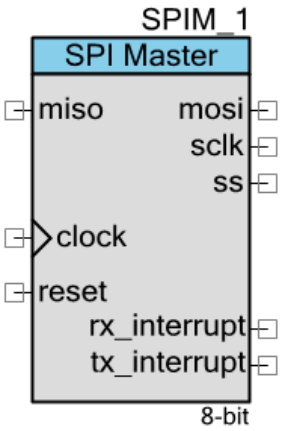
Serial Peripheral Interface (SPI) Master

2.30



Features

- 3- to 16-bit data width
- Four SPI operating modes
- Bit rate up to 18 Mbps*

General Description

The SPI Master component provides an industry-standard, 4-wire master SPI interface. It can also provide a 3-wire (bidirectional) SPI interface. Both interfaces support all four SPI operating modes, allowing communication with any SPI slave device. In addition to the standard 8-bit word length, the SPI Master supports a configurable 3- to 16-bit word length for communicating with nonstandard SPI word lengths.

SPI signals include the standard Serial Clock (SCLK), Master In Slave Out (MISO), Master Out Slave In (MOSI), bidirectional Serial Data (SDAT), and Slave Select (SS).

When to Use the SPI Master

You can use the SPI Master component any time the PSoC device must 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 serial interface.

You should use the SPI Slave component in instances in which the PSoC device must communicate with an SPI master device. You should use the Shift Register component in situations where its low-level flexibility provides hardware capabilities not available in the SPI Master component.

---

* This value is valid only for the case when High Speed Mode Emnable option is set (see DC and AC Electrical Characteristics for details) . Otherwise maximum bit rate value is up to 9 mbps.

Input/Output Connections

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

miso – Input *

The miso input carries the Master In Slave Out (MISO) signal from a slave device. This input is visible when the **Data Lines** parameter is set to **MOSI + MISO**. If visible, this input must be connected.

sdat – Inout *

The sdat inout carries the Serial Data (SDAT) signal. This input is used when the **Data Lines** parameter is set to **Bi-Directional**.

Figure 1. SPI Bidirectional Mode (data transmission from Master to Slave)

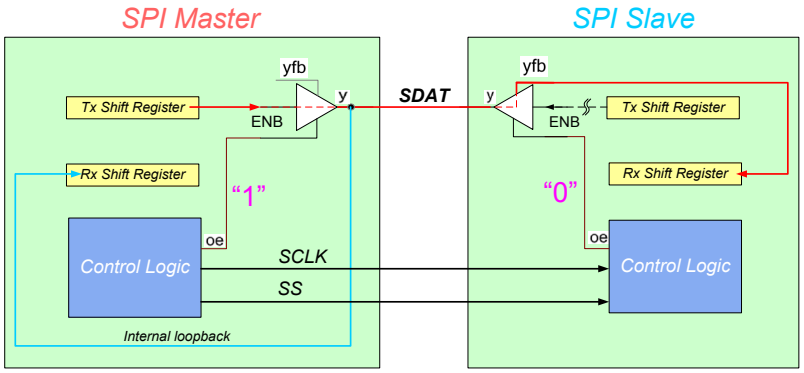
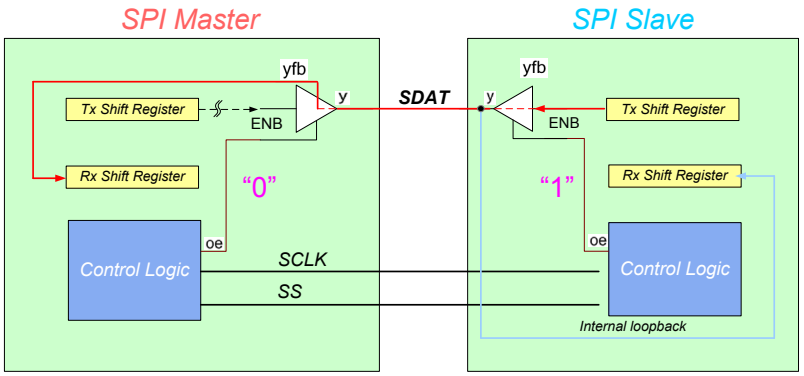


Figure 2. SPI Bidirectional Mode (data transmission from Slave to Master)



Initial component's state in Bi-directional Mode is Rx mode or Data transmission from Slave to Master as it is shown on Figure 2. SPIM_TxEnable() and SPIM_Tx_Disable() API functions should be used to switch between Rx and Tx mode.

clock – Input *

The clock input defines the bit rate of the serial communication. The bit rate is one-half the input clock frequency.

The clock input is visible when the **Clock Selection** parameter is set to **External Clock**. If visible, this input must be connected. If you select **Internal Clock**, then you must define the desired data bit rate; the required clock is solved and provided by PSoC Creator.

reset – Input

Resets the SPI state machine to the idle state. This throws out any data that was currently being transmitted or received but does not clear data from the FIFO that has already been received or is ready to be transmitted. Note that PSoC 5 silicon does not support routed reset functionality, so this input should be unconnected when the component is used in PSoC 5 silicon projects.

mosi – Output *

The mosi output carries the Master Out Slave In (MOSI) signal from the master device on the bus. This output is visible when the **Data Lines** parameter is set to **MOSI + MISO**.

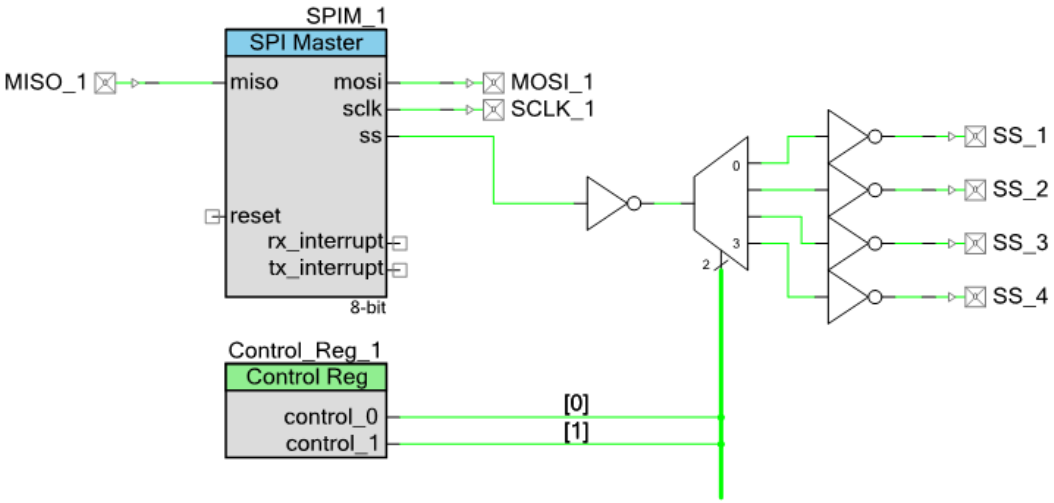
sclk– Output

The sclk output carries the Serial Clock (SCLK) signal. It provides the master synchronization clock output to the slave devices on the bus.

ss – Output

The ss output is hardware controlled. It carries the Slave Select (SS) signal to the slave devices on the bus. You can connect a digital demultiplexer to handle multiple slave devices, or to have a completely firmware-controlled SS. See Figure 3 and Figure 4 for examples.

Figure 3. Slave Select Output to Demultiplexer



Unless inverters are put at the output of the demux, a logic ‘0’ can’t be transferred through it, because the other unselected channels will also have logic ‘0’. To compensate for this, an inverter is put at the input of demux.

Figure 4. Firmware Controlled Slave Selects

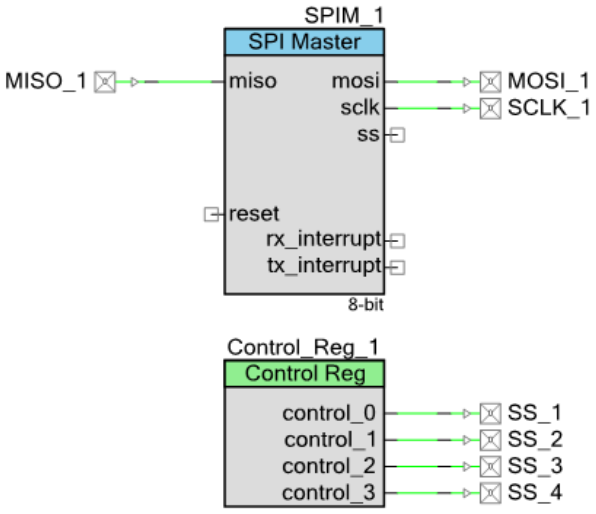
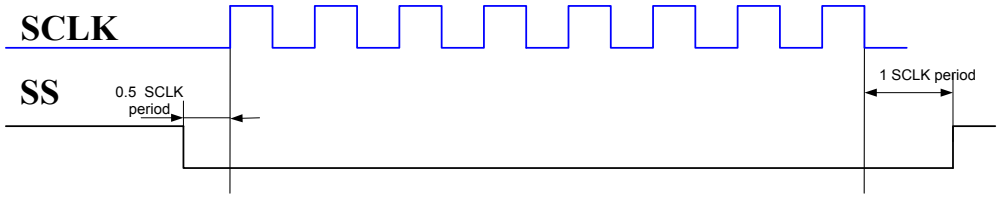


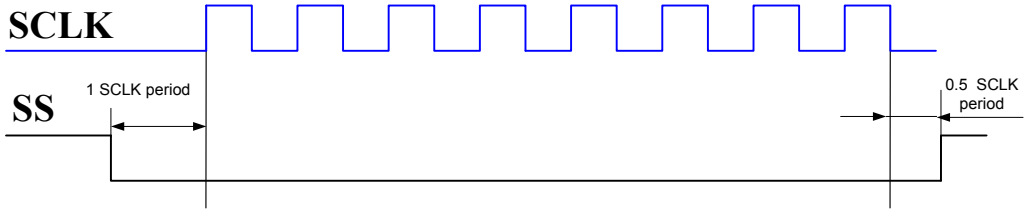
Figure 5 shows the timing correlation between SS and SCLK.

Figure 5. SS and SCLK Timing Correlation

CPHA =0:



CPHA =1:



Note SS is not set to high during a multi-byte/word transmission if the “SPI Done” condition was not generated.

rx_interrupt – Output

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

tx_interrupt – Output

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

Schematic Macro Information

By default, the PSoC Creator Component Catalog contains Schematic Macro implementations for the SPI Master component. These macros contain already connected and adjusted input and output pins and clock source. Schematic Macros are available both for 4-wire (Full Duplex) and 3-wire (Bidirectional) SPI interfacing.

Figure 6. 4-Wire (Full Duplex) Interfacing Schematic Macro

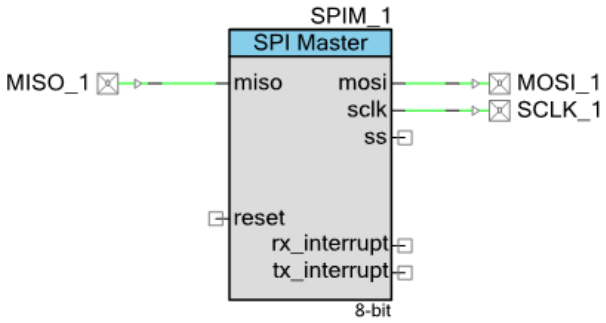
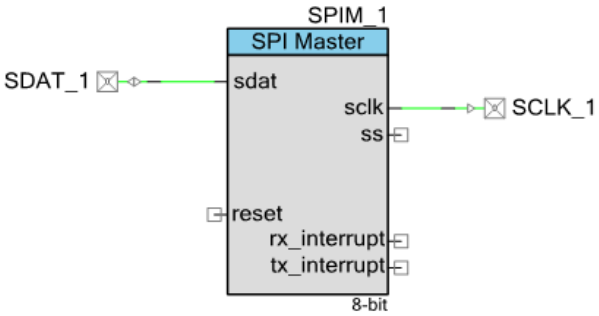


Figure 7. 3-Wire (Bidirectional) Interfacing Schematic Macro



Note If you do not use a Schematic Macro, configure the Pins component to deselect the Input Synchronized parameter for each of the assigned input pins (MISO or SDAT inout). The parameter is located under the **Pins > Input** tab of the applicable Pins Configure dialog.

Component Parameters

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

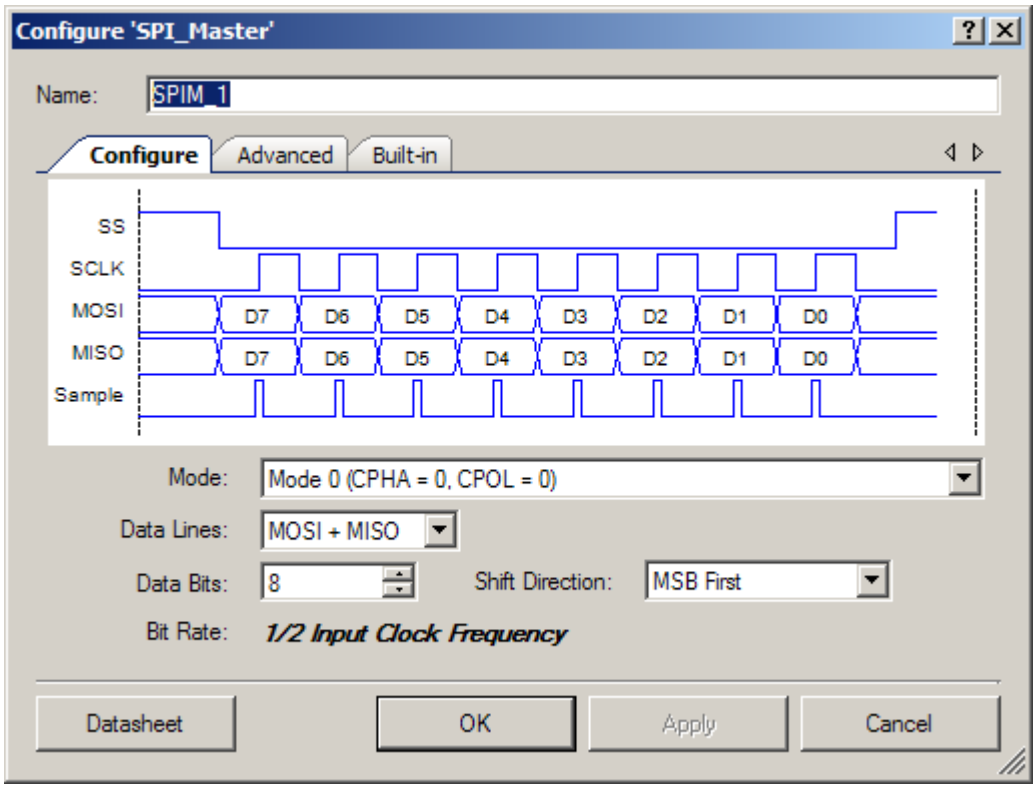
The following sections describe the SPI Master parameters, and how they are configured using the Configure dialog. They also indicate whether the options are implemented in hardware or software.

Hardware versus 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 can be modified at any time with the provided APIs. Hardware-only parameters are marked with an asterisk (*).

Configure Tab

The **Configure** tab contains basic parameters required for every SPI component. These parameters are the first ones that appear when you open the **Configure** dialog.



Note The sample signal in the waveform is not an input or output of the system; it only indicates when the data is sampled at the master and slave for the mode settings selected.

Mode *

The **Mode** parameter defines the clock phase and clock polarity mode you want to use in the communication. These modes are defined in the following table. See Functional Description for more information.

CPHA	CPOL
0	0
0	1
1	0
1	1

Data Lines

The **Data Lines** parameter defines which interface is used for SPI communication – 4-wire (MOSI + MISO) or 3-wire (Bidirectional).

Data Bits *

The number of **Data Bits** defines the bit width of a single transfer as transferred with the SPIM_ReadRxData() and SPIM_WriteTxData() functions. The default number of bits is a single byte (8 bits). Any integer from 3 to 16 is a valid setting.

Shift Direction *

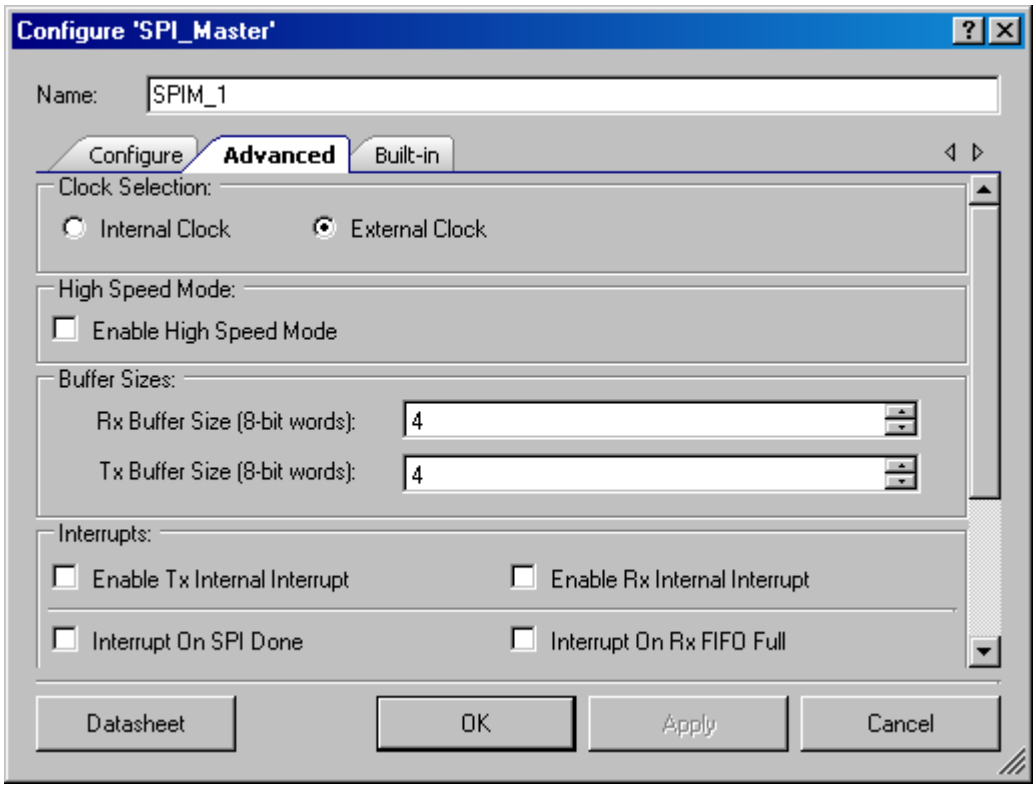
The **Shift Direction** parameter defines the direction in which the serial data is transmitted. When set to **MSB First**, the most-significant bit is transmitted first. This is implemented by shifting the data left. When set to **LSB First**, the least-significant bit is transmitted first. This is implemented by shifting the data right.

Bit Rate *

If the **Clock Selection** parameter (on the **Advanced** tab) is set to **Internal Clock**, the **Bit Rate** parameter defines the SCLK speed in Hertz. The clock frequency of the internal clock will be 2x the SCLK rate. This parameter has no affect if the **Clock Selection** parameter is set to **External Clock**.

Advanced Tab

The **Advanced** tab contains parameters that provide additional functionality.



Clock Selection *

The **Clock Selection** parameter allows you to choose between an internally configured clock or an externally configured clock for data rate and SCLK 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 will display the expected bit rate based on the user-connected clock source. If this parameter is set to **Internal Clock**, the clock input is not visible on the symbol.

Note When setting the bit rate or external clock frequency value, make sure that PSoC Creator can provide this value using the current system clock frequency. Otherwise, a warning about the clock accuracy range is generated while building the project. This warning contains the actual clock value set by PSoC Creator. Choose whether the system clock or component clock should be changed to fit the clocking system requirements and achieve an optimal value.

RX Buffer Size *

The **RX Buffer Size** parameter defines the size (in bytes/words) of memory allocated for a circular data buffer. If this parameter is set to 1 to 4, the fourth byte/word of the FIFO is implemented in the hardware. Values 1 to 3 are available only for compatibility with the previous versions; using them causes an error icon to display that the value is incorrect. All other values

up to 255 bytes/words use the 4-byte/word FIFO and a memory array controlled by the supplied API.

TX Buffer Size *

The **TX Buffer Size** parameter defines the size (in bytes/words) of memory allocated for a circular data buffer. If this parameter is set to 1 to 4, the fourth byte/word of the FIFO is implemented in the hardware. Values 1 to 3 are available only for compatibility with the previous versions; using them causes an error icon to display that the value is incorrect. All other values up to 255 use the 4-byte/word FIFO and a memory array controlled by the supplied API.

Using the Software Buffer

Selecting Rx/Tx Buffer Size values greater than 4 allows you to use the Rx/Tx circular software buffers. The internal interrupt handler is used when you select the Tx/Rx software buffer option. Its main purpose is to provide interaction between software and hardware Tx/Rx buffers. In the initial state, the BufferRead and BufferWrite pointers point to the zero element of the software buffer. After writing the first data, the BufferWrite pointer moves to the first element of the software buffer and points to writing data; the BufferRead pointer stays on the zero element. As the buffers work, the pointers move to the next buffer elements. The BufferWrite pointer points to the last written data. The BufferRead pointer points to the oldest data that has not been read. Software buffer overflow can happen without any overflow indication. You must handle any software buffer overflow situation.

You should also consider that using the software buffer leads to greater timing intervals between transmitted words because of the extra time the interrupt handler needs to execute (depending on the selected bus clock value). When setting timing intervals between transmitted words, use DMA along with a hardware buffer.

Enable High Speed Mode

The Enable High Speed Mode parameter allows enlarging of the maximum bit rate value up to 18 mbps (see DC and AC electrical characteristics section for details).

Enable TX / RX Internal Interrupt

The **Enable TX / RX Internal Interrupt** options allow you to use the predefined Tx and Rx ISRs of the SPI Master component, or supply your own custom ISRs. If enabled, you may add your own code to these predefined ISRs if small changes are required. If the internal interrupt is deselected, you may supply an external interrupt component with custom code connected to the interrupt outputs of the SPI Master.

If the Rx or Tx buffer size is greater than 4, the component automatically sets the appropriate parameters, as the internal ISR is needed to handle transferring data from the hardware FIFO to the Rx buffer, the Tx buffer, or both. The interrupt output pins of the SPI master are always visible and usable, outputting the same signal that goes to the internal interrupt. This output can

then be used as a DMA request source or as a digital signal to be used as required in the programmable digital system.

Notes

- When Rx buffer size is greater than 4 bytes/words, the ‘RX FIFO NOT EMPTY’ interrupt is always enabled and cannot be disabled, because it causes incorrect buffer functionality.
- When Tx buffer size is greater than 4 bytes/words, the ‘TX FIFO NOT FULL’ interrupt is always enabled and cannot be disabled, because it causes incorrect buffer functionality.
- For buffer sizes greater than 4 bytes/words, the SPI slave and global interrupt must be enabled for proper buffer handling.

Interrupts

The **Interrupts** selection parameters allow you to configure the internal events that are enabled to cause an interrupt. Interrupt generation is a masked OR of all of the enabled Tx and Rx status register bits. The bits chosen with these parameters define the mask implemented with the initial component configuration.

Clock Selection

When the internal clock configuration is selected, PSoC Creator calculates the needed frequency and clock source, and generates the clocking resource needed for implementation. Otherwise, you must supply the clock component and calculate the required clock frequency. That frequency is 2x the desired bit rate and SCLK frequency.

Note When setting the bit rate or external clock frequency value, make sure that PSoC Creator can provide this value by using the current system clock frequency. Otherwise, a warning about the clock accuracy range is generated while building the project. This warning contains the real clock value set by PSoC Creator. Choose whether the system clock or component clock should be changed to fit the clocking system requirements and achieve an optimal value.

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software at runtime. 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 the instance. For readability, the instance name used in the following table is “SPIM.”

Function	Description
SPIM_Start()	Calls both SPIM_Init() and SPIM_Enable(). Should be called the first time the component is started.
SPIM_Stop()	Disables SPI Master operation.
SPIM_EnableTxInt()	Enables the internal Tx interrupt irq.
SPIM_EnableRxInt()	Enables the internal Rx interrupt irq.
SPIM_DisableTxInt()	Disables the internal Tx interrupt irq.
SPIM_DisableRxInt()	Disables the internal Rx interrupt irq.
SPIM_SetTxInterruptMode()	Configures the Tx interrupt sources enabled.
SPIM_SetRxInterruptMode()	Configures the Rx interrupt sources enabled.
SPIM_ReadTxStatus()	Returns the current state of the Tx status register.
SPIM_ReadRxStatus()	Returns the current state of the Rx status register.
SPIM_WriteTxData()	Places a byte/word in the transmit buffer which will be sent at the next available bus time.
SPIM_ReadRxData()	Returns the next byte/word of received data available in the receive buffer.
SPIM_GetRxBufferSize()	Returns the size (in bytes/words) of received data in the Rx memory buffer.
SPIM_GetTxBufferSize()	Returns the size (in bytes/words) of data waiting to transmit in the Tx memory buffer.
SPIM_ClearRxBuffer()	Clears the Rx buffer memory array and Rx FIFO of all received data.
SPIM_ClearTxBuffer()	Clears the Tx buffer memory array or Tx FIFO of all transmit data. **Note** Tx FIFO will be cleared only if software buffer is not used.
SPIM_TxEnable()	If configured for bidirectional mode, sets the SDAT inout to transmit.
SPIM_TxDisable()	If configured for bidirectional mode, sets the SDAT inout to receive.
SPIM_PutArray()	Places an array of data into the transmit buffer.
SPIM_ClearFIFO()	Clears any received data from the Rx hardware FIFO.

Function	Description
SPIM_Sleep()	Prepares SPI Master component for low-power modes by calling SPIM_SaveConfig() and SPIM_Stop() functions.
SPIM_Wakeup()	Restores and re-enables the SPI Master component after waking from low-power mode.
SPIM_Init()	Initializes and restores the default SPI Master configuration.
SPIM_Enable()	Enables the SPI Master to start operation.
SPIM_SaveConfig()	Saves SPI Master hardware configuration.
SPIM_RestoreConfig()	Restores SPI Master hardware configuration.

Global Variables

Variable	Description
SPIM_initVar	Indicates whether the SPI Master has been initialized. The variable is initialized to 0 and set to 1 the first time SPIM_Start() is called. This allows the component to restart without reinitialization after the first call to the SPIM_Start() routine. If reinitialization of the component is required, then the SPIM_Init() function can be called before the SPIM_Start() or SPIM_Enable() function.
SPIM_txBufferWrite	Transmits buffer location of the last data written into the buffer by the API.
SPIM_txBufferRead	Transmits buffer location of the last data read from the buffer and transmitted by SPI Master hardware.
SPIM_rxBufferWrite	Receives buffer location of the last data written into the buffer after received by SPI Master hardware.
SPIM_rxBufferRead	Receives buffer location of the last data read from the buffer by the API.
SPIM_rxBufferFull	Indicates the software buffer has overflowed.
SPIM_RXBUFFER[]	Used to store received data.
SPIM_TXBUFFER[]	Used to store data for sending.

void SPIM_Start(void)

Description: This function calls both SPIM_Init() and SPIM_Enable(). This should be called the first time the component is started.

Parameters: None

Return Value: None

Side Effects: None

void SPIM_Stop(void)

Description: Disables SPI Master operation by disabling the internal clock and internal interrupts, if the SPI Master is configured that way.

Parameters: None

Return Value: None

Side Effects: None

void SPIM_EnableTxInt (void)

Description: Enables the internal Tx interrupt irq.

Parameters: None

Return Value: None

Side Effects: None

void SPIM_EnableRxInt (void)

Description: Enables the internal Rx interrupt irq.

Parameters: None

Return Value: None

Side Effects: None

void SPIM_DisableTxInt (void)

Description: Disables the internal Tx interrupt irq.

Parameters: None

Return Value: None

Side Effects: None

void SPIM_DisableRxInt (void)

Description: Disables the internal Rx interrupt irq.

Parameters: None

Return Value: None

Side Effects: None

void SPIM_SetTxInterruptMode (uint8 intSrc)

Description: Configures which status bits trigger an interrupt event.

Parameters: uint8 intSrc: Bit field containing the interrupts to enable.

Bit	Description
SPIM_INT_ON_SPI_DONE	Enable interrupt due to SPI done
SPIM_INT_ON_TX_EMPTY	Enable interrupt due to Tx FIFO empty
SPIM_INT_ON_TX_NOT_FULL	Enable interrupt due to Tx FIFO not full
SPIM_INT_ON_BYTE_COMP	Enable interrupt due to byte/word complete
SPIM_INT_ON_SPI_IDLE	Enable interrupt due to SPI IDLE

Based on the bit-field arrangement of the Tx status register. This value must be a combination of Tx status register bit masks defined in the header file.

For more information, see Defines.

Return Value: None

Side Effects: None

void SPIM_SetRxInterruptMode (uint8 intSrc)

Description: Configures which status bits trigger an interrupt event.

Parameters: uint8 intSrc: Bit field containing the interrupts to enable.

Bit	Description
SPIM_INT_ON_RX_FULL	Enable interrupt due to Rx FIFO Full
SPIM_INT_ON_RX_NOT_EMPTY	Enable interrupt due to Rx FIFO Not Empty
SPIM_INT_ON_RX_OVER	Enable interrupt due to Rx Buf Overrun

Based on the bit-field arrangement of the Rx status register. This value must be a combination of Rx status register bit masks defined in the header file.

For more information, see Defines.

Return Value: None

Side Effects: None

uint8 SPIM_ReadTxStatus (void)

Description: Returns the current state of the Tx status register. For more information, see Status Register Bits.

Parameters: None

Return Value: uint8: Current Tx status register value

Bit	Description
SPIM_STS_SPI_DONE	SPI done
SPIM_STS_TX_FIFO_EMPTY	Tx FIFO empty
SPIM_STS_TX_FIFO_NOT_FULL	Tx FIFO not full
SPIM_STS_BYTE_COMPLETE	Byte/Word complete
SPIM_STS_SPI_IDLE	SPI IDLE

Side Effects: Tx Status register bits are cleared on read.

uint8 SPIM_ReadRxStatus (void)

Description: Returns the current state of the Rx status register. For more information, see Status Register Bits.

Parameters: None

Return Value: uint8: Current Rx status register value

Bit	Description
SPIM_STS_RX_FIFO_FULL	Rx FIFO Full
SPIM_STS_RX_FIFO_NOT_EMPTY	Rx FIFO Not Empty
SPIM_STS_RX_FIFO_OVERRUN	Rx Buf Overrun

Side Effects: Rx Status register bits are cleared on read.

void SPIM_WriteTxData (uint8/uint16 txData)

Description: Places a byte/word in the transmit buffer to be sent at the next available SPI bus time.

Parameters: uint8/uint16 txData: The data value to transmit from the SPI.

Return Value: None

Side Effects: Data may be placed in the memory buffer and will not be transmitted until all other previous data has been transmitted. This function blocks until there is space in the output memory buffer.

Clears the Tx status register of the component.

uint8/uint16 SPIM_ReadRxData (void)

Description: Returns the next byte/word of received data available in the receive buffer.

Parameters: None

Return Value: uint8/uint16: The next byte/word of data read from the FIFO.

Side Effects: Returns invalid data if the FIFO is empty. Call SPIM_GetRxBufferSize(), and if it returns a nonzero value then it is safe to call the SPIM_ReadRxData() function.

uint8 SPIM_GetRxBufferSize (void)

Description: Returns the number of bytes/words of received data currently held in the Rx buffer.

- If the Rx software buffer is disabled, this function returns 0 = FIFO empty or 1 = FIFO not empty.
- If the Rx software buffer is enabled, this function returns the size of data in the Rx software buffer. FIFO data not included in this count.

Parameters: None

Return Value: uint8: Integer count of the number of bytes/words in the Rx buffer.

Side Effects: Clears the Rx status register of the component.

uint8 SPIM_GetTxBufferSize (void)

Description: Returns the number of bytes/words of data ready to transmit currently held in the Tx buffer.

- If Tx software buffer is disabled, this function returns 0 = FIFO empty, 1 = FIFO not full, or 4 = FIFO full.
- If the Tx software buffer is enabled, this function returns the size of data in the Tx software buffer. FIFO data not included in this count.

Parameters: None

Return Value: uint8: Integer count of the number of bytes/words in the Tx buffer

Side Effects: Clears the Tx status register of the component.

void SPIM_ClearRxBuffer (void)

Description: Clears the Rx buffer memory array and Rx hardware FIFO of all received data. Clears the Rx RAM buffer by setting both the read and write pointers to zero. Setting the pointers to zero indicates that there is no data to read. Thus, writing resumes at address 0, overwriting any data that may have remained in the RAM.

Parameters: None

Return Value: None

Side Effects: Any received data not read from the RAM buffer and FIFO is lost when overwritten by new data.

void SPIM_ClearTxBuffer (void)

Description: Clears the Tx buffer memory array of data waiting to transmit. Clears the Tx RAM buffer by setting both the read and write pointers to zero. Setting the pointers to zero indicates that there is no data to transmit. Thus, writing resumes at address 0, overwriting any data that may have remained in the RAM.

Parameters: None

Return Value: None

Side Effects: If the software buffer is used, it does not clear data already placed in the Tx FIFO. Any data not yet transmitted from the RAM buffer is lost when overwritten by new data.

void SPIM_TxEnable (void)

Description: If the SPI Master is configured to use a single bidirectional pin, this sets the bidirectional pin to transmit.

Parameters: None

Return Value: None

Side Effects: None

void SPIM_TxDisable (void)

Description: If the SPI master is configured to use a single bidirectional pin, this sets the bidirectional pin to receive.

Parameters: None

Return Value: None

Side Effects: None

void SPIM_PutArray (uint8/uint16 * buffer, uint8/uint16 byteCount)

Description: Places an array of data into the transmit buffer

Parameters: uint8 * buffer: Pointer to the location in RAM containing the data to send

uint8/uint16 byteCount: The number of bytes/words to move to the transmit buffer.

Return Value: None

Side Effects: The system will stay in this function until all data has been transmitted to the buffer. This function is blocking if there is not enough room in the Tx buffer. It may get locked in this loop if data is not being transmitted by the master and the Tx buffer is full.

void SPIM_ClearFIFO (void)

Description: Clears any received data from the Tx and Rx FIFOs.

Parameters: None

Return Value: None

Side Effects: Clears status register of the component.

void SPIM_Sleep (void)

Description: Prepares SPI Master for low-power modes by calling SPIM_SaveConfig() and SPIM_Stop() functions.

Parameters: None

Return Value: None

Side Effects: None

void SPIM_Wakeup (void)

Description: Prepares SPI Master to wake up from a low-power mode. Calls SPIM_RestoreConfig() and SPIM_Enable() functions. Clears all data from Rx buffer, Tx buffer, and hardware FIFOs.

Parameters: None

Return Value: None

Side Effects: None

void SPIM_Init(void)

Description: Initializes or restores the component according to the customizer **Configure** dialog settings. It is not necessary to call SPIM_Init() because the SPIM _Start() routine calls this function and is the preferred method to begin component operation.

Parameters: None

Return Value: None

Side Effects: When this function is called, it initializes all of the necessary parameters for execution. These include setting the initial interrupt mask, configuring the interrupt service routine, configuring the bit-counter parameters, and clearing the FIFO and Status Register.

void SPIM_Enable(void)

Description: Enables SPI Master for operation. Starts the internal clock if the SPI Master is configured that way. If it is configure for an external clock it must be started separately before calling this function. The SPIM_Enable() function should be called before SPI Master interrupts are enabled. This is because this function configures the interrupt sources and clears any pending interrupts from device configuration, and then enables the internal interrupts if there are any. A SPIM_Init() function must have been previously called.

Parameters: None

Return Value: None

Side Effects: None

void SPIM_SaveConfig (void)

Description: Saves SPI Master hardware configuration before entering a low-power mode.

Parameters: None

Return Value: None

Side Effects: None

void SPIM_RestoreConfig (void)

Description: Restores SPI Master hardware configuration saved by the SPIM_SaveConfig() function after waking from a lower-power mode.

Parameters: None

Return Value: None

Side Effects: If this function is called without first calling SPIM_SaveConfig() then in the following registers will be restored to the default values from the Configure dialog:

```
SPIM_STATUS_MASK_REG
SPIM_COUNTER_PERIOD_REG
```

Defines

- **SPIM_TX_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 Tx status register that have been enabled at configuration as sources for the interrupt. See Status Register Bits for bit-field details.
- **SPIM_RX_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 Rx status register that have been enabled at configuration as sources for the interrupt. See Status Register Bits for bit-field details.

Status Register Bits

SPIM_TXSTATUS

Bits	7	6	5	4	3	2	1	0
Value	Interrupt	Unused	Unused	SPI IDLE	Byte/Word Complete	Tx FIFO Not Full	Tx FIFO. Empty	SPI Done

SPIM_RXSTATUS

Bits	7	6	5	4	3	2	1	0
Value	Interrupt	Rx Buf. Overrun	Rx FIFO Not Empty	Rx FIFO Full	Unused	Unused	Unused	Unused

- Byte/Word Complete: Set when a byte/word of data transmit has completed.
- Rx FIFO Overrun: Set when Rx Data has overrun the 4-byte/word FIFO without being moved to the Rx buffer memory array (if one exists)
- Rx FIFO Not Empty: Set when the Rx Data FIFO is not empty. That is, at least one byte/word is in the Rx FIFO (does not indicate the Rx buffer RAM array conditions).
- Rx FIFO Full: Set when the Rx Data FIFO is full (does not indicate the Rx buffer RAM array conditions).
- Tx FIFO Not Full: Set when the Tx Data FIFO is not full (does not indicate the Tx buffer RAM array conditions).
- Tx FIFO Empty: Set when the Tx Data FIFO is empty (does not indicate the Tx buffer 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/word complete status. (Set when Byte/Word Complete has been set and Tx Data FIFO is empty.)

- SPI IDLE: Set when the SPI Master state machine is in the IDLE State. This is the default state after the component starts. It is also the next state after SPI Done. IDLE is still set until Tx FIFO Not Empty status has been detected.

SPIM_TXBUFFERSIZE

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

SPIM_RXBUFFERSIZE

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

SPIM_DATAWIDTH

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

Sample Firmware Source Code

PSoC Creator provides numerous example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

Refer to the “Find Example Project” topic in the PSoC Creator Help for more information.

Functional Description

Default Configuration

The default configuration for the SPI Master is as an 8-bit SPI Master with (CPHA = 0 CPOL = 0) configuration. By default, the Internal clock is selected with a bit rate of 1 Mbps.

Modes

The following four waveforms show the component’s status bits and the signal values that they assume during data transmission. They are based on the assumption that five data bytes are transmitted (four bytes are written to the SPI Master’s Tx buffer at the beginning of transmission and the fifth is thrown after the first byte has been loaded into the A0 register). The numbers in circles on the waveforms represent the following events:

1 – Tx FIFO Empty is cleared when four bytes are written to the Tx buffer.

2 – Tx FIFO Not Full is cleared because Tx FIFO is full after four bytes are written.

3 – SPI IDLE state bit is cleared because of bytes detected in the Tx buffer.

4 – Tx FIFO Not Full status is set when the first byte has been loaded into the A0 register and cleared after the fifth byte has been written to the empty place in the Tx buffer.

5 – Slave Select line is set low, indicating beginning of the transmission.

6 – Tx FIFO Not Full status is set when the second bit is loaded to the A0. Rx Not Empty status is set when the first received byte is loaded into the Rx buffer. Byte/Word Complete is also set.

7 – Tx FIFO Empty status is set at the moment that the last byte to be sent is loaded into the A0 register (to simplify, this is not shown in detail).

8 – At the moment the fourth byte is received, Rx FIFO Full is set along with Byte/Word Complete.

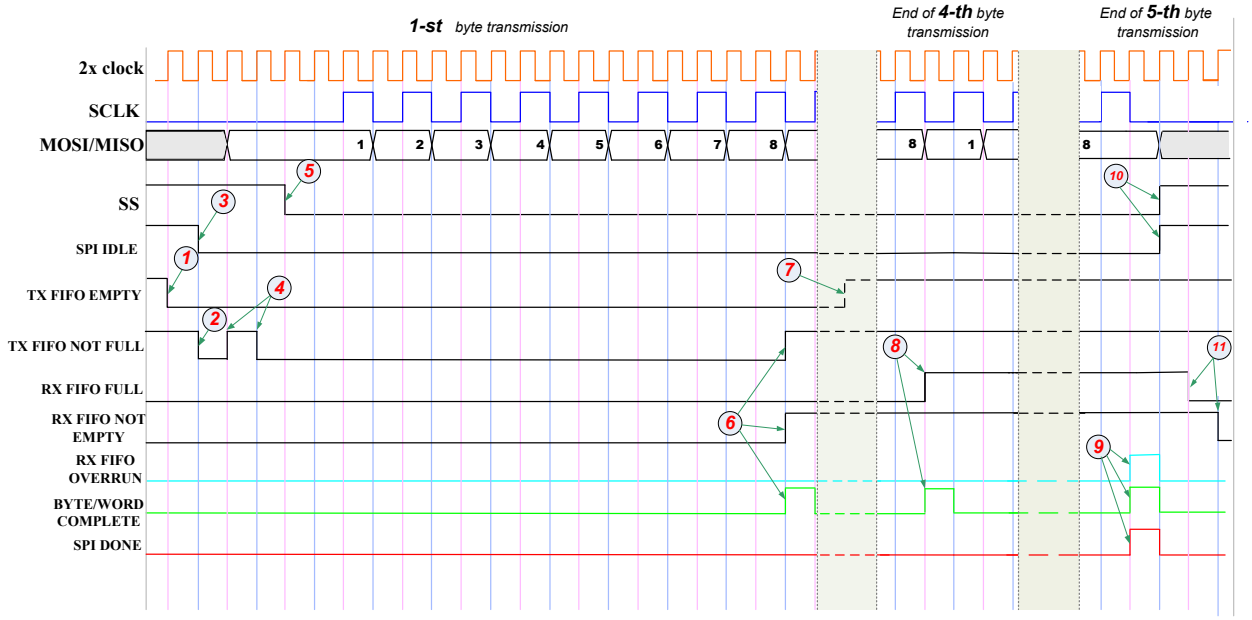
9 – Byte/Word Complete, SPI Done, and Rx Overrun are set because all bytes have been transmitted and an attempt to load data into the full Rx buffer has been detected.

10 – SS line is set high to indicate that transmission is complete. SPI IDLE state is also set.

11 – Rx FIFO Full is cleared when the first byte has been read from the Rx buffer and Rx FIFO Empty is set when all of them have been read.

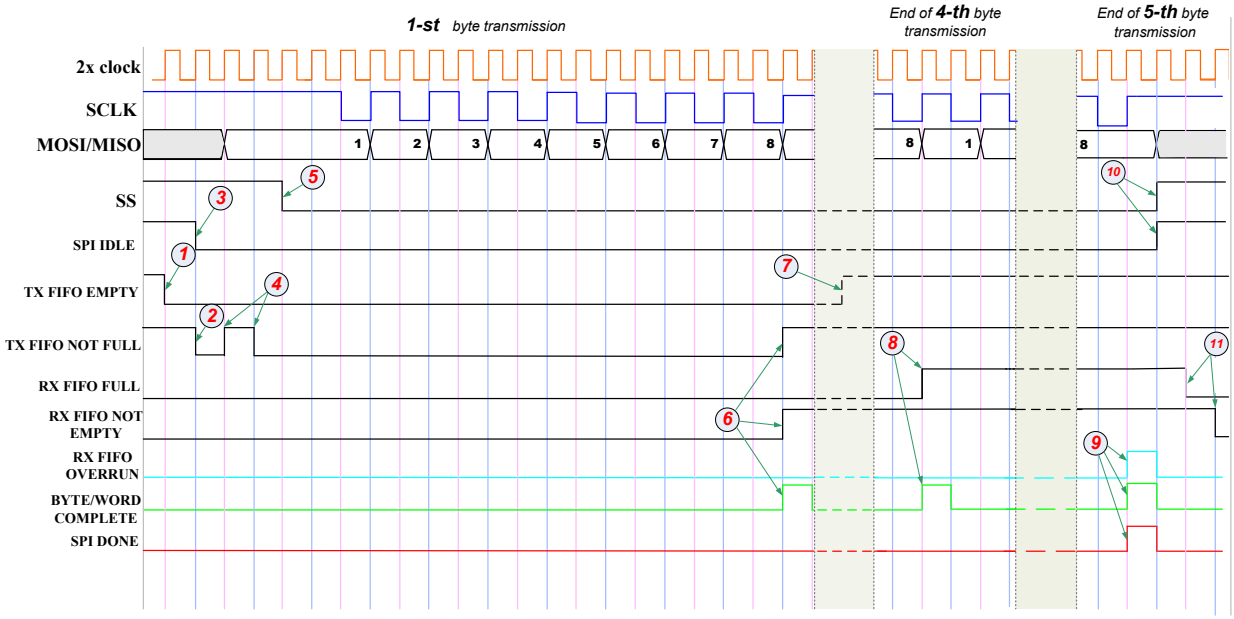
SPI Master Mode:(CPHA == 0, CPOL == 0)

This mode has the following characteristics:



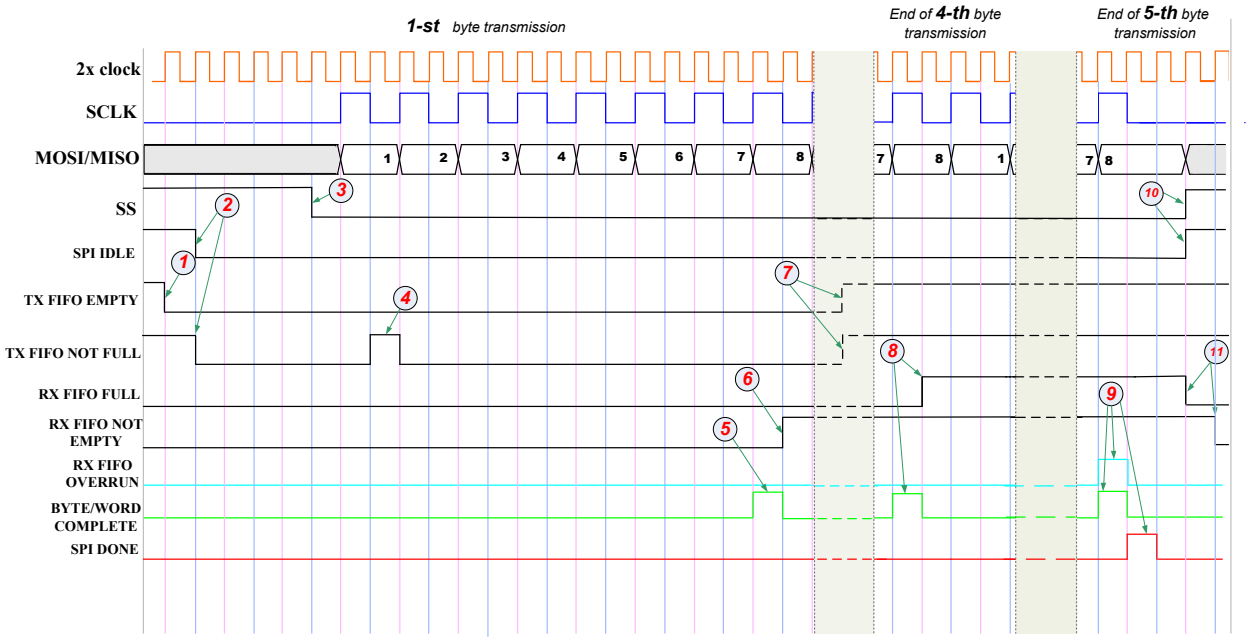
SPI Master Mode: (CPHA == 0, CPOL == 1)

This mode has the following characteristics:



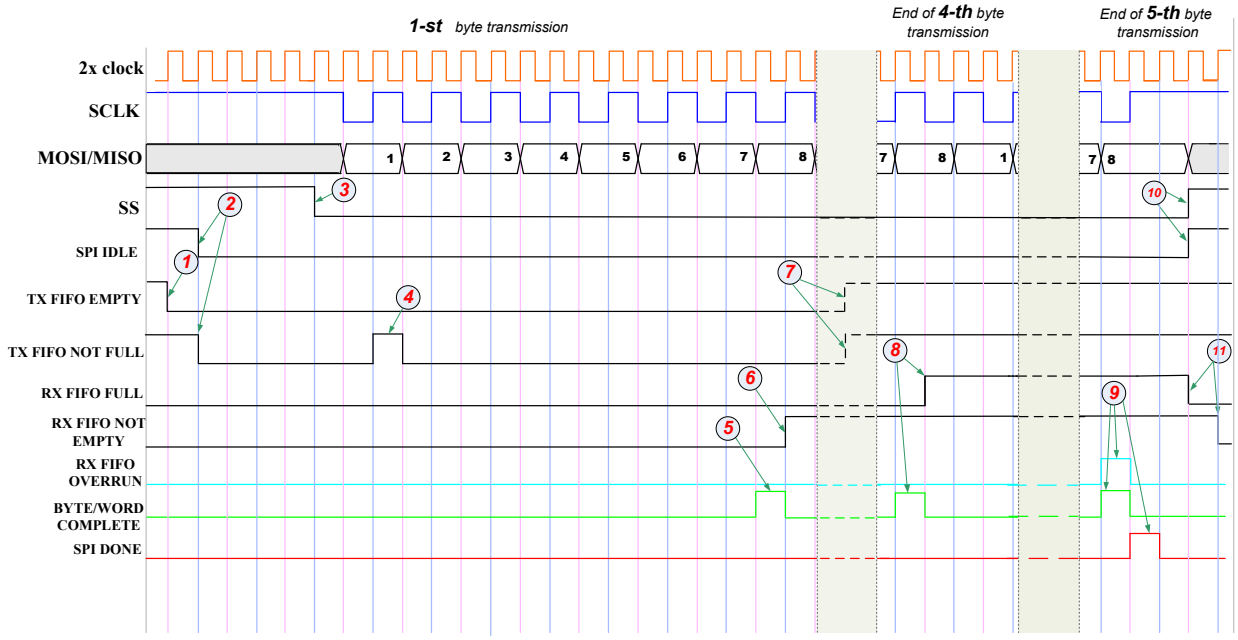
SPI Master Mode: (CPHA == 1, CPOL == 0)

This mode has the following characteristics:



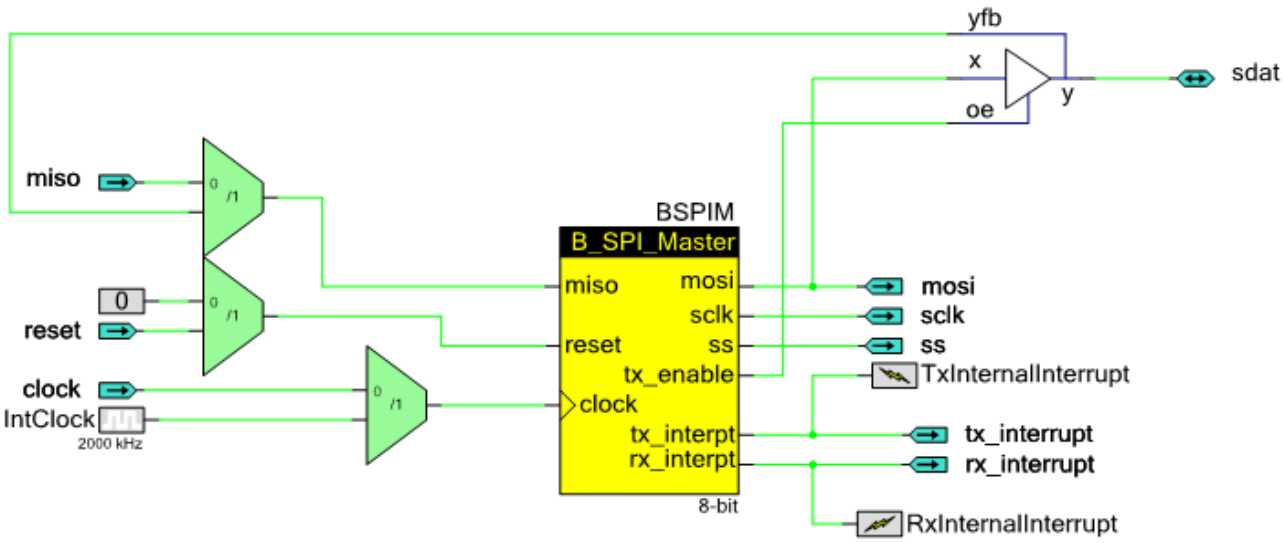
SPI Master Mode:(CPHA == 1, CPOL == 1)

This mode has the following characteristics:

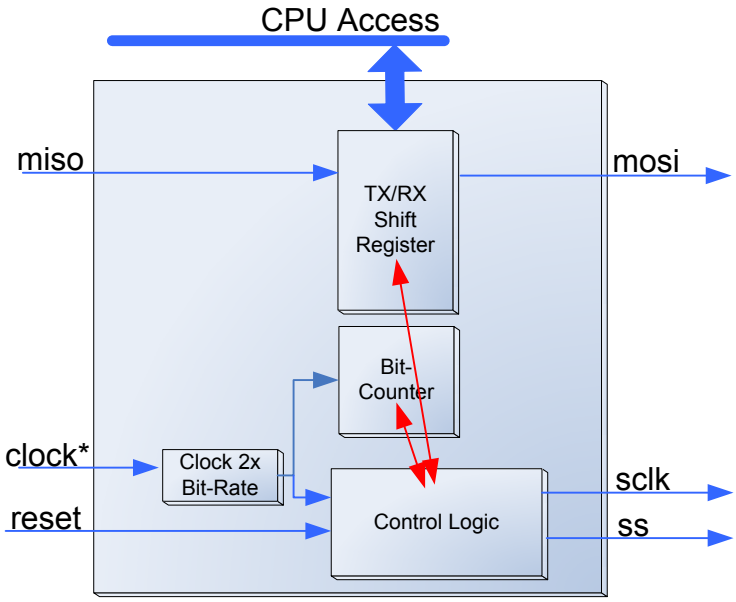


Block Diagram and Configuration

The SPI Master is only available as a UDB configuration. The registers are described here to define the hardware implementation of the SPI Master.



The implementation is described in the following block diagram.



Registers

Tx Status Register

The Tx status register is a read-only register that contains the various transmit status bits defined for a given instance of the SPI Master component. Assuming that an instance of the SPI Master is named “SPIM,” you can get the value of this register using the SPIM_ReadTxStatus() function.

The interrupt output signal is generated by ORing the masked bit fields within the Tx status register. You can set the mask using the SPIM_SetTxInterruptMode() function. Upon receiving an interrupt, you can retrieve the interrupt source by reading the Tx status register with the SPIM_ReadTxStatus () function.

Sticky bits in the Tx status register are cleared on reading, so the interrupt source is held until the SPIM_ReadTxStatus() function is called. All operations on the Tx status register must use the following defines for the bit fields, because these bit fields may be moved within the Tx status register at build time. Sticky bits used to generate an interrupt or DMA transaction must be cleared with either a CPU or DMA read to avoid continuously generating the interrupt or DMA.

There are several bit fields defined for the Tx status registers. Any combination of these bit fields may be included as an interrupt source. The bit fields indicated with an asterisk (*) in the following list are configured as sticky bits in the Tx status register. All other bits are configured as real-time indicators of status. Sticky bits latch a momentary state so that they may be read at a later time and cleared on read. The following #defines are available in the generated header file (for example, *SPIM.h*):

- SPIM_STS_SPI_DONE * – Set high as the data-latching edge of SCLK (edge is mode dependent) is output. This happens after the last bit of the configured number of bits in a single SPI word is output onto the MOSI line and the transmit FIFO is empty. Cleared when the SPI Master is transmitting data or the transmit FIFO has pending data. Tells you when the SPI Master is complete with a multi-word transaction.
- SPIM_STS_TX_FIFO_EMPTY – Reads high while the transmit FIFO contains no data pending transmission. Reads low if data is waiting for transmission.
- SPIM_STS_TX_FIFO_NOT_FULL – Reads high while the transmit FIFO is not full and has room to write more data. Reads low if the FIFO is full of data pending transmit and there is no room for more writes at this time. Tells you when it is safe to pend more data into the transmit FIFO.
- SPIM_STS_BYTE_COMPLETE * – Set high as the last bit of the configured number of bits in a single SPI word is output onto the MOSI line. Cleared* as the data latching edge of SCLK (edge is mode dependent) is output.
- SPIM_STS_SPI_IDLE * – This bit is set high as long as the component state machine is in the SPI IDLE state (component is waiting for Tx data and is not transmitting any data).

RX Status Register

The Rx status register is a read-only register that contains the various receive status bits defined for the SPI Master. You can get the value of this register using the SPIM_ReadRxStatus() function.

The interrupt output signal is generated by ORing the masked bit fields within the Rx status register. You can set the mask using the SPIM_SetRxInterruptMode() function. Upon receiving an interrupt, you can retrieve the interrupt source by reading the Rx status register with the SPIM_ReadRxStatus () function.

Sticky bits in the Rx status register are cleared on reading, so the interrupt source is held until the SPIM_ReadRxStatus() function is called. All operations on the Rx status register must use the following defines for the bit fields, because these bit fields may be moved within the Rx status register at build time. Sticky bits used to generate an interrupt or DMA transaction must be cleared with either a CPU or DMA read to avoid continuously generating the interrupt or DMA.

There are several bit fields defined for the Rx status register. Any combination of these bit fields can be included as an interrupt source. The bit fields indicated with an asterisk (*) in the following list are configured as sticky bits in the Rx status register. All other bits are configured as real-time indicators of status. Sticky bits latch a momentary state so that they may be read at a later time and cleared when read. The following #defines are available in the generated header file (for example, *SPIM.h*):

- SPIM_STS_RX_FIFO_FULL – Reads high when the receive FIFO is full and has no more room to store received data. Reads low if the FIFO is not full and has room for additional received data. Tells you if there is room for new received data to be stored.

- SPIM_STS_RX_FIFO_NOT_EMPTY – Reads high when the receive FIFO is not empty. Reads low if the FIFO is empty and has room for additional received data.
- SPIM_STS_RX_FIFO_OVERRUN * – Reads high when the receive FIFO is already full and additional data was written to it. Tells you if data has been lost from the FIFO because of slow FIFO servicing.

Tx Data Register

The Tx data register contains the transmit data value to send. This is implemented as a FIFO in the SPI Master. There is an optional higher-level software state machine that controls data from the transmit memory buffer. It handles large amounts of data to be sent that exceed the FIFO's capacity. All APIs that involve transmitting data must go through this register to place the data onto the bus. If there is data in this register and the control state machine indicates that data can be sent, then the data is 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 can be set up to fill this FIFO when empty, using the TXDATA_REG address defined in the header file.

Rx Data Register

The Rx data register contains the received data. This is implemented as a FIFO in the SPI Master. There is an optional higher-level software state machine that controls data movement from this receive FIFO into the memory buffer. Typically, the Rx interrupt indicates that data has been received. At that time, that data has several routes to the firmware. DMA can be set up from this register to the memory array, or the firmware can simply call the SPIM_ReadRxData() function. DMA must use the RXDATA_REG address defined in the header file.

Conditional Compilation Information

The SPI Master requires only one conditional compile definition to handle the 8- or 16-bit datapath configuration necessary to implement the configured NumberOfDataBits. The API must conditionally compile for the data width defined. APIs should never use these parameters directly but should use the following define:

- SPIM_DATAWIDTH – This defines how many data bits will make up a single "byte" transfer. Valid range is 3 to 16 bits.

Resources

The SPI Master component is placed throughout the UDB array. The component utilizes the following resources.

Configuration	Resource Type					
	Datapath Cells	Macrocells	Status Cells	Control Cells	DMA Channels	Interrupts
8-bit (MOSI+MISO)	1	12	2	1	–	2
8-bit (Bidirectional)	1	12	2	2	–	2
16-bit (MOSI+MISO)	2	12	2	1	–	2
16-bit (Bidirectional)	2	12	2	2	–	2
8-bit High Speed (MOSI+MISO)	1	18	2	1	–	2
8-bit High Speed (Bidirectional)	1	18	2	2	–	2
16-bit High Speed (MOSI+MISO)	2	18	2	1	–	2
16-bit High Speed (Bidirectional)	2	18	2	2	–	2

API Memory Usage

The component memory usage varies significantly, depending on the compiler, device, number of APIs used and component configuration. The following table provides the memory usage for all APIs available in the given component configuration.

The measurements have been done with associated compiler configured in Release mode with optimization set for Size. For a specific design the map file generated by the compiler can be analyzed to determine the memory usage.

Configuration	PSoC 3 (Keil_PK51)		PSoC 5 (GCC)		PSoC 5LP (GCC)	
	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
8-bit (MOSI+MISO)	423	3	624	5	516	5
8-bit (Bidirectional)	423	3	604	5	516	5
16-bit (MOSI+MISO)	483	3	644	9	556	5
16-bit (Bidirectional)	489	3	644	9	556	5
8-bit High Speed (MOSI+MISO)	423	3	608	5	516	5

Configuration	PSoC 3 (Keil_PK51)		PSoC 5 (GCC)		PSoC 5LP (GCC)	
	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
8-bit High Speed (Bidirectional)	417	3	604	5	516	5
16-bit High Speed (MOSI+MISO)	487	3	644	9	556	5
16-bit High Speed (Bidirectional)	489	3	644	9	556	5

DC and AC Electrical Characteristics

Specifications are valid for –40 °C ≤ T_A ≤ 85 °C and T_J ≤ 100 °C, except where noted.
Specifications are valid for 1.71 V to 5.5 V, except where noted.

DC Characteristics

Parameter	Description	Min	Typ[1]	Max	Units[2]
$I_{DD(8\text{-bit})}$	Component current consumption (8-bit; MOSI+MISO)				
	Idle current[3]	–	20	–	µA/MHz
	Operating current[4]	–	28	–	µA/MHz
	Component current consumption (8-bit; Bidirectional)				
	Idle current[3]	–	21	–	µA/MHz
	Operating current[4]	–	30	–	µA/MHz
	Component current consumption (8-bit; High Speed; MOSI+MISO)				
	Idle current[3]	–	23	–	µA/MHz
	Operating current[4]	–	34	–	µA/MHz
	Component current consumption (8-bit; High Speed; Bidirectional)				
	Idle current[3]	–	27	–	µA/MHz
	Operating current[4]	–	36	–	µA/MHz

1. Device IO and clock distribution current not included. The values are at 25 °C.
2. Current consumption is specified with respect to the incoming component clock.
3. Current consumed by component while it is enabled but not transmitting/receiving data.
4. Current consumed by component while it is enabled and transmitting/receiving data.

Parameter	Description	Min	Typ[1]	Max	Units[2]
$I_{DD(16\text{-}bit)}$	Component current consumption (16-bit; MOSI+MISO)				
	Idle current[3]	–	23	–	µA/MHz
	Operating current[4]	–	30	–	µA/MHz
	Component current consumption (16-bit; Bidirectional)				
	Idle current[3]	–	25	–	µA/MHz
	Operating current[4]	–	32	–	µA/MHz
	Component current consumption (16-bit; High Speed; MOSI+MISO)				
	Idle current[3]	–	27	–	µA/MHz
	Operating current[4]	–	38	–	µA/MHz
	Component current consumption (16-bit; High Speed; Bidirectional)				
	Idle current[3]	–	30	–	µA/MHz
	Operating current[4]	–	40	–	µA/MHz

AC Characteristics

Parameter	Description	Min	Typ	Max [5]	Units
f_{SCLK}	SCLK frequency				
	8-bit (MOSI+MISO)	–	–	9	MHz
	8-bit (Bidirectional)	–	–	9	MHz
	16-bit (MOSI+MISO)	–	–	8	MHz
	16-bit (Bidirectional)	–	–	8	MHz
	8-bit High Speed (MOSI+MISO)	–	–	18	MHz
	8-bit High Speed (Bidirectional)	–	–	18	MHz
	16-bit High Speed (MOSI+MISO)	–	–	16	MHz
	16-bit High Speed (Bidirectional)	–	–	16	MHz
f_{CLOCK}	Component clock frequency	–	$2 \times f_{SCLK}$	–	MHz
t_{CKH}	SCLK high time	–	0.5	–	$1/f_{SCLK}$

5. The maximum component clock frequency is derived from t_{SCLK_MISO} in combination with the routing path delays of the SCLK input and the MISO output (described later in this document). These “Nominal” numbers provide a maximum safe operating frequency of the component under nominal routing conditions. It is possible to run the component at higher clock frequencies, at which point you will need to validate the timing requirements with STA results.

Parameter	Description	Min	Typ	Max [5]	Units
t_{CKL}	SCLK low time	–	0.5	–	$1/f_{SCLK}$
t_{S_MISO}	MISO input setup time	25	–	–	ns
t_{H_MISO}[6]	MISO input hold time	–	0	–	ns
t_{SS_SCLK}	SS active to SCLK active	-20		20	ns
t_{SCLK_SS}	SCLK inactive to SS inactive	-20		20	ns

6. The V_{DDIO} supply voltage from 3.0 V to 5.5 V. The value can vary for PSoC 5, so must be validated with STA results.

Figure 8. Mode CPHA = 0 Timing Diagram

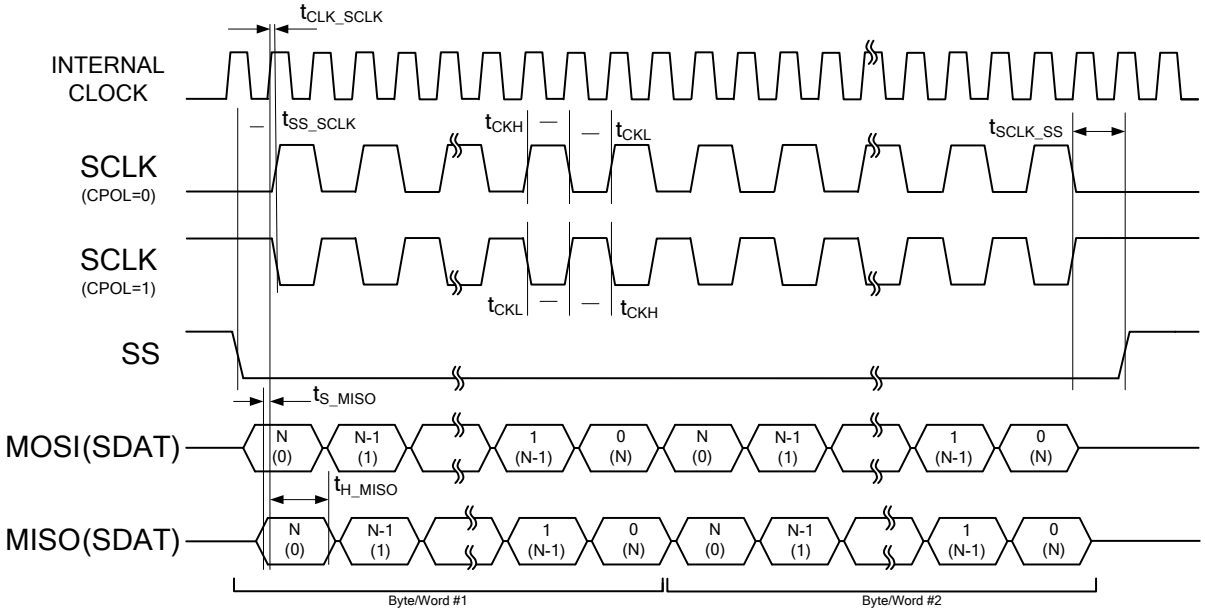
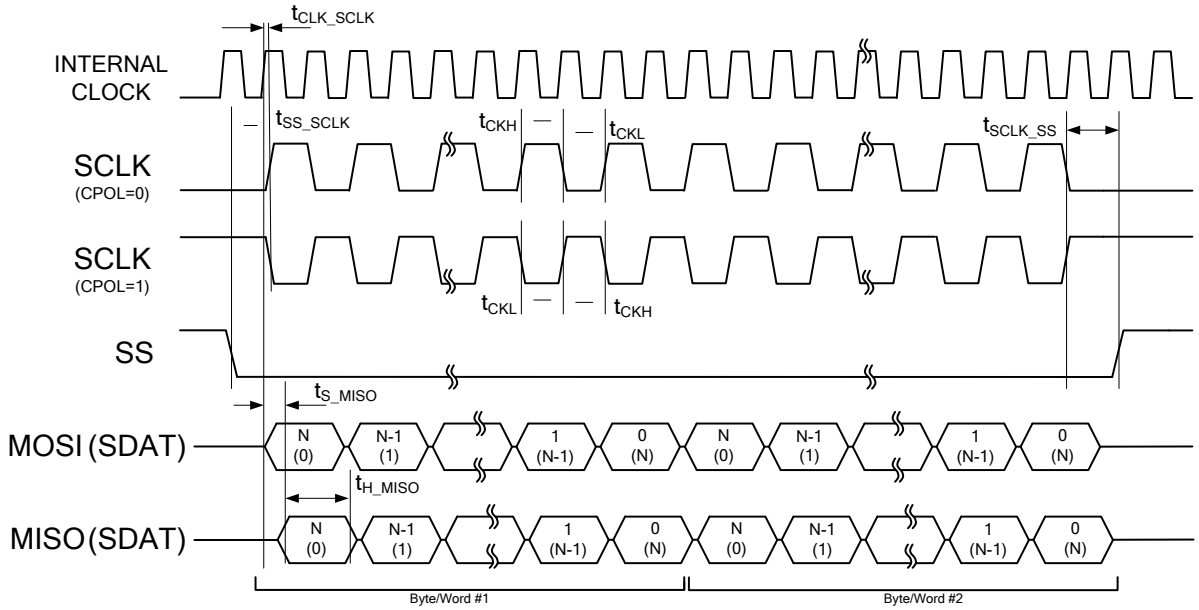


Figure 9. Mode CPHA = 1 Timing Diagram

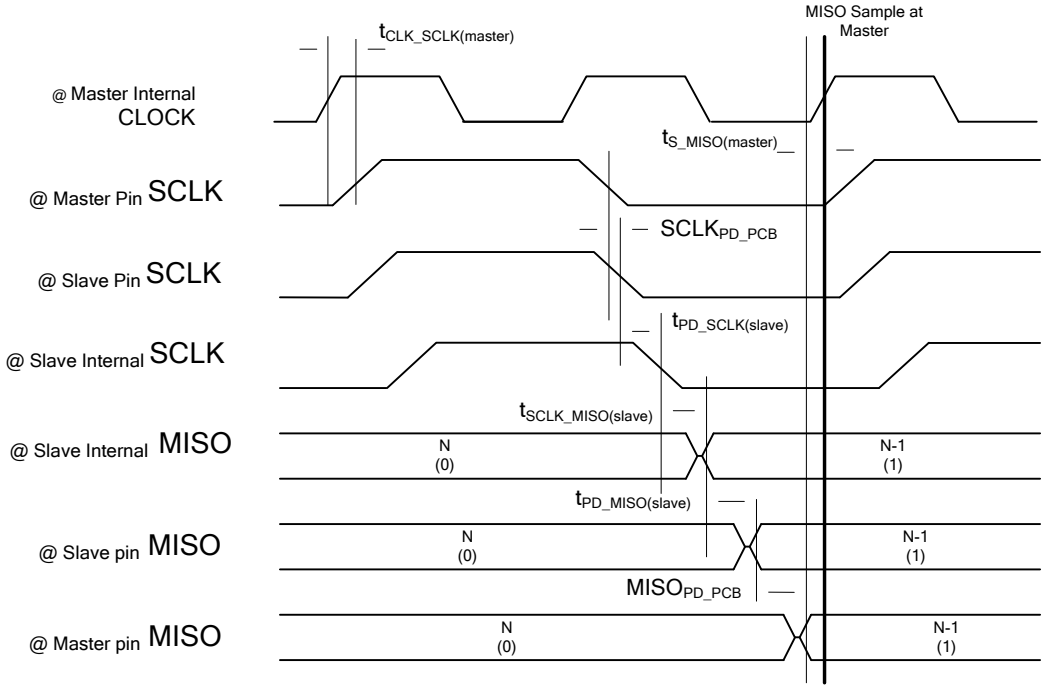


How to Use STA Results for Characteristics Data

Nominal route maximums are gathered through multiple test passes with Static Timing Analysis (STA). You can calculate the maximums for your designs using the STA results with the following mechanisms:

f_{SCLK} The maximum frequency of SCLK (or the maximum bit rate) is not provided directly in the STA. However, the data provided in the STA results indicates some of the internal logic timing constraints. To calculate the maximum bit rate, you must consider several factors. You will need board layout and slave communication device specs to fully understand the maximum. The main limiting factor in this parameter is the round trip path delay from the falling edge of SCLK at the pin of the master, to the slave and the path delay of the MISO output of the slave back to the master.

Figure 10 Calculating Maximum f_{SCLK} Frequency



In this case, the component must meet the setup time of MISO at the Master using the equation below:

$$t_{RT_PD} < 1 / \{ [½ \times f_{SCLK}] - t_{CLK_SCLK(master)} - t_{S_MISO(master)} \}$$

-- OR --

$$f_{SCLK} < 1 / \{2 * [T_{RT_PD} + t_{CLK_SCLK(master)} + t_{S_MISO(master)}] \}$$

Where:

$t_{CLK_SCLK(master)}$ is the path delay of the input CLK to the SCLK output. This is provided in the STA results clock to output section as shown below:

- **Clock To Output Section**

  - **SPIM_1_IntClock**

Source	Destination	Delay (ns)
Net 25/q	SCLK 1(0) PAD	24.800

$t_{S_MISO(Master)}$ is the path delay from MISO input pin to internal logic of the master component. This is provided in the STA results input to clock section as shown below:

- **Input To Clock Section**

  - **SPIM_1_IntClock**

Source	Destination	Delay (ns)
MISO 1(0) PAD	\SPIM 1:BSPIM:sR8:Dp:u0\/route si	20.906

For High Speed Mode setup time equation is as follows:

$$t_{RT_PD} < 1 / \{ [f_{SCLK}] - t_{CLK_SCLK(master)} - t_{S_MISO(master)}\}$$

-- OR --

$$f_{SCLK} < 1 / \{ [T_{RT_PD} + t_{CLK_SCLK(master)} + t_{S_MISO(master)}] \}$$

Note There are two values in the report: setup times to the component clock at register A0 and register A1. You should select the larger one for calculation.

And t_{RT_PD} is defined as:

$$t_{RT_PD} = [SCLK_{PD_PCB} + t_{PD_SCLK(slave)} + t_{SCLK_MISO(slave)} + t_{PD_MISO(slave)} + MISO_{PD_PCB}]$$

and:

$SCLK_{PD_PCB}$ is the PCB path delay of SCLK from the pin of the master component to the pin of the slave component.

$t_{PD_SCLK(slave)} + t_{SCLK_MISO(slave)} + t_{PD_MISO(slave)}$ must come from the slave device datasheet. $MISO_{PD_PCB}$ is the PCB path delay of MISO from the pin of the slave component to the pin of the master component.

The final equation that provides the maximum frequency of SCLK, and therefore the maximum bit rate, is:

$$f_{SCLK} (Max.) = 1 / \{2 \times [t_{CLK_SCLK(master)} + SCLK_{PD_PCB} + t_{PD_SCLK(slave)} + t_{SCLK_MISO(slave)} + t_{PD_MISO(slave)} + MISO_{PD_PCB} + t_{S_MISO(master)}] \}$$

For High Speed Mode:

$$f_{SCLK} (Max.) = 1 / \{ [t_{CLK_SCLK(master)} + SCLK_{PD_PCB} + t_{PD_SCLK(slave)} + t_{SCLK_MISO(slave)} + t_{PD_MISO(slave)} + MISO_{PD_PCB} + t_{S_MISO(master)}] \}$$

f_{CLOCK} Maximum component clock frequency is provided in Timing results in the clock summary as the IntClock (if internal clock is selected) or the named external clock. An example of the internal clock limitations from the STA report file is shown below:

- Clock Summary Section

Clock	Type	Nominal Frequency (MHz)	Required Frequency (MHz)	Maximum Frequency (MHz)	Violation
BUS_CLK	Sync	60.000	60.000	N/A	
ClockBlock/clk_bus	Async	60.000	60.000	N/A	
ClockBlock/dclk_0	Async	15.000	15.000	N/A	
ILO	Async	0.001	0.001	N/A	
IMO	Async	3.000	3.000	N/A	
MASTER_CLK	Sync	60.000	60.000	N/A	
PLL_OUT	Async	60.000	60.000	N/A	
SPIM_1_IntClock	Sync	15.000	15.000	61.870	

Its value is limited to either 2x of the f_{CLOCK} or by the number from the STA report but in practice the limitation is 2x of the f_{CLOCK}.

t_{CKH} The SPI Master component generates a 50-percent duty cycle SCLK.

t_{CKL} The SPI Master component generates a 50-percent duty cycle SCLK.

t_{CLK_SCLK} Internal clock to SCLK output time. Time from posedge of Internal Clock to SCLK available on master pin.

t_{S_MISO} To meet the setup time of the internal logic, MISO must be valid at the pin, before Internal clock is valid at the pin, by this amount of time.

t_{H_MISO} To meet the hold time of the internal logic, MISO must be valid at the pin, after Internal clock is valid at the pin, by this amount of time.

t_{SS_SCLK} To meet the internal functionality of the block, Slave Select (SS) must be valid at the pin before SCLK is valid at the pin, by this parameter.

t_{SCLK_SS} Maximum - To meet the internal functionality of the block. Slave Select (SS) must be valid at the pin after the last falling edge of SCLK at the pin, by this parameter.

Component Changes

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

Version	Description of Changes	Reason for Changes / Impact
2.30	Added all component APIs with the CYREENTRANT keyword when they are included in the .cyre file.	Not all APIs are truly reentrant. Comments in the component API source files indicate which functions are candidates. This change is required to eliminate compiler warnings for functions that are not reentrant used in a safe way: protected from concurrent calls by flags or Critical Sections.
	Added PSoC 5LP support	
	Added DC characteristics section to datasheet	
2.21	High Speed Mode functionality is added. Fixed SPI Modes diagrams	Verilog implementation is updated to allow maximum bit rate up to 18 Mbps. Added details to description of Bi-directional Mode. SPI Modes diagrams changed to hide Rx data as internal implementation details. Description of High Speed Mode is added to “Advanced Tab” section. “Resources”, “AC and DC electrical Characteristics” and “How to use STA results for Characteristics data” sections are updated with data related to High Speed Mode.
2.20	Internal clock component has been updated with cy_clock_v1_60. Fixed verilog defect that caused STA warning.	Clock v1_60 is the latest component version. Verilog defect fixed to remove STA warning founded using the updated STA tool. Screenshots in “AC and DC electrical characteristics” section are updated by report generated by the new STA tool.
2.10.a	Datasheet corrections (CPHA = 1 Diagrams for modes where CPHA = 1 corrected)	Datasheet defects fixed
2.10	Data Bits range is changed from 2 to 16 bits to 3 to 16.	Changes related to status synchronization issues fixed in current version.
	“Interrupt on SPI Idle” checkbox is added to the component configure dialog.	Component customizer defect fixed
	“Byte transfer complete” checkbox name is changed to the “Byte/Word transfer complete”	To fit the real meaning
	Added characterization data to datasheet	
	Minor datasheet edits and updates	
2.0.a	Moved component into subfolders of the component catalog.	

Version	Description of Changes	Reason for Changes / Impact
2.0	Added SPIM_Sleep()/SPIM_Wakeup() and SPIM_Init()/SPIM_Enable APIs.	To support low-power modes, and to provide common interfaces to separate control of initialization and enabling of most components.
	Number and positions of component I/Os have been changed: ▪ The clock input is now visible in default placement (external clock source is the default setting now) ▪ The reset input has a different position ▪ The interrupt output was removed. rx_interrupt, tx_interrupt outputs are added instead.	The clock input was added for consistency with SPI Slave. The reset input place changed because the clock input was added. Two status interrupt registers (Tx and Rx) are now presented instead of one shared. These changes be taken into account to prevent binding errors when migrating from previous SPI versions
	Removed SPIM_EnableInt(), SPIM_DisableInt(), SPIM_SetInterruptMode(), and SPIM_ReadStatus() APIs. Added SPIM_EnableTxInt(), SPIM_EnableRxInt(), SPIM_DisableTxInt(), SPIM_DisableRxInt(), SPIM_SetTxInterruptMode(), SPIM_SetRxInterruptMode(), SPIM_ReadTxStatus(), SPIM_ReadRxStatus() APIs.	The removed APIs are obsolete because the component now contains Rx and Tx interrupts instead of one shared interrupt. Also updated the interrupt handler implementation for Tx and Rx Buffer.
	Renamed SPIM_ReadByte(), SPIM_WriteByte()APIs to SPIM_ReadRxData(), SPIM_WriteTxData().	Clarifies the APIs and how they should be used.
The following changes were made to the base SPI Master component B_SPI_Master_v2_0, which is implemented using Verilog:		
	spim_ctrl internal module was replaced by a new state machine.	It uses fewer hardware resources and does not contain any asynchronous logic.

Version	Description of Changes	Reason for Changes / Impact

	Two status registers are now presented (status are separate for Tx and Rx) instead of using one common status register for both.
	`/*SPI_Master_v1_20 status bits*/`
	`SPIM_STS_SPI_DONE_BIT = 3'd0;`
	`SPIM_STS_TX_FIFO_EMPTY_BIT = 3'd1;`
	`SPIM_STS_TX_FIFO_NOT_FULL_BIT = 3'd2;`
	`SPIM_STS_RX_FIFO_FULL_BIT = 3'd3;`
	`SPIM_STS_RX_FIFO_NOT_EMPTY_BIT = 3'd4;`
	`SPIM_STS_RX_FIFO_OVERRUN_BIT = 3'd5;`
	`SPIM_STS_BYTE_COMPLETE_BIT = 3'd6;`
	
	`/*SPI_Master_v2_0 status bits*/`
	`localparam SPIM_STS_SPI_DONE_BIT = 3'd0;`
	`localparam SPIM_STS_TX_FIFO_EMPTY_BIT = 3'd1;`
	`localparam SPIM_STS_TX_FIFO_NOT_FULL_BIT = 3'd2;`
	`localparam SPIM_STS_BYTE_COMPLETE_BIT = 3'd3;`
	`localparam SPIM_STS_SPI_IDLE_BIT = 3'd4;`
	`localparam SPIM_STS_RX_FIFO_FULL_BIT = 3'd4;`
	`localparam SPIM_STS_RX_FIFO_NOT_EMPTY_BIT = 3'd5;`
	`localparam SPIM_STS_RX_FIFO_OVERRUN_BIT = 3'd6;`
		Fixed a defect found in previous versions of the component where software buffers were not working as expected.

Version	Description of Changes	Reason for Changes / Impact
	'BidirectMode' boolean parameter is added to base component. Control Register with 'clock' input and SYNC mode bit is now selected to drive 'tx_enable' output for PSoC 3 Production silicon. Control Register w/o clock input drives 'tx_enable' when ES2 silicon is selected. Bufoe component is used on component schematic to support Bidirectional Mode. MOSI output of base component is connected to bufoe 'x' input. 'yfb' is connected to 'miso' input. Bufoe 'y' output is connected to 'sdat' output terminal. Routed reset is connected to datapaths, Counter7 and State Machine.	Added Bidirectional Mode support to the component
	udb_clock_enable component is added to Verilog implementation with sync = `TRUE` parameter.	New requirements for all clocks used in Verilog to indicate functionality so the tool can support synchronization and Static Timing Analysis.
	'*2' is replaced by '<< 1' in Counter7 period value.	Verilog improvements.
	Maximum Bit Rate value is changed to 10 Mbps	Bit Rate value more than 10 Mbps is not supported (verified during the component characterization)
	Description of the Bidirectional Mode is added	Data sheet defect fixed
	Reset input description now contains the note about ES2 silicon incompatibility	Data sheet defect fixed
	Timing correlation diagram between SS and SCLk signals is changed	Data sheet defect fixed
	Sample firmware source code is removed	Reference to the component example project is added instead
	SPI Modes diagrams are changed (Tx and Rx FIFO status values are added)	Data sheet defect fixed

© Cypress Semiconductor Corporation, 2012. 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.