SPI Master Datasheet SPIM V 2.6

Copyright © 2002-2014 Cypress Semiconductor Corporation. All Rights Reserved.

Resources	PSoC® Blocks			API Memory (Bytes)		Pins (per External I/O)
	Digital	Analog CT	Analog SC	Flash	RAM	
CY8C29/27/24/22/21xxx, CY7C603xx, CY7C64215, CYWUSB6953, CY8C23x33, CY8CLED02/04/08/16, CY8CLED0xD, CY8CLED0xG, CY8CTST110, CY8CTMG110, CY8CTST120, CY8CTMG120, CY8CTMA120, CY8CTMA140, CY8C21x45, CY8C22x45, CY8CTMA30xx, CY8C28x45, CY8CPLC20, CY8CLED16P01, CYRF69xx3, CY8C28xxx						
	1	0	0	27	0	3 - 4

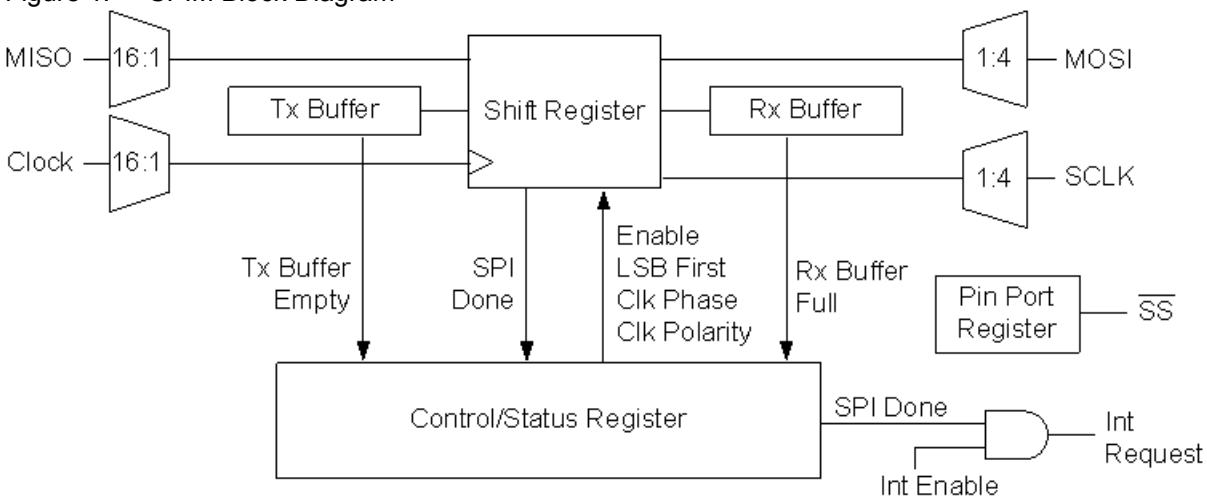
Features and Overview

- Supports Serial Peripheral Interconnect (SPI) Master protocol
- Supports SPI clocking modes 0, 1, 2, and 3
- Selectable input sources for clock and MISO
- Selectable output routing for MOSI and SCLK
- Programmable interrupt on SPI done condition
- SPI Slave devices can be independently selected

For more details on how SPI works in PSoC 1 devices, read the application note AN51234 - Getting Started with SPI in PSoC® 1.

The SPIM User Module is a Serial Peripheral Interconnect Master. It performs full duplex synchronous 8-bit data transfers. SCLK phase, SCLK polarity, and LSB First can be specified to accommodate most SPI clocking modes. Controlled by user-supplied software, the slave select signal can be configured to control one or more SPI Slave devices. The SPIM PSoC block has selectable routing for the input and output signals, and programmable interrupt-driven control.

Figure 1. SPIM Block Diagram



Functional Description

SPIM is a user module that implements a Serial Peripheral Interconnect Master. It uses the Tx Buffer, Rx Buffer, Control, and Shift registers of a Digital Communications Type PSoC block and one or more Pin Port registers.

The Control register is initialized and configured using the Device Editor and/or the SPIM User Module firmware Application Programming Interface (API) routines. Initialization includes setting the LSB First configuration and the SPI transmission/receive clocking modes. SPI modes 0, 1, 2, and 3 are supported. Both the SPI master and slaves must be set with the same clock mode and bit configuration in order to properly communicate. The SPI modes are defined as listed in Table 1:

Table 1. SPI Modes

Mode	SCLK Edge Performing Data Latch	Clock Polarity	Notes
0	Leading	Non-inverting	Leading edge latches data. Data changes on trailing edge of clock.
1	Leading	Inverted	
2	Trailing	Non-inverting	Trailing edge latches data. Data changes on leading edge.
3	Trailing	Inverted	

Note Check to ensure that the clock polarity and phase for the selected mode match the settings used for any attached SPI devices, as mode numbers may not be the same for all devices.

The active low slave select signal(s), ~SS, must be set with user-supplied software routines to control selected Pin Port register bits, enabling the SPI Slave devices properly. One or more slave select signals can be configured, but only one slave select signal can be active at a time.

To accommodate all of the SPI clocking modes, the slave select signal should be asserted and de-asserted for each byte of data transmitted from the SPI Master to the selected SPI Slave device. However, this is not a strict requirement, since there are variations in the specific byte transport protocols used for SPI communications between SPI Master and Slave devices.

The SCLK signal is the SPI transmit/receive clock. It is one-half the clock rate of the input clock signal. The effective transmit/receive bit rate is the input clock divided by two. The input clock is specified using the Device Editor.

The MOSI signal is the Master-Out-Slave-In data signal that transmits the data from the master to a slave SPI protocol-compliant device. The MISO signal is the Master-In-Slave-Out data signal that transmits the data from the slave SPI device to this user module.

The SPIM hardware transmits data from the master SPI device on the MOSI signal and simultaneously receives data from the selected slave SPI device on the MISO signal. The same SCLK signal is used for both transmit and receive of the master and slave data.

The SPI protocol is a master-only initiated response protocol. It is the master’s responsibility to determine that the selected slave device is ready for a command or has data ready to be received.

The SPIM User Module is enabled for operation when the SPI enable bit is set in the Control register using an API routine. At this time, all slave select signals should be asserted high.

Before transmitting a byte to a selected SPI Slave device, the specified slave select signal should be asserted low.

The data byte to be transmitted is written to the Tx Buffer register. This clears the Tx Buffer Empty status bit in the Control register. On the next clock, the data in the Tx Buffer is transferred to the Shift register and the Tx Buffer Empty status bit is set. The Buffer Empty status bit should be checked before writing a byte to the Tx Buffer register, to prevent a transmit overrun condition. Another data byte to transmit can be written (preloaded), to the Tx Buffer register at this time. Upon completion of the transmission of the current byte, this data is transmitted without delay on the next SCLK signal.

For each bit of the data byte in the Shift register, the SCLK output signal is generated, the data in the Shift register is shifted to the MOSI output, and the input data from the slave SPI is shifted into the Shift register from the MISO input. The specific timing of the SCLK, MOSI, and MISO signals is based on the SPI clock mode configuration.

After all of the bits have been transmitted and simultaneously received, the received data is transferred from the Shift register to the Rx Buffer register, and the Tx Buffer register is transferred to the Shift register. The Rx Buffer Full and the SPI Done status bits are set. The SPI Done status bit causes an interrupt to trigger, if interrupts are enabled.

If the SPI Done interrupt condition is not used to retrieve the data byte from the Rx Buffer register, the Control register should be polled to monitor the Rx Buffer Full status bit. The received data must be read from the Rx Buffer register before the next data byte is fully received or the Overrun Error status bit is set.

If a pending byte of data has been preloaded into the Tx Buffer register, then this byte restarts the SPI state machine and transmission commences immediately.

Two methods can be deployed to control the slave select signal after each data byte has been transmitted: non-interrupt or interrupt level. The selection of which method to employ should depend on the SPI clocking mode, the byte transport protocol, and the required bit-rate speed of the transmitted data. SPI clocking mode 0 and 1 require that the slave select be toggled off and then on again before the next byte of data is transmitted. The byte transport protocol used for multi-byte transmissions to the same slave device may or may not require that the slave select be toggled. Toggling the slave select may also have a slave device ramification, in that a set up time may be required to allow the slave device to assert its data on the MISO signal before the SCLK signal is initiated.

Controlling the slave select signal at noninterrupt level requires the microprocessor to be dedicated to monitoring or sampling the SPI Done status bit, while SPI data is being transmitted. If the data bit rate is substantially high, on the order of 1 MHz or faster, the overhead in monitoring the status bit is very limited.

For slower bit rates where the overhead in monitoring the SPI Done status bit may tie up the microprocessor from doing any other required operations, performing the slave select control can be done at interrupt level. There is a minimum interrupt latency of 833 ns (at a 24 MHz clock rate), plus the time it takes to exercise the instructions to manage the slave select.

Multi-byte data transfers, where the Tx Buffer is preloaded with data to transmit, may be a cause for concern. However, it may be possible to achieve a 1 MHz or better bit transfer rate if minimal or no inter-byte processing is required to manage the selected slave signal, and when multi-byte transmissions are performed to the same SPI Slave device.

When the final data is transmitted, the SPI Done bit should be monitored to determine when to disable the SPIM User Module. This guarantees that all of the clocking signals have completed between the master and slave SPI devices.

Timing

Typical timing for SPIM transfer is shown in the following diagrams. See the Technical Reference Manual for addition SPIM timing information.

Figure 2. Typical SPIM Timing for Modes 0 and 1

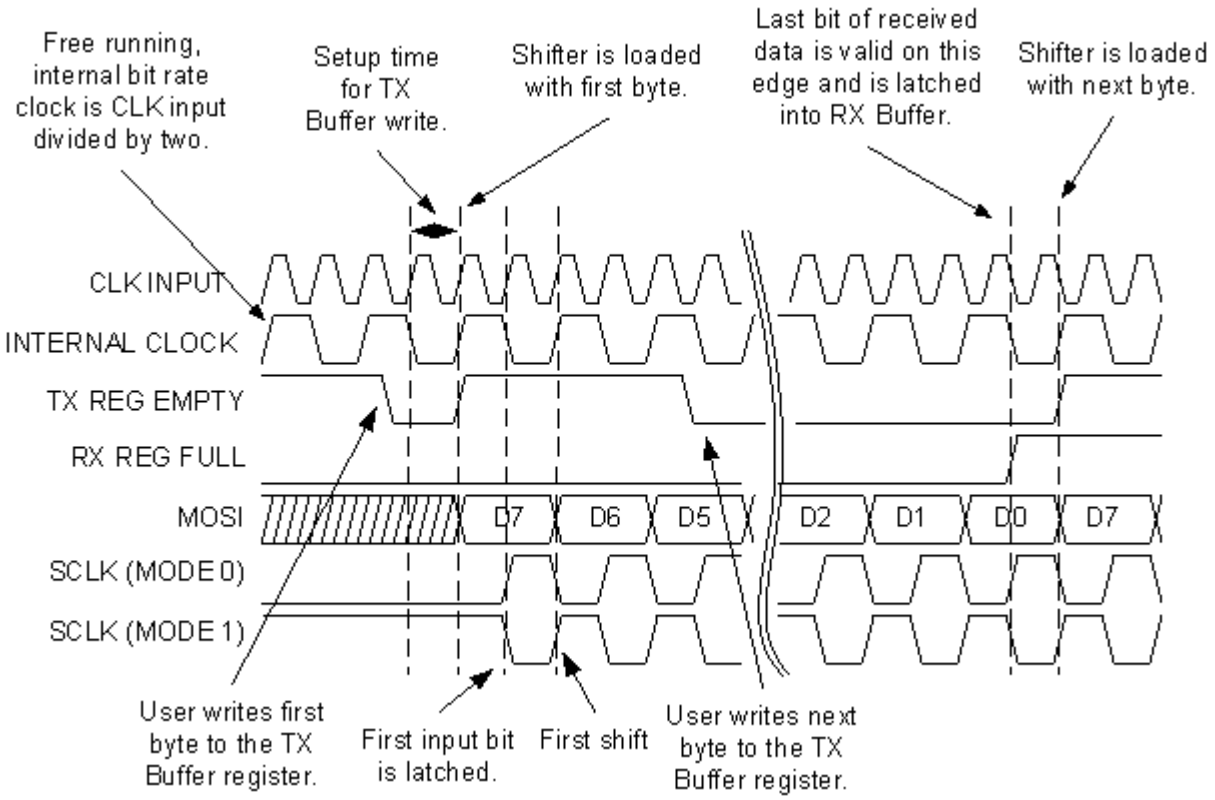
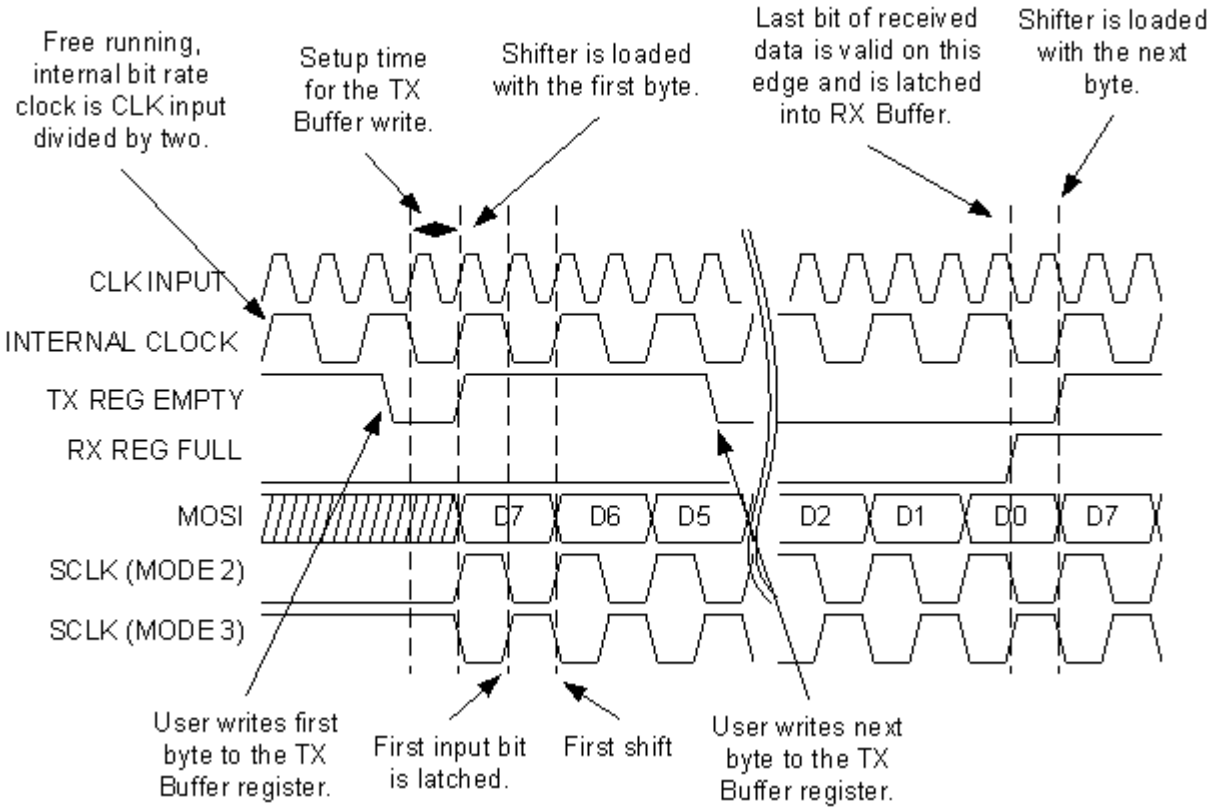


Figure 3. Typical SPIM Timing for Modes 2 and 3



DC and AC Electrical Characteristics

Table 2. SPIM DC and AC Electrical Characteristics

Parameter	Conditions and Notes	Typical	Limit	Units
F_{max}	Maximum bit rate	4.1[a]	--	Mbps

a. The Typical rate is true of many PSoC devices, but maximum bit rate varies by device. The maximum data rate is the maximum clock value divided by two. This can also be limited due to the interrupt latency of the device. See the device datasheet for information regarding the maximum values for your device.

Placement

SPIM maps onto a single PSoC block and may be placed in any of the Digital Communications blocks.

Pin Port register bit(s) needs to be reserved for use by the slave select signal(s) to control SPI Slave devices. These port bits should be configured as standard CPU port pins.

Parameters and Resources

Clock

SPIM is clocked by one of 16 possible sources. The global I/O busses may be used to connect the clock input to an external pin or a clock function generated by a different PSoC block. When using an external digital clock for the block, the row input synchronization should be turned off for best accuracy, and sleep operation. The 48 MHz clock, the CPU_32 kHz clock, one of the divided clocks (24V1 or 24V2), or another PSoC block output can be specified as the SPIM clock input.

The clock rate must be set to two times the desired bit rate. One data bit is transmitted and/or received for every two input clocks.

MISO

The Master-In-Slave-Out input signal can be routed to one of 16 possible input sources. Allowable MISO sources include high, low, the global I/O buses, analog comparator buses, or another PSoC block can be specified to supply the MISO input.

Note: The Row Input synchronization for MISO should be set to Async for High SPI data rates >1MHz.

MOSI

The Master-Out-Slave-In output of the SPIM can be routed to one of the global output buses. The global output bus can then be connected to an external pin or to another PSoC block for further processing.

SCLK

This output clock is generated by the SPI Master. It is normally routed out through one of the global output lines to a port pin, then connected to a SPI slave. This clock defines the effective bit transfer rate.

Interrupt Mode

This option determines when an interrupt is generated for the TX block. The “TxRegEmpty" option causes an interrupt to be generated as soon as the data has been transferred from the Data register to the Shift register. Choosing the second option, “TxComplete" delays the interrupt until the last bit is shifted out of the Shift register. This second option is useful in that it is important to know when the character has been completely sent. The first option, “TxRegEmpty" is best used to maximize the output of the transmitter. It allows a byte to be loaded while the previous byte is being sent.

ClockSync

In the PSoC devices, digital blocks may provide clock sources in addition to the system clocks. Digital clock sources may even be chained in ripple fashion. This introduces skew with respect to the system clocks. These skews are more critical in the CY8C29/27/24/22/21xxx and CY8CLED04/08/16 PSoC device families because of various data-path optimizations, particularly those applied to the system busses. This parameter may be used to control clock skew and ensure proper operation when reading and writing PSoC block register values. Appropriate values for this parameter should be determined from the following table.

ClockSync Value	Use
Sync to SysClk	Use this setting for any 24 MHz (SysClk) derived clock source that is divided by two or more. Examples include VC1, VC2, VC3 (when VC3 is driven by SysClk), 32KHz, and digital PSoC blocks with SysClk-based sources. Externally generated clock sources should also use this value to ensure that proper synchronization occurs.
Sync to SysClk*2	Use this setting for any 48 MHz (SysClk*2) based clock unless the resulting frequency is 48 MHz (in other words, when the product of all divisors is 1).
Use SysClk Direct	Use when a 24 MHz (SysClk/1) clock is desired. This does not actually perform synchronization but provides low-skew access to the system clock itself. If selected, this option overrides the setting of the Clock parameter, above. It should always be used instead of VC1, VC2, VC3 or Digital Blocks where the net result of all dividers in combination produces a 24 Mhz output.
Unsynchronized	Use when the 48 MHz (SysClk*2) input is selected. Use when unsynchronized inputs are desired. In general this use is advisable only when interrupt generation is the sole application of the Counter.

InvertMISO

This parameter gives the user the option to invert the MISO input.

Timing

The clock rate must be set to two times the desired bit rate.

The timing of the SPI data transmission must also account for the delay caused by processing the slave select signal. It is important to take into account the SPI Slave device’s select signal set up time.

Interrupt Generation Control

There are two additional parameters that become available when the **Enable interrupt generation control** check box in PSoC Designer is checked. This is available under **Project > Settings > Chip Editor**. Interrupt Generation Control is important when multiple overlays are used with interrupts shared by multiple user modules across overlays:

- Interrupt API
- IntDispatchMode

InterruptAPI

The InterruptAPI parameter allows conditional generation of a user module’s interrupt handler and interrupt vector table entry. Select “Enable" to generate the interrupt handler and interrupt vector table entry. Select “Disable" to bypass the generation of the interrupt handler and interrupt vector table entry. Properly selecting whether an Interrupt API is to be generated is recommended particularly with projects that have multiple overlays where a single block resource is used by the different overlays. By selecting only Interrupt API generation when it is necessary the need to generate an interrupt dispatch code might be eliminated, thereby reducing overhead.

IntDispatchMode

The IntDispatchMode parameter is used to specify how an interrupt request is handled for interrupts shared by multiple user modules existing in the same block but in different overlays. Selecting “ActiveStatus" causes firmware to test which overlay is active before servicing the shared interrupt request. This test occurs every time the shared interrupt is requested. This adds latency and also produces a nondeterministic procedure of servicing shared interrupt requests, but does not require any RAM. Selecting “OffsetPreCalc" causes firmware to calculate the source of a shared interrupt request only when an overlay is initially loaded. This calculation decreases interrupt latency and produces a deterministic procedure for servicing shared interrupt requests, but at the expense of a byte of RAM.

Application Programming Interface

The Application Programming Interface (API) routines are provided as part of the user module to allow the designer to deal with the module at a higher level. This section specifies the interface to each function together with related constants provided by the “include" files.

Each time a user module is placed, it is assigned an instance name. By default, PSoC Designer assigns the SPIM_1 to the first instance of this user module in a given project. It can be changed to any unique value that follows the syntactic rules for identifiers. The assigned instance name becomes the prefix of every global function name, variable and constant symbol. In the following descriptions the instance name has been shortened to SPIM for simplicity.

Note

In this, as in all user module APIs, the values of the A and X register may be altered by calling an API function. It is the responsibility of the calling function to preserve the values of A and X before the call if those values are required after the call. This “registers are volatile" policy was selected for efficiency reasons and has been in force since version 1.0 of PSoC Designer. The C compiler automatically takes care of this requirement. Assembly language programmers must ensure their code observes the policy, too. Though some user module API function may leave A and X unchanged, there is no guarantee they may do so in the future.

For Large Memory Model devices, it is also the caller's responsibility to preserve any value in the CUR_PP, IDX_PP, MVR_PP, and MVW_PP registers. Even though some of these registers may not be modified now, there is no guarantee that will remain the case in future releases.

This section lists the SPIM supplied API functions.

SPIM_Start

Description:

Sets the mode configuration of the SPI interface and enables the SPIM module by setting the proper bits in the Control register. Before calling this function, all of the slave select signal(s) should be asserted high to deselect connected SPI Slave devices. This should be done in a user-supplied routine.

C Prototype:

```
void  SPIM_Start(BYTE bConfiguration)
```

Assembly:

```
mov   A, SPIM_SPIM_MODE_2 | SPIM_SPIM_LSB_FIRST
lcall  SPIM_Start
```

Parameters:

bConfiguration: One byte that specifies the SPI mode and LSB First configurations. It is passed in the Accumulator. Symbolic names provided in C and assembly, and their associated values, are given in the following table. Note that the symbolic names can be OR'ed together to form the configuration of the SPI interface. Also note that the instance name of the user module is prepended to the symbolic name listed below. For example, if you named the user module SPIM1 when you placed it, the symbolic name of the first mode is SPIM1_SPIM_MODE_0.

Symbolic Name	Value
SPIM_SPIM_MODE_0	0x00
SPIM_SPIM_MODE_1	0x02
SPIM_SPIM_MODE_2	0x04
SPIM_SPIM_MODE_3	0x06
SPIM_SPIM_LSB_FIRST	0x80
SPIM_SPIM_MSB_FIRST	0x00

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

SPIM_Stop

Description:

Disables the SPIM module by clearing the enable bit in the Control register. After a call to this function is made, all of the slave select signal(s) must be asserted high to disable all of the connected SPI Slave devices. This should be done in a user-supplied routine.

C Prototype:

```
void  SPIM_Stop(void)
```

Assembly:

```
lcall  SPIM_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

SPIM_EnableInt

Description:

Enables the SPIM interrupt on the SPI Done condition. The placement location of the SPIM determines the specific interrupt vector and priority.

C Prototype:

```
void  SPIM_EnableInt(void)
```

Assembly:

```
lcall  SPIM_EnableInt
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

SPIM_DisableInt

Description:

Disables the SPIM interrupt on the SPI Done condition.

C Prototype:

```
void  SPIM_DisableInt(void)
```

Assembly:

```
lcall  SPIM_DisableInt
```

Parameters:

None

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

SPIM_SendTxData

Description:

Initiates the SPI transmission to a slave SPI device. Just before this call, the specified SPI slave device's signal must be asserted low. This should be done in a user-supplied routine.

C Prototype:

```
void  SPIM_SendTxData(BYTE bSPIMData)
```

Assembly:

```
mov   A, bSPIMData
lcall  SPIM_SendTxData
```

Parameters:

BYTE bSPIMData: Data to be sent to the SPI slave device. It is passed in Accumulator.

Return Value:

None

Side Effects:

The A and X registers may be altered by this function.

SPIM_bReadRxData

Description:

Returns a received data byte from a slave device. The Rx Buffer Full flag should be checked before calling this routine, to verify that a data byte has been received.

C Prototype:

```
BYTE  SPIM_bReadRxData(void)
```

Assembly:

```
lcall  SPIM_bReadRxData
mov   bRxData, A
```

Parameters:

None

Return Value:

Data byte received from the slave SPI and returned in the Accumulator.

Side Effects:

The A and X registers may be altered by this function.

SPIM_bReadStatus

Description:

Reads and returns the current SPIM Control/Status register.

C Prototype:

```
BYTE  SPIM_bReadStatus(void)
```

Assembly:

```
lcall  SPIM_bReadStatus
and   A, SPIM_SPIM_SPI_COMPLETE | SPIM_SPIM_RX_BUFFER_FULL
jnz   SpimCompleteGetRxData
```

Parameters:

None

Return Value:

Returns status byte read and is returned in the Accumulator. Utilize defined masks to test for specific status conditions. Note that masks can be OR'ed together to test for multiple conditions. Also note that the instance name of the user module is prepended to the symbolic name listed below. For example, if you named the user module SPIM1 when you placed it, the symbolic name of the first mask is SPIM1_SPIM_SPI_COMPLETE.

SPIM Status Masks	Value
SPIM_SPIM_SPI_COMPLETE	0x20
SPIM_SPIM_RX_OVERRUN_ERROR	0x40
SPIM_SPIM_TX_BUFFER_EMPTY	0x10
SPIM_SPIM_RX_BUFFER_FULL	0x08

Side Effects:

The status bits are cleared after this function is called. The A and X registers may be altered by this function.

Sample Firmware Source Code

In the following C and Assembly sample code, The SPI Master transmits a zero-terminated string stored in RAM. The function loads bytes, one at a time, into the Digital PSoC block TX register using the SPIM API. The digital block begins transmission by transferring its TX register to its Shift register. Repeating the API call, this function immediately reloads the transmit register with the next byte. This way, the transmission proceeds at the maximum continuous rate. In this simple version, any data received from this slave is discarded. This function returns just after the final byte is loaded in the TX register and while the next-to-last byte is still being shifted out on the MOSI line. Subsequent calls to this function does not corrupt an ongoing transmission, as the status is always checked before loading a value into the TX register.

```
#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all user modules

CHAR Message1[] = "Hello World.";
CHAR *pbStrPtr = Message1;

void main(void)
{
    SPIM_Start(SPIM_SPIM_MODE_0 | SPIM_SPIM_MSB_FIRST);

while( *pbStrPtr != 0 )  /* While data remains to be sent */
    {
 /* Ensure the transmit buffer is free */
 while( ! (SPIM_bReadStatus() & SPIM_SPIM_TX_BUFFER_EMPTY ) );

 SPIM_SendTxData( *pbStrPtr );  /* load the next byte */
 pbStrPtr++;
    }
}
```

The equivalent code, written in Assembly, is:

```
include "m8c.inc"        ; part specific constants and macros
include "memory.inc"     ; Constants & macros for SMM/LMM and Compiler
include "PSoCAPI.inc"    ; PSoC API definitions for all user modules

export _main
export Message

AREA text (ROM, REL)

.LITERAL
Message:
    ASCIZ "Hello World."
.ENDLITERAL

_main:
    mov   A, SPIM_SPIM_MODE_0 | SPIM_SPIM_MSB_FIRST
lcall SPIM_Start           ;   Initialize the digital PSoC Block
    ; [...]                    ;   Set SlaveSelect signal low to enable slave
    mov   X, 0                 ;   Set index to beginning of string
.TxNextByteLoop:
mov   A, X
```

```
    index Message
    jz    .Finish               ;    No, bail out of the loop.
push  A
.WaitForTxEmpty:
lcall SPIM_bReadStatus      ;   Transmit buffer free? ...
and   A, SPIM_SPIM_TX_BUFFER_EMPTY
jz    .WaitForTxEmpty       ;   No,  keep checking the status
pop   A
    lcall SPIM_SendTxData       ;   Yes, load TX with the next byte, ...
inc   X                     ;        Advance the pointer, ...
jmp   .TxNextByteLoop       ;        and repeat until done.
.Finish:                        ;   Transmit complete!
lcall SPIM_bReadStatus      ;   Buffer empty for last time? ...
    and   A, SPIM_SPIM_TX_BUFFER_EMPTY
jz    .Finish               ;   No,  keep checking the status
.WaitForTxComplete:
    lcall SPIM_bReadStatus      ;   Last byte transmission complete? ...
and   A, SPIM_SPIM_SPI_COMPLETE
jz    .WaitForTxComplete    ;   No,  keep checking the status
; [...]                     ;   (Reset SlaveSelect signal back high)
.AllDone:
    ; Endless loop
    jmp   .AllDone
```

Configuration Registers

The Digital Communication Type A PSoC block registers used to configure this user module are described below. Only the parameterized symbols are explained.

Table 3. Block SPIM, Register: Function

Bit	7	6	5	4	3	2	1	0
Value	0	0	0	0	0	1	1	0

This register defines the personality of this Digital Communications Type ‘A’ Block to be a SPIM User Module.

Table 4. Block SPIM, Register: Input

Bit	7	6	5	4	3	2	1	0
Value	MISO				Clock			

MISO is the Master-In-Slave-Out input signal. Clock is the selected clock to drive the SPIM timing. Both are set using the Device Editor during parameter selection.

Table 5. Block SPIM, Register: Output

Bit	7	6	5	4	3	2	1	0
Value	0	0	SCLK			MOSI		

MOSI is the Master-Out-Slave-In output signal. SCLK is the SPI clock signal. Both are set using the Device Editor during parameter selection.

Table 6. Block SPIM, Shift Register: DR0

Bit	7	6	5	4	3	2	1	0
Value	Shift Register							

Shift Register is a SPI Shift register.

Table 7. Block SPIM, TX Data Buffer Register: DR1

Bit	7	6	5	4	3	2	1	0
Value	TX Buffer Register							

TX Buffer Register: data written to this buffer is transferred to the Shift register when the PSoC block is enabled.

Table 8. Block SPIM, RX Data Buffer Register: DR2

Bit	7	6	5	4	3	2	1	0
Value	RX Buffer Register							

RX Buffer Register: data received in the Shift register is transferred to this register after completion of the SPI transmit cycle.

Table 9. Block SPIM, Control Register: CR0

Bit	7	6	5	4	3	2	1	0
Value	LSB First	RX Overrun Error	SPI Done	TX Buffer Empty	RX Buffer Full	Clock Phase	Clock Polarity	SPIM Enable

LSB First specifies that the LSB bit should be transmitted first.

RX Overrun Error is a flag that indicates that the previously received data byte was not read before the next byte was received.

SPI Done is a flag that indicates that the complete SPI transmit/receive cycle has been completed.

TX Buffer Empty is a flag that indicates that the TX buffer is empty.

RX Buffer Full is a flag that indicates that a byte of data has been received from the Shift register.

Clock Phase indicates the phase of the SCLK signal. It is one of the parameters that defines the SPI mode.

Clock Polarity indicates the polarity of the SCLK signal. It is one of the parameters that defines the SPI mode.

SPIM Enable enables the SPIM PSoC block when set.

Version History

Version	Originator	Description
2.6	TDU	Updated Clock description to include: When using an external digital clock for the block, the row input synchronization should be turned off for best accuracy, and sleep operation.
2.6.b	DHA	Updated MISO parameter description.

Note PSoC Designer 5.1 introduces a Version History in all user module datasheets. This section documents high level descriptions of the differences between the current and previous user module versions.

Copyright © 2002-2014 Cypress Semiconductor Corporation. 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 Designer™ and Programmable System-on-Chip™ are trademarks and PSoC® is a registered trademark 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.