_First

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:



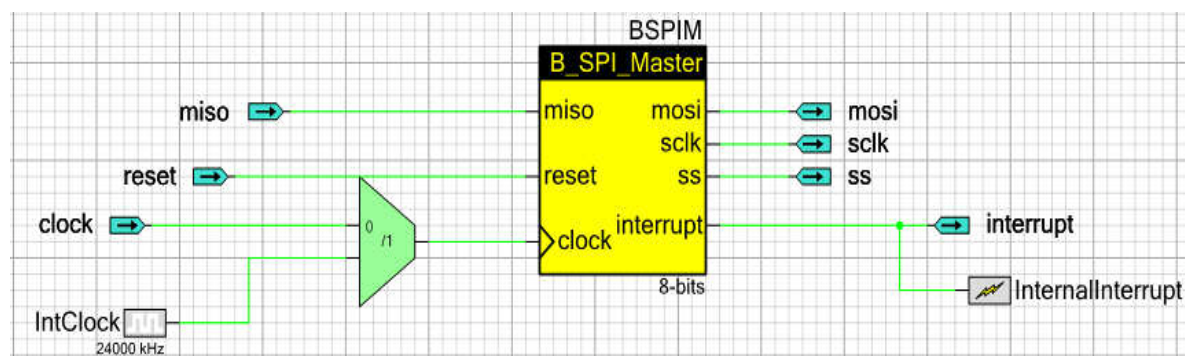
## SPIM ShiftDir: LSB\_First

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:



## Block Diagram and Configuration

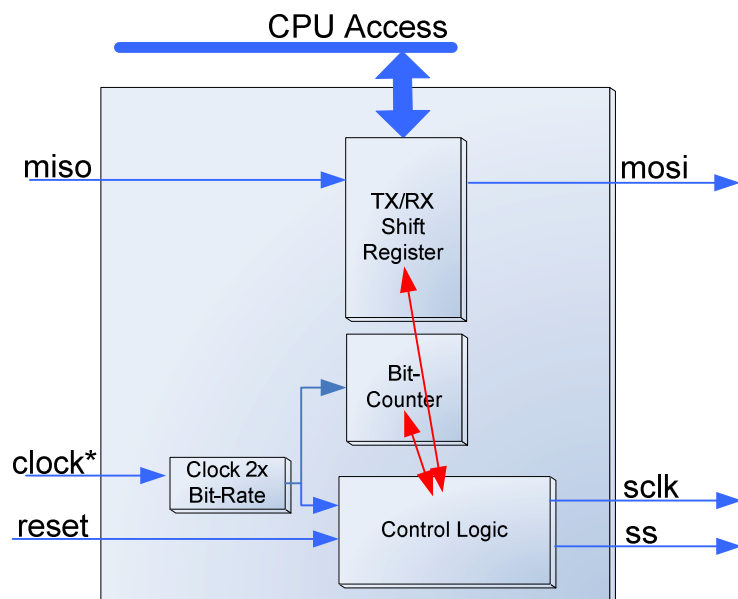
The SPIM is only available as a UDB configuration of blocks. The API is described above and the registers are described here to define the overall implementation of the SPIM.





The implementation is described in the following block diagram.

**Figure 3 UDB Implementation**



## Registers

### Status

The status register is a read only register which contains the various status bits defined for the SPIM. The value of this register is available with the `SPIM_ReadStatus()` function call. The interrupt output signal is generated from an OR'ing of the masked bit-fields within the status register. You can set the mask using the `SPIM_SetInterruptMode()` function call and upon receiving an interrupt you can retrieve the interrupt source by reading the Status register with the `SPIM_ReadStatus()` function call. The Status register is clear on read so the interrupt source is held until the `SPIM_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:

#### **SPIM\_STS\_SPI\_DONE \***

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

**PRELIMINARY**



**SPIM\_STS\_TX\_FIFO\_EMPTY**

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

**SPIM\_STS\_TX\_FIFO\_NOT\_FULL**

Defined as the bit-mask of the Status register bit “Transmit FIFO Not Full”.

**SPIM\_STS\_RX\_FIFO\_FULL**

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

**SPIM\_STS\_RX\_FIFO\_NOT\_EMPTY**

Defined as the bit-mask of the Status register bit “Receive FIFO Not Empty”.

**SPIM\_STS\_RX\_FIFO\_OVERRUN \***

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

**SPIM\_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 SPIM. 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.

**RX Data**

The RX data register contains the received data. This is implemented as a FIFO in the SPIM. 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.



**PRELIMINARY**

## Conditional Compilation Information

The SPIM 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.

### SPIM\_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	

© Cypress Semiconductor Corporation, 2010. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

PSoC® is a registered trademark, and PSoC Creator™ and Programmable System-on-Chip™ are trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are property of the respective corporations.

Any Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and/or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.

**PRELIMINARY**

