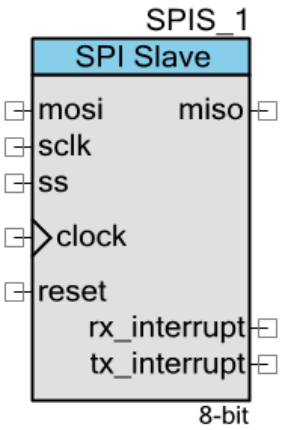
Serial Peripheral Interface (SPI) Slave

2.41

Features

- 3- to 16-bit data width
- Four SPI modes
- Bit rate up to 5 Mbps[1]



General Description

The SPI Slave provides an industry-standard, 4-wire slave 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 master device. In addition to the standard 8-bit word length, the SPI Slave 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 Slave

You can use the SPI Slave component any time a PSoC device is required to interface with an SPI Master device. In addition to SPI Master labeled devices, you can use the SPI Slave with many devices implementing a shift register type interface.

Use the SPI Master component in instances requiring a PSoC device to communicate with an SPI Slave device. Use the Shift Register component in situations where its low-level flexibility provides hardware capabilities not available in the SPI Slave component.

---

[1] This value is valid only for MOSI+MISO (Full Duplex) interfacing mode (see the DC and AC Electrical Characteristics section for details) and is restricted up to 1 Mbps in Bidirectional mode because of internal bidirectional pin constraints.

Input/Output Connections

This section describes the various input and output connections for the SPI. 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.

mosi – Input *

The mosi input carries the Master Output Slave Input (MOSI) signal from a master 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 **Bidirectional**. For both PSoC 3 and PSoC 5 silicon an asynchronous clock crossing warning will be reported between the component clock and the SCLK signal when timing analysis is performed. The following is an example of such a message: “Path(s) exist between clocks IntClock and SCLK(0)_PAD, but the clocks are not synchronous to each other.” This message applies to a path from the register that controls the direction and the sampling of data by SCLK. SCLK should not be running when the direction is being changed. As long as this rule is followed, there is no problem and you can ignore this message.

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

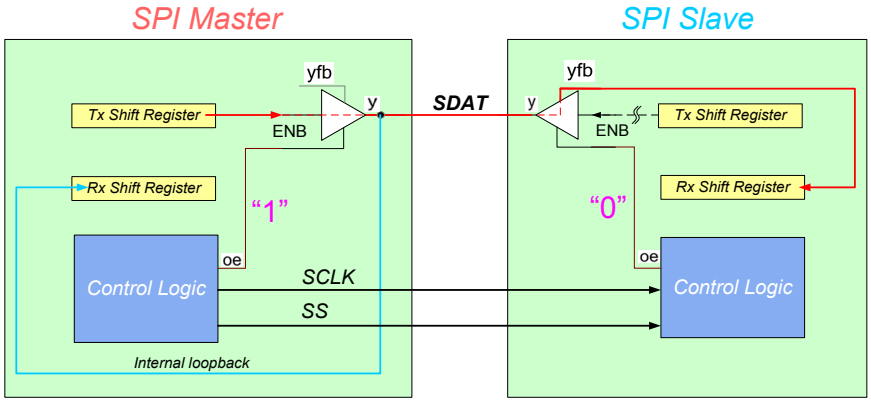
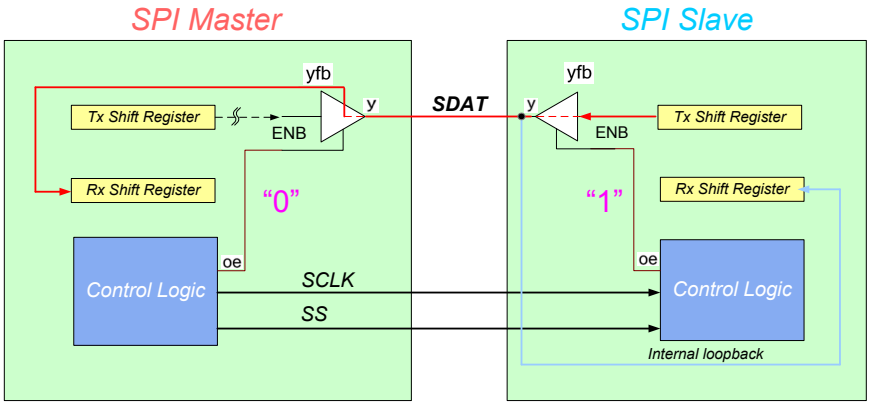


Figure 2. SPI Bidirectional Mode (data transmission from Slave to Master)



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

sclk– Input

The sclk input carries the Serial Clock (SCLK) signal. It provides the slave synchronization clock input to the device. This input is always visible and must be connected.

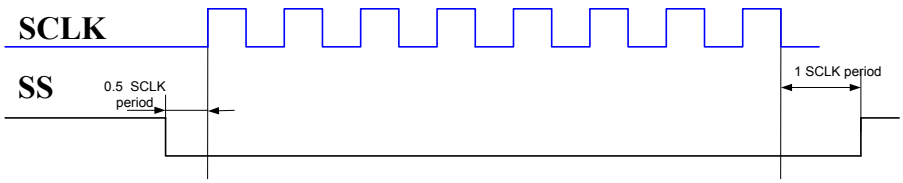
Note Some SPI Master devices (such as the TotalPhase Aardvark I2C/SPI host adapter) drive the sclk output in a specific way. For the SPI Slave component to function properly with such devices in modes where CPOL = 1, the sclk pin should be set to resistive pull-up drive mode. Otherwise, it puts out corrupted data. See the Functional Description section for more information about modes.

ss – Input

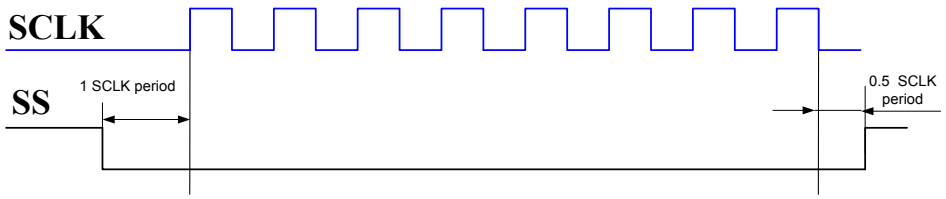
The ss input carries the Slave Select (SS) signal to the device. This input is always visible and must be connected.

The following diagrams show the timing correlation between SCLK and SS signals

CPHA = 0:



CPHA = 1:



Note The SS timing shown in these diagrams is valid for the PSoC Creator SPI Master. Generally, 0.5 of the SCLK period is enough delay between the SS negative edge and the first SCLK edge for the SPI Slave to work correctly in all supported bit-rate ranges.

reset – Input

The reset input resets the SPI Slave. It deletes 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. PSoC 5 silicon do not support this reset functionality, so this input is ignored when used with those devices. Use of the reset input results in an asynchronous clock crossing warning being reported between the clock that generates the Reset input and the SCLK signal when timing analysis is performed. The following is an example of such a message: “Path(s) exist between clocks BUS_CLK and SCLK(0)_PAD, but the clocks are not synchronous to each other.” This message applies to a path from the Reset signal to the operation of the SPI component clocked by SCLK. SCLK should not be running when the Reset signal is changed. As long as this rule is followed, there is no problem and you can ignore this message.

The reset input may be left floating with no external connection. If nothing is connected to the reset line the component will assign it a constant logic 0.

clock – Input *

The clock input defines the sampling rate of the status register. All data clocking happens on the sclk input, so the clock input does **not** handle the bit-rate of the SPI Slave.

The clock input is visible when the **Clock Selection** parameter is set to **External**. If visible, this input must be connected.

miso – Output *

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

interrupt – Output

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

Schematic Macro Information

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

Figure 3. 4-wire (Full Duplex) Interfacing Schematic Macro

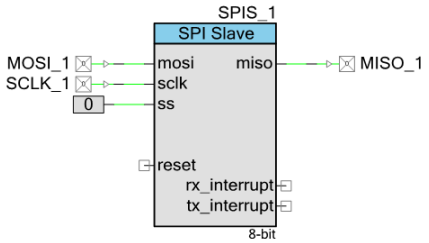


Figure 4. 3-wire (Bidirectional) Interfacing Schematic Macro

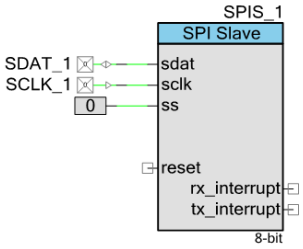
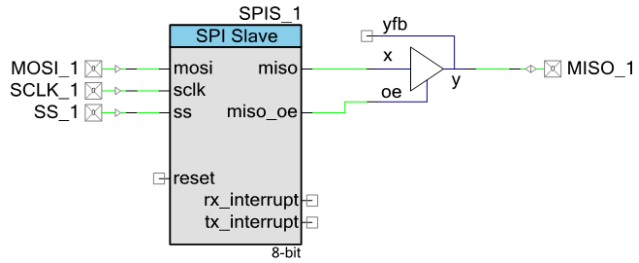


Figure 5. Multislave Mode Schematic Macro



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

Component Parameters

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

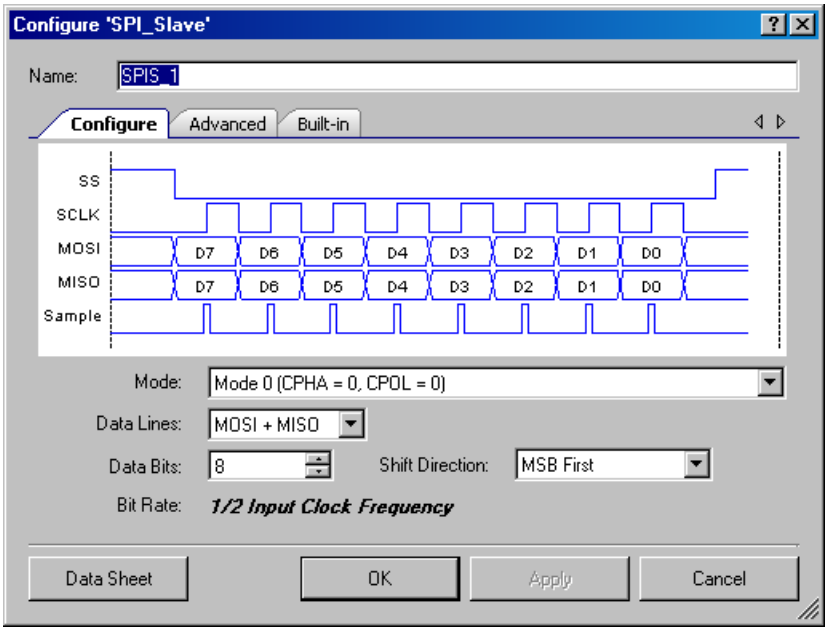
The following sections describe the SPI Slave parameters, and how they are configured using the Configure dialog. They also indicate whether the options are hardware or software.

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 provided APIs. Hardware-only parameters are marked with an asterisk (*).

Configure Tab

The **Configure** tab contains basic parameters required for every SPI component.



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

Mode *

The **Mode** parameter defines the desired clock phase and clock polarity mode used in the communication. These modes are defined in the following table. Refer also to the Functional Description section of this datasheet.

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 SPIS_ReadRxData() and SPIS_WriteTxData() APIs. The default number of bits is a single byte (8 bits). Any integer from 3 to 16 is a valid value.

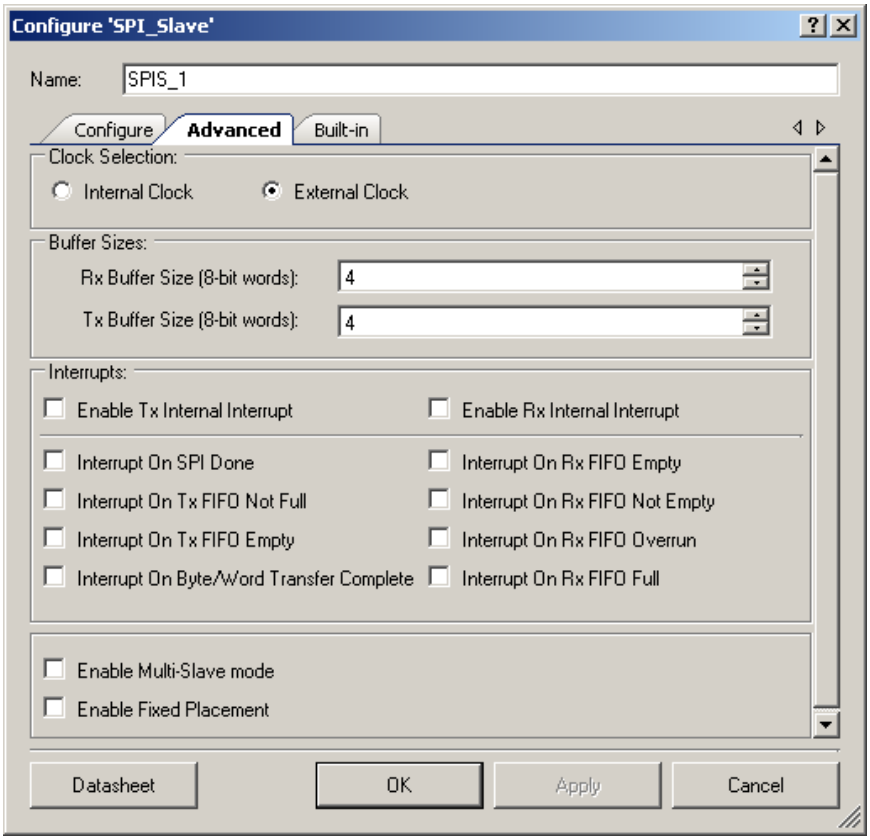
Shift Direction *

The **Shift Direction** parameter defines the direction the serial data is transmitted. When set to **MSB First**, the most significant bit is transmitted first. 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 is 2x the SCLK rate. This parameter has no effect if the **Clock Selection** parameter is set to **External Clock**.

Advanced Tab



Clock Selection

The **Clock Selection** parameter specifies whether to use an internal clock or external clock. Refer to the Clock Selection section later in this datasheet for more information.

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 previous versions; using them causes an error message saying 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.

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 FIFO is implemented in the hardware. Values 1 to 3 are available only for compatibility with previous versions; setting them causes an error message saying 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 Tx/Rx Internal Interrupt

The **Enable Tx/Rx Internal Interrupt** options allow you to use the predefined Tx and Rx ISRs of the SPI Slave component, or supply your own custom ISRs. If these options are 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 Slave.

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 the transfer of data from the hardware FIFO to the Rx or Tx buffer, or both. The interrupt output pins of the SPI Slave 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 global interrupt and component’s 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.

Enable Multi-Slave mode

This setting is used when the current SPI Slave component is connected to the shared bus with other SPI Slave devices. The MISO_OE output becomes visible on the component symbol. The external BUF_OE component should be connected to the MISO output in this mode. This mode allows you to turn MISO output to a high-impedance state when SS line is high. The multi-slave mode macro can be used to provide all necessary connections quickly.

Enable Fixed Placement

This setting is used to improve SPI Slave component performance in comparison with unconstrained placement. Fixed placement offers a single placement for a component. This means that only one instance of a component can be used in a single design. Placement of the pins connected to the component is not controlled, but it is preferable to use pins from P[0] to achieve the best performance.

The fixed placement aspect of the component removes the variability that is accounted for with the “Maximum with All Routing” case (see DC and AC Electrical Characteristics for details). It also allows the fixed placement to continue to operate the same as a non-fixed placed design would in a fairly empty design.

Clock Selection

When the internal clock configuration is selected, PSoC Creator calculates the required 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, at a minimum, 2x the maximum bit-rate and SCLK frequency.

Note When setting the bit-rate or external clock frequency value, make sure that this value can be provided by PSoC Creator using the current system clock frequency. Otherwise, a warning about the clock accuracy range will be 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.

Placement

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

Application Programming Interface

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

By default, PSoC Creator assigns the instance name “SPIS_1” to the first instance of a component in a given design. You can rename the instance to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance name used in the following table is “SPIS.”

Function	Description
SPIS_Start()	Calls both SPIS_Init() and SPIS_Enable(). Should be called the first time the component is started.
SPIS_Stop()	Disables SPIS operation.
SPIS_EnableTxInt()	Enables the internal Tx interrupt irq.
SPIS_EnableRxInt()	Enables the internal Rx interrupt irq.
SPIS_DisableTxInt()	Disables the internal Tx interrupt irq.
SPIS_DisableRxInt()	Disables the internal Rx interrupt irq.
SPIS_SetTxInterruptMode()	Configures the Tx interrupt sources enabled.
SPIS_SetRxInterruptMode()	Configures the Rx interrupt sources enabled.
SPIS_ReadTxStatus()	Returns the current state of the Tx status register.
SPIS_ReadRxStatus()	Returns the current state of the Rx status register.
SPIS_WriteTxData()	Places a byte/word in the transmit buffer that will be sent at the next available bus time.
SPIS_WriteTxDataZero()	Places a byte/word in the shift register directly. This is required for SPI Modes where CPHA = 0.
SPIS_ReadRxData()	Returns the next byte/word of received data available in the receive buffer.
SPIS_GetRxBufferSize()	Returns the size (in bytes/words) of received data in the Rx memory buffer.
SPIS_GetTxBufferSize()	Returns the size (in bytes/words) of data waiting to transmit in the Tx memory buffer.
SPIS_ClearRxBuffer()	Clears the Rx buffer memory array and Rx FIFO of all received data.
SPIS_ClearTxBuffer()	Clears the Tx buffer memory array and Tx FIFO of all transmit data. **Note** Tx FIFO will be cleared only if software buffer is not used.
SPIS_TxEnable()	If configured for bidirectional mode, sets the SDAT inout to transmit.
SPIS_TxDisable()	If configured for bidirectional mode, sets the SDAT inout to receive.
SPIS_PutArray()	Places an array of data into the transmit buffer.

Function	Description
SPIS_ClearFIFO()	Clears any received data from the Rx hardware FIFO.
SPIS_Sleep()	Prepares SPIS component for low-power modes by calling SPIS_SaveConfig() and SPIS_Stop() functions.
SPIS_Wakeup()	Restores and re-enables the SPIS component after waking from low-power mode.
SPIS_Init()	Initializes and restores the default SPIS configuration.
SPIS_Enable()	Enables the SPIS to start operation.
SPIS_SaveConfig()	Saves SPIS hardware configuration.
SPIS_RestoreConfig()	Restores SPIS hardware configuration.

Global Variables

Function	Description
SPIS_initVar	SPIS_initVar indicates whether the SPI Slave component has been initialized. The variable is initialized to 0 and set to 1 the first time SPIS_Start() is called. This allows the component to restart without reinitialization after the first call to the SPIS_Start() routine. If reinitialization of the component is required, then the SPIS_Init() function can be called before the SPIS_Start() or SPIS_Enable() function.
SPIS_txBufferWrite	Transmit buffer location of the last data written into the buffer by the API.
SPIS_txBufferRead	Transmit buffer location of the last data read from the buffer and transmitted by SPIS hardware.
SPIS_rxBufferWrite	Receive buffer location of the last data written into the buffer after received by SPIS hardware.
SPIS_rxBufferRead	Receive buffer location of the last data read from the buffer by the API.
SPIS_rxBufferFull	Indicates the software buffer has overflowed.
SPIS_RXBUFFER[]	Used to store received data.
SPIS_TXBUFFER[]	Used to store data for sending.

void SPIS_Start(void)

Description: This is the preferred method to begin component operation. SPIS_Start() sets the initVar variable, calls the SPIS_Init() function, and then calls the SPIS_Enable() function.

Parameters: None

Return Value: None

Side Effects: None

void SPIS_Stop(void)

Description: Disables the SPI Slave component interrupts. Has no affect on the SPIS operation.

Parameters: None

Return Value: None

Side Effects: None

void SPIS_EnableTxInt(void)

Description: Enables the internal Tx interrupt irq.

Parameters: None

Return Value: None

Side Effects: None

void SPIS_EnableRxInt(void)

Description: Enables the internal Rx interrupt irq.

Parameters: None

Return Value: None

Side Effects: None

void SPIS_DisableTxInt(void)

Description: Disables the internal Tx interrupt irq

Parameters: None

Return Value: None

Side Effects: None

void SPIS_DisableRxInt(void)

Description: Disables the internal Rx interrupt irq

Parameters: None

Return Value: None

Side Effects: None

void SPIS_SetTxInterruptMode(uint8 intSrc)

Description: Configures the Tx interrupt sources that are enabled.

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

Bit	Description
SPIS_STS_SPI_DONE	Enable interrupt due to SPI done
SPIS_STS_TX_FIFO_EMPTY	Enable interrupt due to Tx FIFO empty
SPIS_STS_TX_FIFO_NOT_FULL	Enable interrupt due to Tx FIFO not full
SPIS_STS_BYTE_COMPLETE	Enable interrupt due to byte/word complete

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, refer to the Defines section in this datasheet.

Return Value: None

Side Effects: None

void SPIS_SetRxInterruptMode(uint8 intSrc)

Description: Configures the Rx interrupt sources that are enabled.

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

Bit	Description
SPIS_STS_RX_FIFO_EMPTY	Enable interrupt due to Rx FIFO empty
SPIS_STS_RX_FIFO_NOT_EMPTY	Enable interrupt due to Rx FIFO not empty
SPIS_STS_RX_FIFO_OVERRUN	Enable interrupt due to Rx Buf overrun
SPIS_STS_RX_FIFO_FULL	Enable interrupt due to Rx FIFO full

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, refer to the Defines section in this datasheet.

Return Value: None

Side Effects: None

uint8 SPIS_ReadTxStatus(void)

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

Parameters: None

Return Value: uint8: Current Tx status register value

Bit	Description
SPIS_STS_SPI_DONE	SPI done
SPIS_STS_TX_FIFO_EMPTY	TX FIFO empty
SPIS_STS_TX_FIFO_NOT_FULL	TX FIFO not full
SPIS_STS_BYTE_COMPLETE	Byte/Word complete

Side Effects: Tx status register bits are clear on read.

uint8 SPIS_ReadRxStatus(void)

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

Parameters: None

Return Value: uint8: Current Rx status register value

Bit	Description
SPIS_STS_RX_FIFO_EMPTY	Rx FIFO empty
SPIS_STS_RX_FIFO_NOT_EMPTY	Rx FIFO not empty
SPIS_STS_RX_FIFO_OVERRUN	Rx Buf overrun
SPIS_STS_RX_FIFO_FULL	Rx FIFO full

Side Effects: Rx status register bits are clear on read.

void SPIS_WriteTxData(uint8/uint16 txData)

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

Parameters: uint8/uint16: txData: The data value to send across 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.

void SPIS_WriteTxDataZero(uint8/uint16 txData)

Description: Places a byte/word directly into the shift register for transmission. This byte/word will be sent from the master device during the next clock phase.

Parameters: uint8/uint16: txData: The data value to send across the SPI

Return Value: None

Side Effects: Required for modes where CPHA == 0 where data must be in the shift register before the first clock edge. Firmware must control this if there is already data being shifted out and if there is more data in the FIFO. This routine should not to be used for modes where CPHA == 1.

uint8/uint16 SPIS_ReadRxData(void)

Description: Reads the next byte of data received across the SPI.

Parameters: None

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

Side Effects: Will return invalid data if the FIFO is empty. Call SPIS_GetRxBufferSize() and if it returns a nonzero value then it is safe to call the SPIS_ReadRxData() function.

uint8 SPIS_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 SPIS_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 SPIS_ClearRxBuffer(void)

Description: Clears the Rx buffer memory array of data waiting to transmit. 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 transmit. Thus, writing will resume 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 will be lost when overwritten by new data.

void SPIS_ClearTxBuffer(void)

Description: Clears the memory array of all transmit data. Clears the Tx RAM buffer by setting the read and write pointers both to zero. Setting the pointers to zero makes the system believe there is no data to read, Thus, writing will resume 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 will be lost when overwritten by new data.

void SPIS_TxEnable(void)

Description: If the SPI Slave is configured to use a single bidirectional pin, this will set the bidirectional pin to transmit.

Parameters: None

Return Value: None

Side Effects: None

void SPIS_TxDisable(void)

Description: If the SPI Slave is configured to use a single bidirectional pin, this will set the bidirectional pin to receive.

Parameters: None

Return Value: None

Side Effects: None

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

Description: Writes available data from RAM/ROM to the Tx buffer while space is available. Keep trying until all data is passed to the Tx buffer. If using modes where CPHA = 0, call the SPIS_WriteTxDataZero() function before calling the SPIS_PutArray() function.

Parameters: uint8/uint16 *buffer: Pointer to the location in RAM containing the data to be sent

uint8 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 Slave and the Tx buffer is full.

void SPIS_ClearFIFO(void)

Description: Clears any received data from the Rx FIFO.

Parameters: None

Return Value: None

Side Effects: Clears the status register of the component.

void SPIS_Sleep(void)

Description: This is the preferred routine to prepare the component for low-power modes. The SPIS_Sleep() routine saves the current component state. Then it calls the SPIS_Stop() function and calls SPIS_SaveConfig() to save the hardware configuration.

Call the SPIS_Sleep() function before calling the CyPmSleep() or the CyPmHibernate() function. Refer to the PSoC Creator *System Reference Guide* for more information about power management functions.

Parameters: None

Return Value: None

Side Effects: None

void SPIS_Wakeup(void)

Description: This is the preferred routine to restore the component to the state when SPIS_Sleep() was called. The SPIS_Wakeup() function calls the SPIS_RestoreConfig() function to restore the configuration. If the component was enabled before the SPIS_Sleep() function was called, the SPIS_Wakeup() function will also re-enable the component. Clears all data from Rx buffer, Tx buffer, and hardware FIFOs.

Parameters: None

Return Value: None

Side Effects: Calling the SPIS_Wakeup() function without first calling the SPIS_Sleep() or SPIS_SaveConfig() function may produce unexpected behavior.

void SPIS_Init(void)

Description: Initializes or restores the component according to the customizer Configure dialog settings. It is not necessary to call SPIS_Init() because the SPIS_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 SPIS_Enable(void)

Description: Enables SPIS to start operation. Starts the internal clock if so configured. If an external clock is configured it must be started separately before calling this API. The SPIS_Enable() function should be called before SPIS 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 so configured. A SPIS_Init() function must have been previously called.

Parameters: None

Return Value: None

Side Effects: None

void SPIS_SaveConfig(void)

Description: This function saves the component configuration and nonretention registers. It also saves the current component parameter values, as defined in the Configure dialog or as modified by appropriate APIs. This function is called by the SPIS_Sleep() function.

Parameters: None

Return Value: None

Side Effects: None

void SPIS_RestoreConfig(void)

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

Parameters: None

Return Value: None

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

```
SPIS_STATUS_MASK_REG
SPIS_COUNTER_PERIOD_REG
```

Defines

SPIS_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. Refer to the Status Register Bits section for bit-field details.

SPIS_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. Refer to the Status Register Bits section for bit-field details.

Status Register Bits

SPIS_TXSTATUS

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

SPIS_RXSTATUS

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

- Byte/Word Complete: Set when a byte/word 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 Full: Set when the Rx Data FIFO is full (does not indicate the Rx buffer RAM array conditions).
- Rx FIFO Empty: Set when the Rx Data FIFO is empty (does not indicate the Rx buffer RAM array conditions).
- 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).
- Tx FIFO Empty: Set when the Tx Data FIFO is empty (does not indicate the Tx 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).
- 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.)

SPIS_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 automatically move data to the FIFO from the circular memory buffer.

SPIS_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.

SPIS_DATAWIDTH

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

Bootloader Support

The SPI Slave component can be used as a communication component for the Bootloader. Use the following configuration to support communication protocol from an external system to the Bootloader:

- **Mode**: Must match Host (boot device) data rate.
- **Data Lines**: MOSI + MISO
- **Data bits**: 8
- **Shift Direction**: Must match Host (boot device) data rate.
- **Bit Rate**: Must match Host (boot device) data rate.
- **RX Buffer Size**: 64
- **TX Buffer Size**: 64

For more information about the Bootloader, refer to the “Bootloader System” section of the *System Reference Guide*.

The SPI Slave Component provides a set of API functions for Bootloader use.

Function	Description
SPIS_CyBtldrCommStart	Starts the SPIS component and enables its interrupt.
SPIS_CyBtldrCommStop	Disables the SPIS component and disables its interrupt.
SPIS_CyBtldrCommReset	Resets the receive and transmit communication buffers.
SPIS_CyBtldrCommRead	Allows the caller to read data from the bootloader host. This function manages polling to allow a block of data to be completely received from the host device.

Function	Description
SPIS_CyBtldrCommWrite	Allows the caller to write data to the boot loader host. This function uses a blocking write function for writing data using SPIS communication component.

void SPIS_CyBtldrCommStart(void)

Description: Starts the SPIS communication component.

Parameters: None

Return Value: None

Side Effects: This component automatically enables global interrupt.

void SPIS_CyBtldrCommStop(void)

Description: This function disables the SPIS component and disables its interrupt.

Parameters: None

Return Value: None

Side Effects: None

void SPIS_CyBtldrCommReset(void)

Description: Resets the receive and transmit communication Buffers.

Parameters: None

Return Value: None

Side Effects: None

cystatus SPIS_CyBtldrCommRead(uint8 pData[], uint16 size, uint16 * count, uint8 timeOut)

Description: This function allows the caller to read data from the bootloader host. The function manages polling to allow a block of data to be completely received from the bootloader host.

Parameters: uint8 pData[]: Pointer to storage for the block of data to be read from the bootloader host

uint16 size: Number of bytes to be read

uint16 *count: Pointer to the variable to write the number of bytes actually read

uint8 timeOut: Number of units in 10 ms to wait before returning because of a timeout

Return Value: cystatus: Returns CYRET_SUCCESS if no problem was encountered or returns the value that best describes the problem. For more information, see the “Return Codes” section of the *System Reference Guide*.

Side Effects: None

cystatus SPIS_CyBtldrCommWrite(const uint8 pData[], uint16 size, uint16 * count, uint8 timeOut)

Description: Allows the caller to write data to the boot loader host. This function uses a blocking write function for writing data using SPIS communication component.

Parameters: const uint8 pData[]: Pointer to the block of data to be written to the bootloader host

uint16 size: Number of bytes to be written

uint16 *count: Pointer to the variable to write the number of bytes actually written

uint8 timeOut: Number of units in 10 ms to wait before returning because of a timeout

Return Value: cystatus: Returns CYRET_SUCCESS if no problem was encountered or returns the value that best describes the problem. For more information see the “Return Codes” section of the *System Reference Guide*.

Side Effects: None

MISRA Compliance

This section describes the MISRA-C:2004 compliance and deviations for the component. There are two types of deviations defined:

- project deviations – deviations that are applicable for all PSoC Creator components
- specific deviations – deviations that are applicable only for this component

This section provides information on component-specific deviations. Project deviations are described in the MISRA Compliance section of the *System Reference Guide* along with information on the MISRA compliance verification environment.

The SPI Slave component has not been verified for MISRA-C:2004 coding guidelines compliance.

Sample Firmware Source Code

PSoC Creator provides many 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 SPIS is as an 8-bit SPIS with (CPHA = 0, CPOL = 0) configuration.

Modes

Modes control the component’s status bits and the component signal values that are assumed during data transmission. Four waveforms are shown. It is assumed that five data bytes are transmitted (four bytes are written to the SPI Slave’s Tx buffer at the beginning of transmission and a fifth is thrown after the first byte has been loaded into the A0 register). Numbers in circles represent the following events (see the following waveforms):

1 – Tx FIFO Empty has being cleared when four bytes are written to the Tx buffer;

2 – Tx FIFO Not Full has been cleared because Tx FIFO is full after 4 bytes written;

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

4 – The Slave Select line is set to Low, indicating the beginning of the transmission.

5 – Tx FIFO Not Full status is set when the second bit is loaded to the A0 register. Rx Not Empty status is set when the first received byte has been loaded into the Rx buffer. Byte/Word Complete is set as well.

6 – Tx FIFO Empty status is set at the moment when the last byte to be sent has been loaded into the A0 register. This is not shown in the waveform details for simplification.

7 – The moment when the fourth byte has been received so Rx FIFO Full is set along with Byte/Word Complete.

8 – 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.

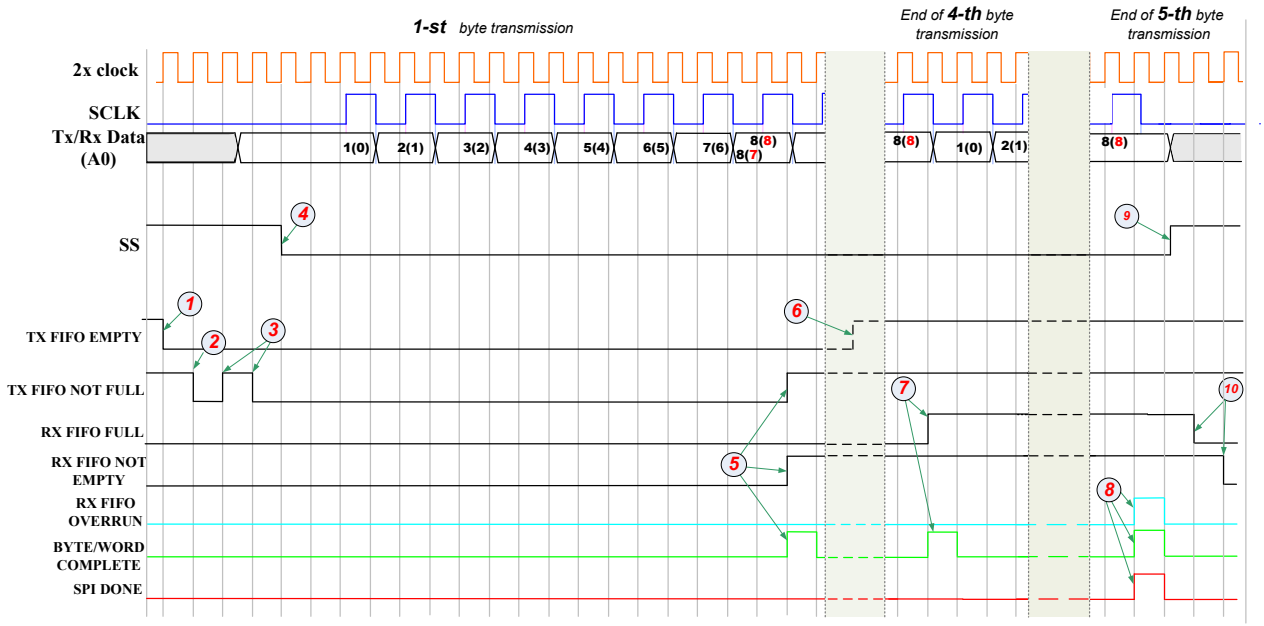
9 – The SS line is set to High to indicate that transmission is complete.

10 – 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.

Note Because the same register is used to transmit and receive data, the diagram section "Tx/Rx Data (A0)" contains two bit numbers in the following format: "Tx bit number (Rx bit number)."

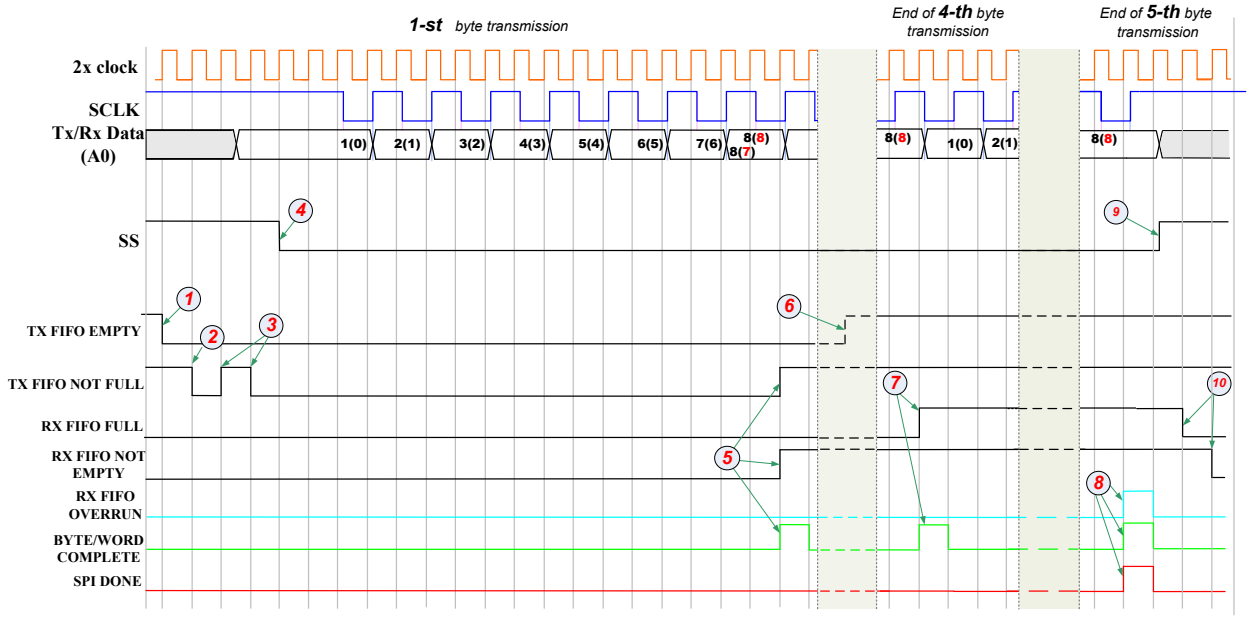
SPIS Mode: (CPHA == 0, CPOL == 0)

Mode 0 has the following characteristics:



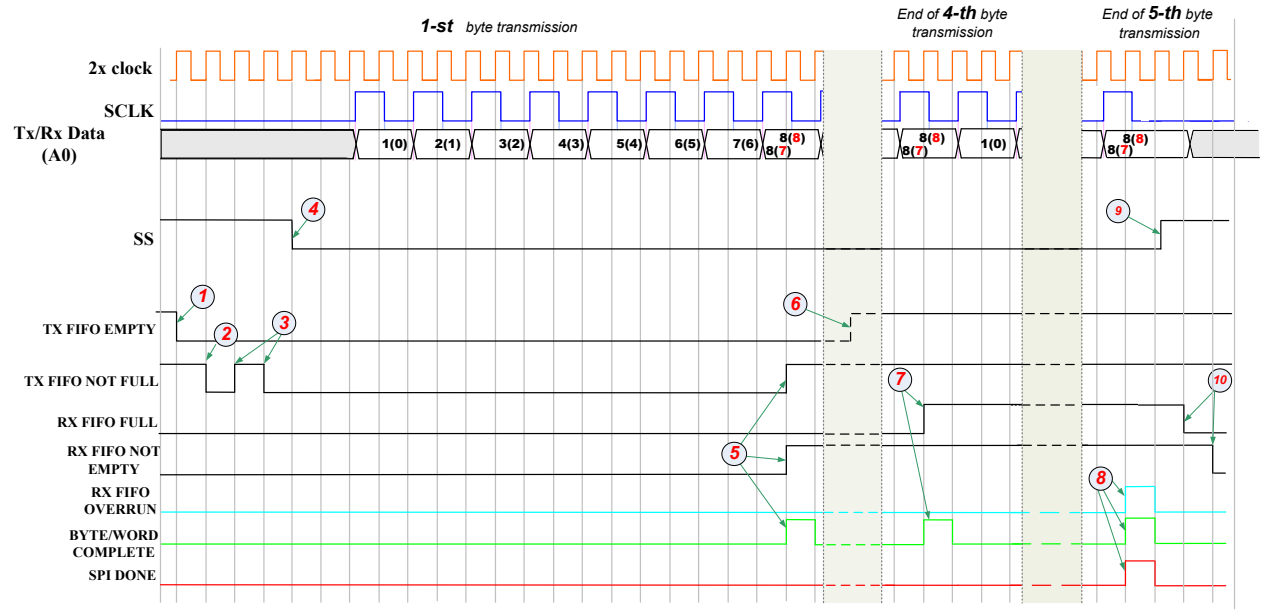
SPIS Mode: (CPHA == 0, CPOL == 1)

Mode 1 has the following characteristics:



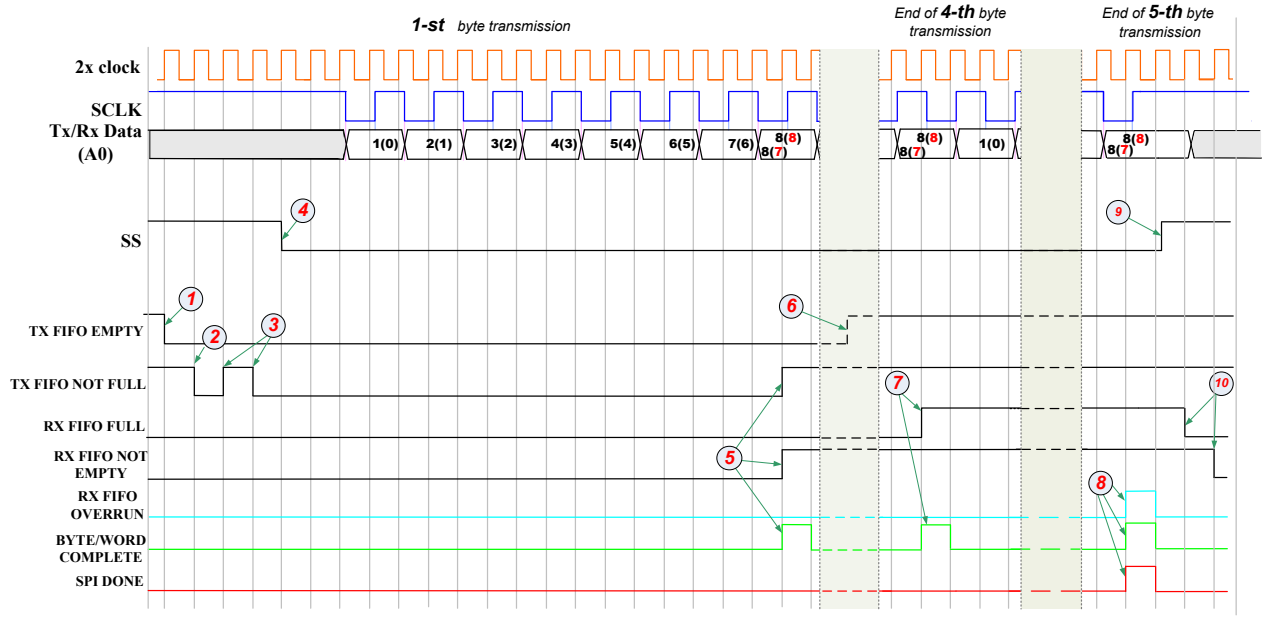
SPIS Mode: (CPHA == 1, CPOL == 0)

Mode 2 has the following characteristics:



SPIS Mode: (CPHA == 1, CPOL == 1)

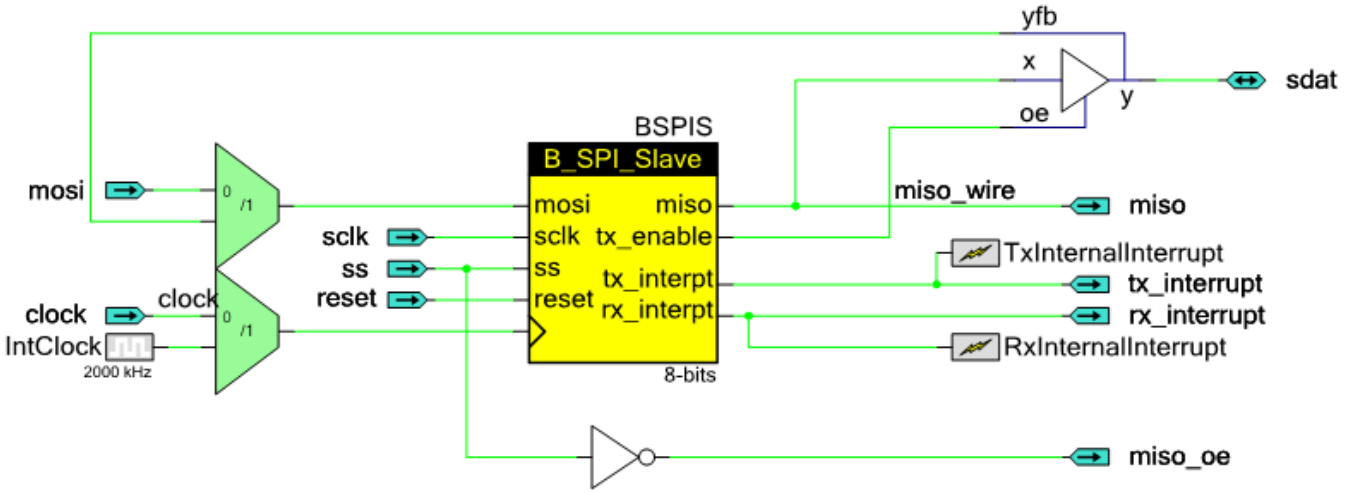
Mode 3 has the following characteristics:



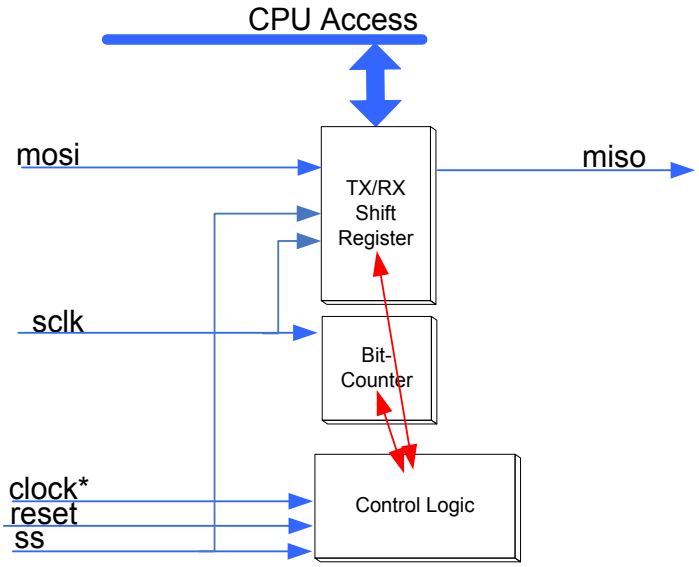
Note Some SPI Master devices (such as the TotalPhase Aardvark I2C/SPI host adapter) drive the sclk output in a specific way. For the SPI Slave component to function properly with such devices in where CPOL = 1), the sclk pin should be set to resistive pull-up drive mode. Otherwise, it gives out corrupted data.

Block Diagram and Configuration

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



The implementation is described in the following block diagram.



Registers

Tx Status

The Tx status register is a read-only register that contains the various status bits defined for a given instance of the SPIS component. Assuming that an instance of the SPI Slave is named “SPIS,” the value of these registers is available from the SPIS_ReadTxStatus() function call. The interrupt output signal is generated from an ORing of the masked bit-fields within the Tx status register. You can set the mask using the SPIS_SetTxInterruptMode() function call. Upon receiving an interrupt you can retrieve the interrupt source by reading the Tx Status register with the SPIS_ReadTxStatus() function call. The Tx Status register is cleared on reading so the interrupt source is held until the SPIS_ReadTxStatus() function is called. All operations on the Tx status register must use the following defines for the bit-fields, as these bit-fields may be moved within the Tx status register at build time.

There are several bit-field masks defined for the Tx status register. Any of these bit-fields may be included as an interrupt source. The bit fields indicated with an asterisk (*) are configured as sticky bits in the TX status register; all other bits are configured as real-time indicators of status.

The #defines are available in the generated header file (.h) as follows:

- SPIS_STS_SPI_DONE * – Defined as the bit-mask of the Status register bit SPI Done.
- SPIS_STS_TX_FIFO_NOT_FULL – Defined as the bit-mask of the Status register bit Transmit FIFO Empty.

- SPIS_STS_TX_FIFO_EMPTY – Defined as the bit-mask of the Status register bit Transmit FIFO Empty.
- SPIS_STS_BYTE_COMPLETE * – Defined as the bit-mask of the Status register bit Byte Complete.

Rx Status

The Rx status register is a read-only register that contains the various status bits defined for the SPIS. The value of these registers is available from the SPIS_ReadRxStatus() function call. The interrupt output signal is generated from an ORing of the masked bit-fields within the Rx status register. You can set the mask using the SPIS_SetRxInterruptMode() function call. Upon receiving an interrupt you can retrieve the interrupt source by reading the Rx Status register with the SPIS_ReadRxStatus() function call. The Rx Status register is cleared on read so the interrupt source is held until the SPIS_ReadRxStatus() function is called. All operations on the Rx status register must use the following defines for the bit-fields as these bit-fields may be moved within the Rx status register at build time.

There are several bit-field masks defined for the Rx status register. Any of these bit-fields may be included as an interrupt source. The bit-fields indicated with an asterisk (*) are configured as sticky bits in the Rx status register; all other bits are configured as real-time indicators of status.

The #defines are available in the generated header file (.h) as follows:

- SPIS_STS_RX_FIFO_FULL – Defined as the bit-mask of the Status register bit Receive FIFO Full.
- SPIS_STS_RX_FIFO_NOT_EMPTY – Defined as the bit-mask of the Status register bit Receive FIFO Not Empty.
- SPIS_STS_RX_FIFO_OVERRUN * – Defined as the bit-mask of the Status register bit Receive FIFO Overrun.

Tx Data

The Tx data register contains the transmit data value to send. This is implemented as a FIFO in the SPIS. There is a software state machine to control data from the transmit memory buffer to handle much larger portions of data to send. All APIs dealing with transmitting the 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 set up to fill this FIFO when empty, using the TXDATA_REG address defined in the header file.

Rx Data

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

Conditional Compilation Information

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

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

Resources

The SPI Slave 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	3	1	–	2
8-bit (Bidirectional)	1	12	3	2	–	2
16-bit (MOSI+MISO)	2	12	3	1	–	2
16-bit (Bidirectional)	2	12	3	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	616	5	528	5
8-bit (Bidirectional)	429	3	616	5	528	5
16-bit (MOSI+MISO)	503	3	656	9	568	5
16-bit (Bidirectional)	503	3	656	9	568	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[2]	Max	Units[3]
$I_{DD(8\text{-bit (MOSI+MISO)}}$	Component current consumption				
	Idle current[3]	–	18	–	µA/MHz
	Operating current[4]	–	27	–	µA/MHz
$I_{DD(8\text{-bit (Bidirectional)}}$	Component current consumption				
	Idle current[3]	–	20	–	µA/MHz
	Operating current[4]	–	32	–	µA/MHz
$I_{DD(16\text{-bit (MOSI+MISO)}}$	Component current consumption				
	Idle current[3]	–	32	–	µA/MHz

[2]. Device IO and clock distribution current not included. The values are at 25 °C.

[3]. Current consumption is specified with respect to the incoming component clock.

Parameter	Description	Min	Typ[2]	Max	Units[3]
	Operating current[4]	–	38	–	µA/MHz
$I_{DD(16\text{-bit}}$ (Bidirectional)	Component current consumption				
	Idle current[3]	–	33	–	µA/MHz
	Operating current[4]	–	40	–	µA/MHz

AC Characteristics

Parameter	Description	Config	Min	Typ	Max [4]	Units
f_{SCLK}	SCLK frequency	Config 1 [5]	–	–	5	MHz
		Config 2 [6]	–	–	5	MHz
		Config 3 [7]	–	–	4	MHz
		Config 4 [8]	–	–	4	MHz
f_{CLOCK}	Component clock frequency [9]	Config 1 [5]	2 * f_{SCLK}	–	10	MHz
		Config 2 [6]	2 * f_{SCLK}	–	10	MHz
		Config 3 [7]	2 * f_{SCLK}	–	8	MHz
		Config 4 [8]	2 * f_{SCLK}	–	8	MHz
t_{CKH}	SCLK High time		–	0.5	–	$1/f_{SCLK}$
t_{CKL}	SCLK Low time		–	0.5	–	$1/f_{SCLK}$

[4] The component 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 need to validate the timing requirements with STA results.

[5] Config 1 options:
Data Lines: MOSI+MISO
Data Bits: 8

[6] Config 2 options:
Data Lines: MOSI+MISO
Data Bits: 16

[7] Config 3 options:
Data Lines: Bidirectional
Data Bits: 8

[8] Config 4 options:
Data Lines: Bidirectional
Data Bits: 16

[9] Component clock is for status register only; it does not affect base functionality or bit-rate. Routing may limit the maximum frequency of this parameter; therefore, the maximum is listed with nominal routing results.

Parameter	Description	Config	Min	Typ	Max [4]	Units
t_{SCLK_MISO}	SCLK to MISO output time		–	–	52	ns
t_{SCLK_SDAT} (Bidirectional Mode only)	SCLK to SDAT output time		–	–	54	ns
t_{S_MOSI}	MOSI input setup time		25	–	–	ns
t_{H_MOSI}	MOSI input hold time		–	0		ns
t_{SS_SCLK}	SS active to SCLK active		20	–	–	ns
t_{SCLK_SS}	SCLK inactive to SS inactive		–20	–	20	ns

Figure 6. Mode CPHA = 0 Timing Diagram

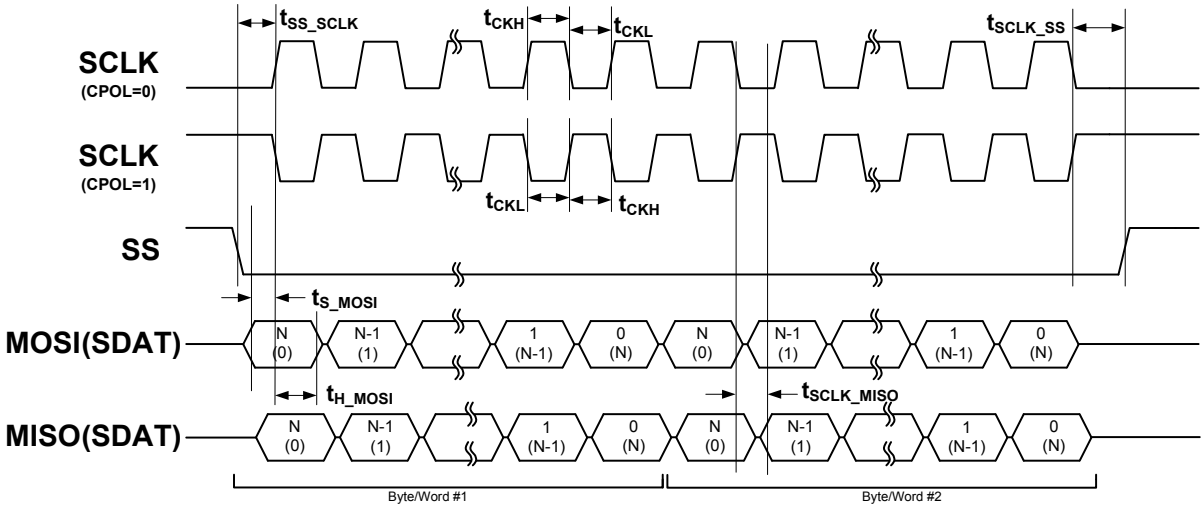
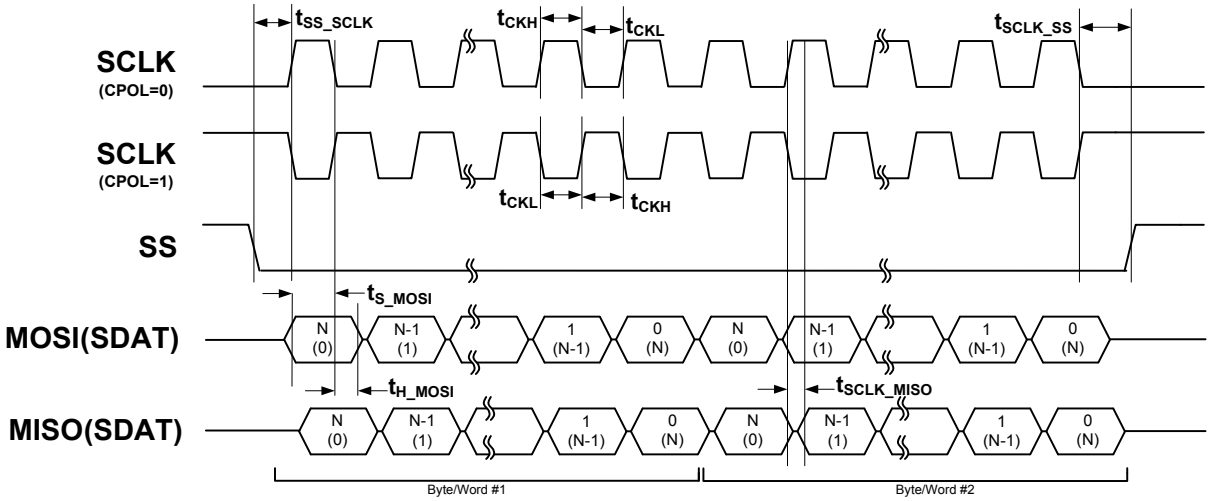


Figure 7. Mode CPHA = 1 Timing Diagram

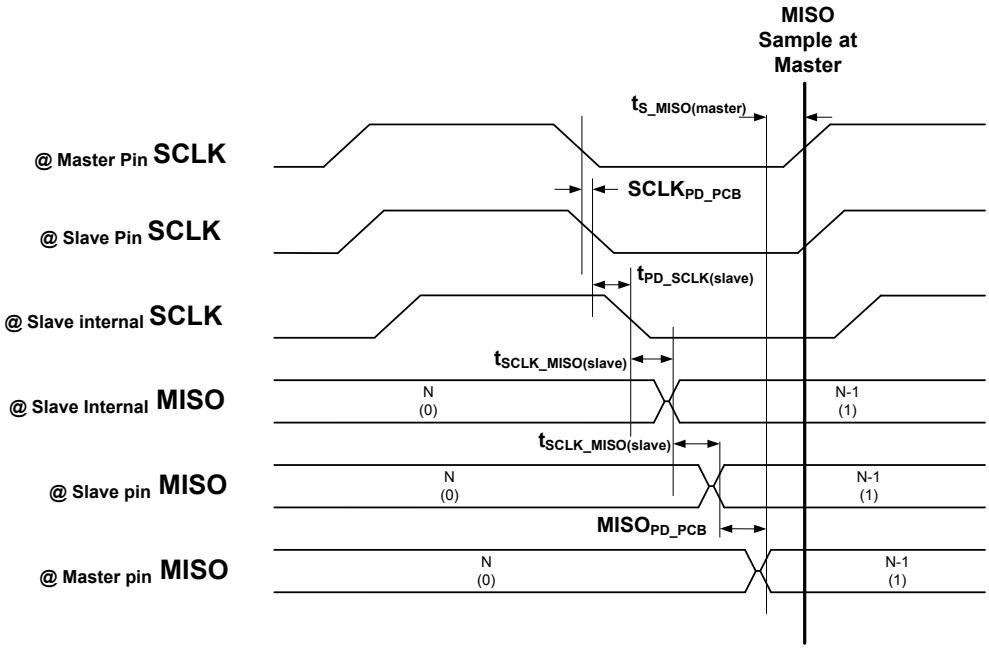


How to Use STA Results for Characteristics Data

Nominal route maximums are gathered through multiple test passes with Static Timing Analysis (STA). The maximums can be calculated 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 take several factors into account. Board layout and slave communication device specs are needed 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 8. Calculating Maximum f_{SCLK} Frequency



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

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

OR

$$f_{SCLK} < 1 \div \{2 \times [T_{RT_PD} + t_{PD_SCLK(master)} + t_{S_MISO(master)}] \}$$

Where $t_{PD_SCLK(master)} + t_{S_MISO(Master)}$ must come from the Master Device datasheet. t_{RT_PD} is defined as:

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

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.

$t_{PD_SCLK(Slave)}$ is the path delay of the input SCLK to the internal logic; $t_{SCLK_MISO(slave)}$ is the SCLK pin to internal logic path delay of the slave component; and $t_{PD_MISO(slave)}$ is the path delay of the internal MISO to the pin. The easiest way to find the values for these three parameters is to take the combined path as directly listed in the STA results, shown in the figure below:

- **Clock To Output Section**

  - **SCLK_1(0)_PAD**

Source	Destination	Delay (ns)
\SPIS_1:BSPIS:es3:SPISlave:sR8:Dp:u0\/so_comb	MISO_1(0)_PAD	47.895

Where $t_{PD_SCLK(Slave)}$ is the first two numbers, $t_{SCLK_MISO(slave)}$ is the second two numbers, and $t_{PD_MISO(slave)}$ is the last two numbers. The full path of the three parameters is 45.889 ns.

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

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

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

f_{CLOCK} Maximum component clock frequency is provided in the 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 below:

- **Clock Summary Section**

Clock	Type	Nominal Frequency (MHz)	Required Frequency (MHz)	Maximum Frequency (MHz)	Violation
BUS_CLK	Sync	24.000	24.000	N/A	
ClockBlock/clk_bus	Async	24.000	24.000	N/A	
ClockBlock/dclk_0	Async	2.000	2.000	N/A	
ILO	Async	0.001	0.001	N/A	
IMO	Async	3.000	3.000	N/A	
MASTER_CLK	Sync	24.000	24.000	N/A	
PLL_OUT	Async	24.000	24.000	N/A	
SCLK_1(0)_PAD	Async	UNKNOWN	UNKNOWN	33.636	
SPIS_1_IntClock	Sync	2.000	2.000	75.746	

t_{CKH} The SPI Slave component requires a 50-percent duty cycle SCLK.

t_{CKL} The SPI Slave component requires a 50-percent duty cycle SCLK.

t_{S_MOSI} To meet the setup time of the internal logic, MOSI must be valid at the pin, before SCLK is valid at the pin, by this amount of time.

t_{H_MOSI} To meet the hold time of the internal logic, MOSI must be valid at the pin, after SCLK 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 Errata

This section lists known problems with the component.

Cypress ID	Component Version	Problem	Workaround
191257	2.40	This component version was modified without a version number change in PSoC Creator 3.0 SP1. For more information, see Knowledge Base Article KBA94159 (www.cypress.com/go/kba94159).	There is no workaround. You must update to the latest version of the component.

Component Changes

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

Version	Description of Changes	Reason for Changes / Impact
2.41	Corrected the component changes made in PSoC Creator 3.0 SP1.	Correction of the Component Errata item – Cypress ID 191257.
2.40	Added MISRA Compliance section.	The component was not verified for MISRA compliance.
	Integrated specific APIs to support the bootloader: CyBtldrCommStart, CyBtldrCommStop, CyBtldrCommReset, CyBtldrCommWrite, CyBtldrCommRead.	SPI slave could be used as a communication component for the Bootloader with this feature.
	Changed tSS_SCLK timing parameter value. Min value is set to 20 ns. Max value is removed as not applicable.	Previous timing values were incorrect.
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.

Version	Description of Changes	Reason for Changes / Impact
	Added PSoC 5LP support	
	Added DC characteristics section to datasheet	
2.20.a	Added fixed placement explanation to datasheet	
2.20	The SPI slave now discards any partial word that has been received and any partial word that has been transferred, if during the transfer the SS pin goes high. Defect appeared on ES3 silicon only.	Verilog defect fixed.
	Added Enable Fixed Placement option to customizer	
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
	“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.	
	Minor datasheet edits and updates	
2.0	Added SPIS_Sleep()/SPIS_Wakeup() and SPIS_Init()/SPIS_Enable() APIs.	To support low power modes, as well as to provide common interfaces to separate control of initialization and enabling of most components.
	Number and positions of component outputs have been changed: ▪ The reset input was added; ▪ The interrupt output was removed; rx_interrupt, tx_interrupt outputs were added instead.	Production PSoC 3 reset functionality was added. Two status interrupt registers (Tx and Rx) are now presented instead of one shared. The changes must be taken into account to prevent binding errors when migrating from previous SPI versions
	Removed SPIS_EnableInt(), SPIS_DisableInt(), SPIS_SetInterruptMode(), and SPIS_ReadStatus() APIs. Added SPIS_EnableTxInt(), SPIS_EnableRxInt(), SPIS_DisableTxInt(), SPIS_DisableRxInt(), SPIS_SetTxInterruptMode(), SPIS_SetRxInterruptMode(), SPIS_ReadTxStatus(), SPIS_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 SPIS_ReadByte(), SPIS_WriteByte(), and SPIS_WriteByteZero() APIs to SPIS_ReadRxData(), SPIS_WriteTxData(), SPIS_WriteTxDataZero().	Clarifies the APIs and how they should be used.

Version	Description of Changes	Reason for Changes / Impact
The following changes were made to the base SPI Slave component B_SPI_Slave_v2_0, which is implemented using Verilog:		
	B_SPI_Slave_v2_0 now contains two separate implementations for ES2 and ES3 silicon. One datapath is used for 8-bit SPI for Tx and Rx in ES3 silicon instead of two in ES2.	Requirement that all components support ES2 and ES3 silicon. Use of ES3 feature updates is a requirement and this helps optimize resource usage in ES3. .
	Changes in ES2 support implementation: Two status registers are now presented (status are separate on Tx and Rx) instead of using one common status register for both.	This provides correct software buffer functionality.
	'BidirectMode' boolean parameter is added to base component (Verilog implementation). Control Register with 'clock' input and SYNC mode bit is now selected to drive 'tx_enable' output for ES3 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.	Added Bidirectional Mode support to the component
	Four udb_clock_enable components are added to Verilog implementation with sync = 'TRUE' parameter. One of them has sync = 'TRUE' (status register clock), other three have sync = 'FALSE'	New requirements for all clocks used in Verilog to indicate functionality so the tool can support synchronization and Static Timing Analysis.
	Rx datapath configuration changed. 'FIFO FAST' option is set to 'DP'' instead of 'BUS'	Fixes a defect in previous versions of the component where there was a timing window affecting correct component capture of data.
	Additional logic to Verilog implementation to change the moments when SPI DONE and BYTE COMPLETE status are generated.	Fixes a defect in previous versions of the component where SPI DONE is sometimes not generated even after communication is complete.
	Maximum Bit Rate value is changed to 10 Mbps	Bit Rate value more than 10 Mbps is not supported (verified during 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

Version	Description of Changes	Reason for Changes / Impact
	Sample firmware source code is removed	Reference to the component example project is added instead
	SPI Modes diagrams were changed (Tx and Rx FIFO status values were added)	Data sheet defect fixed

© Cypress Semiconductor Corporation, 2014. 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.