SPI Slave Datasheet SPIS V 2.5

Copyright © 2002-2015 Cypress Semiconductor Corporation. All Rights Reserved.

Resources	PSoC® Blocks			API Memory (Bytes)		Pins (per External I/O)
	CapSense®	I²C/SPI	Timer	Flash	RAM	
CY8C20x34, CY8C20x24, CY8C20x66, CY8C20x36, CY8C20336AN, CY8C20436AN, CY8C20636AN, CY8C20xx6AS, CY8C20XX6L, CY8C20x46, CY8C20x96, CY7C604xx, CY7C643xx, CYONS2010, CYONS2011, CYONSFN2051, CYONSFN2053, CYONSFN2061, CYONSFN2151, CYONSFN2161, CYONSFN2162, CYONSFN2010-BFXC, CYONSCN2024-BFXC, CYONSCN2028-BFXC, CYONSCN2020-BFXC, CYONSKN2033-BFXC, CYONSKN2035-BFXC, CYONSKN2030-BFXC, CYONSTN2040, CY8CTST200, CY8CTMG2xx, CY8C20xx7/7S, CYRF89x35, CY8C20065, CY8C24x93, CY7C69xxx						
		X		43	0	4

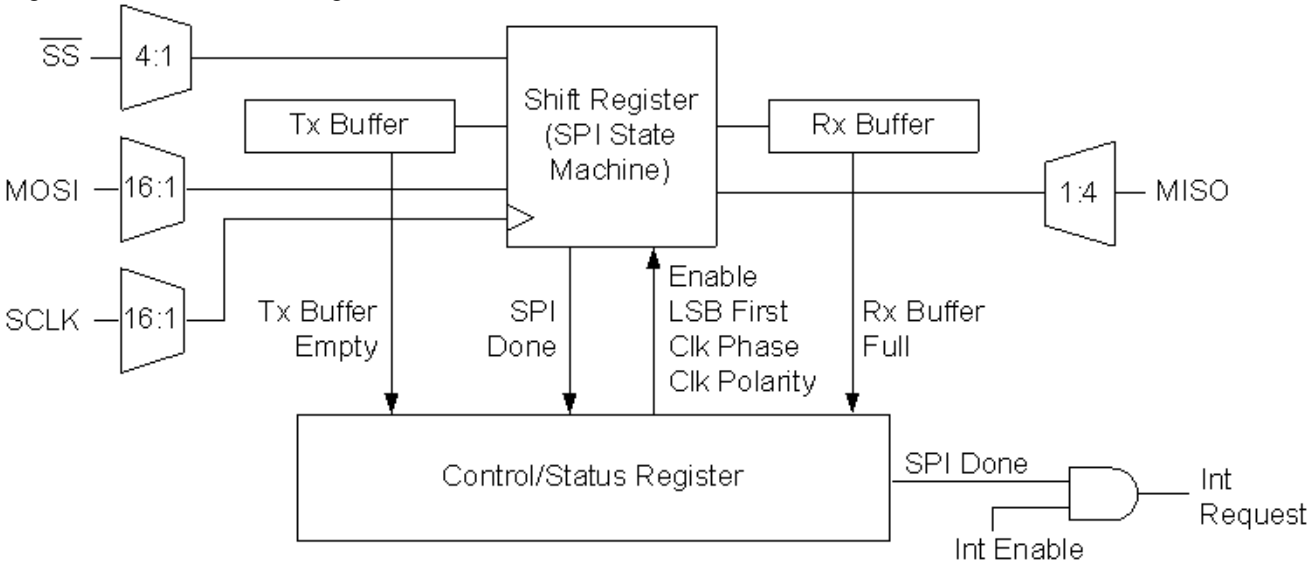
Features and Overview

- Supports serial peripheral interconnect (SPI) slave protocol
- Supports protocol modes 0, 1, 2, and 3
- Selectable input sources for MOSI, SCLK, and ~SS
- Selectable output routing for MISO
- Programmable interrupt on SPI done condition
- SS may be firmware controlled

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

The SPIS User Module is a SPI slave (SPIS). It performs full duplex synchronous 8-bit data transfers. You can specify SCLK phase, SCLK polarity, and LSB First to accommodate most SPI protocols. The SPIS PSoC block has selectable routing for the input and output signals and programmable interrupt driven control. Application Programming Interface (API) firmware provides a high level programming interface for either assembly or C application software.

Figure 1. SPIS Block Diagram



Functional Description

SPIS is a user module that implements a serial peripheral interconnect slave. It uses the Tx Buffer, Rx Buffer, Control, and Configuration registers of the I2C/SPI PSoC block and a data register for data shifting.

The Control register is initialized and configured using the Device Editor and/or the SPIS user module firmware API routines. Initialization includes setting LSB first and the SPI transmission/receive protocol modes. SPI modes 0, 1, 2, and 3 are supported. Set both the SPI Master and SPI Slave with the same mode and bit configuration in order to properly communicate. The SPI modes are defined as follows.

Table 1. SPI Modes

Mode	SCLK Edge Performing Data Latch	Clock Polarity	Notes
0	Leading	Idle low	Leading edge latches data. Data changes on trailing edge of clock.
1	Leading	Idle high	
2	Trailing	Idle low	Trailing edge latches data. Data changes on leading edge.
3	Trailing	Idle high	

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 SCLK input signal is the SPI transmit/receive clock generated by the SPI Master. It defines the bit rate of the transmitted/received data.

The MOSI input signal is the master out slave In data signal that receives the data from the SPI Master.

The MISO output signal is the master In SlaveOut data signal that transmits the data from the Shift register to the SPI master device. Typically, the MISO signals of multiple slaves are tied together. Each slave tri-states its respective MISO signal, until its slave select signal is asserted. This user module does not tri-state the MISO output signal. Add this functionality to the user module if required.

The SPIS hardware receives data from the master SPI device on the MOSI signal and simultaneously transmits data to the SPI Master 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. The master asserts the Slave Select (~SS) input signal low, to enable the specific SPIS device for active communication.

It is the master's responsibility to determine if the selected slave device is ready for a command or is ready to receive data.

The SPIS user module is enabled for operation when the SPI enable bit is set in the control register using an API routine.

Data transmitted to the SPI Master is written to the Tx buffer register. This clears the Tx Buffer Empty status bit.

On the falling edge of the Slave Select signal, the data is transferred from the Tx Buffer register to the Shift register. The first transmitted bit of the data byte is then asserted on the MISO output signal. Another data byte to transmit is written to the Tx Buffer register at this time. When transmission of the current byte is complete, this data is ready to send to the SPI master.

On the assertion of each SCLK input signal, the data is simultaneously shifted out of the Shift register to the MISO output and shifted into the Shift register from the MOSI input. The specific timing of the SCLK, MOSI, and MISO signals are based upon the SPI mode configuration.

After all of the bits are 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. If the interrupt is enabled, the SPI Done status bit cause it to trigger. This interrupt is used to alert the software that a byte of data was received or that a byte transmission succeeded.

If a pending byte of data is currently loaded in the Tx Buffer register, then this byte is ready to transmit when the master performs the next transaction.

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

Monitor the SPI Done bit to determine when to disable the SPIS user module. This guarantees that all of the clocking signals are complete between the master and slave SPI devices.

If interrupts are used, the SPIS status register must be cleared following each interrupt for the next interrupt to be recognized. Reading the status register clears the internal signal that is recognized by the interrupt controller. If this signal remains high, subsequent SPIS interrupts are masked. This applies to the TxComplete and RxComplete interrupts, not to the TxRegEmpty and RxRegEmpty interrupts. Either the assembly language **MOV** or **TST** opcode can be used to read and clear the register.

DC and AC Electrical Characteristics

Table 2. SPIS DC and AC Electrical Characteristics

Parameter	Conditions and Notes	Typical	Limit	Units
F_{max}	Maximum RX/TX frequency	--	12	MHz

Placement

The SPIS maps onto a single PSoC block. The block name is SPIS in the PSoC Designer Device Editor. It may be placed in the I2C/SPI block.

Parameters and Resources

SS Enable

This option determines if the slave select (SS_) signal is driven internally. If the SW_SlaveSelect option is chosen, its logic level is determined by the SS_ bit (Mask 0x08) of the SPI_CFG register. If it is driven externally, its logic level is determined by an external pin.

Interrupt Mode

This option determines when an interrupt is generated for the TX block. The TxRegEmpty option generates an interrupt as soon as the data is 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 when it is important to know when the character is completely sent. Use the first option, TxRegEmpty, to maximize the output of the transmitter. It allows you to load a byte while the previous byte is sent.

Interrupt Generation Control

There following parameter is only available if 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.

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 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.

Note

In this, as in all user module APIs, you can alter the values of the A and X register 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 is used for efficiency reasons since version 1.0 of PSoC Designer. The C compiler automatically handles this requirement. Assembly language programmers must also ensure their code observes the policy. Though some user module API function may leave A and X unchanged, there is no guarantee they may 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.

The following is the list of SPIS supplied API functions.

SPIS_Start

Description:

Sets the mode configuration of the SPI interface and enables the SPIS module by setting the proper bits in the Control register.

Before calling this function, set all the slave select signal(s) asserted high to deselect connected SPI Slave devices. Do this in a user supplied routine.

C Prototype:

```
void  SPIS_Start(BYTE bConfiguration)
```

Assembly:

```
mov   A, SPIS_SPI_MODE_2 | SPIS_SPI_LSB_FIRST
lcall  SPIS_Start
```

Parameters:

bConfiguration: One byte that specifies the SPI mode and LSB First configurations. 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’d together to form the configuration of the SPI interface.

Symbolic Name	Value
SPIS_SPI_MODE_0	0x00
SPIS_SPI_MODE_1	0x02
SPIS_SPI_MODE_2	0x04
SPIS_SPI_MODE_3	0x06
SPIS_SPI_LSB_FIRST	0X80
SPIS_SPI_MSB_FIRST	0X00

Return Value:

None

Side Effects:

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

SPIS_Stop

Description:

Disables the SPIS module by clearing the enable bit in the Control register.

C Prototype:

```
void SPIS_Stop(void)
```

Assembly:

```
lcall  SPIS_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

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

SPIS_EnableInt

Description:

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

C Prototype:

```
void SPIS_EnableInt(void)
```

Assembly:

```
lcall  SPIS_EnableInt
```

Parameters:

None

Return Value:

None

Side Effects:

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

SPIS_DisableInt

Description:

Disables the SPIS interrupt on the SPI Done condition.

C Prototype:

```
void SPIS_DisableInt(void)
```

Assembly:

```
lcall  SPIS_DisableInt
```

Parameters:

None

Return Value:

None

Side Effects:

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

SPIS_SetupTxData

Description:

Writes the data byte to transmit to the SPI Master into the Tx Buffer register.

C Prototype:

```
void SPIS_SetupTxData(BYTE bTxData)
```

Assembly:

```
mov   A, bTxData
lcall  SPIS_SetupTxData
```

Parameters:

bTxData: Data to send to the SPI Master device and passed in the Accumulator.

Return Value:

None

Side Effects:

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

SPIS_bReadRxData

Description:

Returns a received data byte from a slave device. Check the Rx Buffer Full flag before calling this routine to verify that a data byte was received.

C Prototype:

```
BYTE SPIS_bReadRxData(void)
```

Assembly:

```
lcall  SPIS_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.

SPIS_bReadStatus

Description:

Reads and returns the current SPIS Control/Status register.

C Prototype:

```
BYTE SPIS_bReadStatus(void)
```

Assembly:

```
lcall  SPIS_bReadStatus
and   A, SPIS_SPIS_SPI_COMPLETE | SPIS_SPIS_RX_BUFFER_FULL
jnz   SPI_COMPLETE_GET_RX_DATA
```

Parameters:

None

Return Value:

Returns status byte read and is returned in the Accumulator. Use defined masks to test for specific status conditions. Note that masks can be OR'ed together to test for multiple conditions.

SPIS Status Masks	Value
SPIS_SPIS_SPI_COMPLETE	0x20
SPIS_SPIS_RX_OVERRUN_ERROR	0x40
SPIS_SPIS_TX_BUFFER_EMPTY	0x10
SPIS_SPIS_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.

SPIS_DisableSS

Description:

Sets the active-low Slave Select signal (~SS) to the high state using firmware when an external ~SS signal is not required. In order to use this function, set the SPI parameter named Slave Select Input to the value of SW_SlaveSelect, rather than one of the row inputs. If an external signal is connected, this function is not effective.

C Prototype:

```
void SPIS_DisableSS(void)
```

Assembly:

```
lcall SPIS_DisableSS
```

Parameters:

None

Return Value:

None

Side Effects:

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

SPIS_EnableSS

Description:

Sets the active-low Slave Select signal (~SS) to the low state using firmware when an external ~SS signal is not required. In order to use this function, set the SPIS parameter named SS Enable to the value of SW_SlaveSelect, rather than SlaveSelect_Pin. If an external signal is connected, this function is not effective.

C Prototype:

```
void SPIS_EnableSS(void)
```

Assembly:

```
lcall SPIS_EnableSS
```

Parameters:

None

Return Value:

None

Side Effects:

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

Sample Firmware Source Code

This example shows how to create an SPI loop back. It echoes the data received from the SPI Master. If an error condition is detected, a NAK is sent.

```
;-------------------------------------------------------------
;  Loop Back:
;
;  This code sample illustrates how to setup a SPI Slave loop back.
;  Data received is echoed back to the SPI master.
;
;  This sample is written so that it is performed in the non interrupt
;  processing. This code is easily written as interrupt driven.
;
;  You must first properly set and start the SPI Slave user module.
;
;-------------------------------------------------------------
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   LoopBack
export   _main

_main:
        mov   A, SPIS_SPI_MODE_2 | SPIS_SPI_LSB_FIRST
        lcall SPIS_Start
        call  LoopBack
LoopBack:
        ; wait for data to be received
        lcall SPIS_bReadStatus
        and   A, SPIS_SPIS_SPI_COMPLETE
        jz    LoopBack

        ; read the data from the receiver
        lcall SPIS_bReadRxData

        ; setup to transmit the response data
        lcall SPIS_SetupTxData

        ; go wait for next byte
        jmp   LoopBack
```

The same code written in C:

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

void  LoopBack(void)
{
    BYTE  bData;
    while(1)
    {
        /* wait for data to be received */
        while( !( SPIS_bReadStatus() & SPIS_SPIS_SPI_COMPLETE ) );
```

```
        /* read the received data */
        bData = SPIS_bReadRxData();


        /* setup to transmit the response data */
        SPIS_SetupTxData(bData);
    }
}
void main(void)
{
   SPIS_Start(SPIS_SPI_MODE_2 | SPIS_SPI_LSB_FIRST);
   LoopBack ();
}
```

Configuration Registers

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

Table 3. Block SPIS: Configuration Register

Bit	7	6	5	4	3	2	1	0
Value	Clock Sel			Bypass	SS_	SS_EN_	IntSel	Slave

Clock Sel - Clock Selection. These bits determine the operating frequency of the SPI Slave.

000b - SysClk / 2

001b - SysClk / 4

010b - SysClk / 8

011b - SysClk / 16

100b - SysClk / 32

101b - SysClk / 64

110b - SysClk / 128

111b - SysClk / 256

Bypass - Bypass Synchronization. This bit determines whether or not the inputs are synchronized to SYSCLK.

SS_ - Slave Select. This bit determines the logic value of the SS_ signal when the SS_EN_ signal is asserted (SS_EN_ = 0).

SS_EN_ - Slave Select Enable. This active low bit determines if the slave select (SS_) signal is driven internally. If it is driven internally, its logic level is determined by the SS_ bit. If it is driven externally, its logic level is determined by the external pin.

Int Sel - Interrupt Select. This bit selects which condition produces an interrupt, and whether it is based upon the TX Reg Empty condition or the SPI Complete condition.

Slave - this bit determines if the block functions as a master or slave.

Table 4. Block SPIS: TX Data Buffer Register

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

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

Table 5. Block SPIS: RX Data Buffer Register

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 6. Block SPIS: Control Register

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	Enable

LSB First - transmits the LSB bit first.

RX Overrun Error - a flag indicating that the previously received data byte was not read before receiving the next byte.

SPI Done - a flag that indicates that the completion of the SPI transmit/receive cycle.

TX Buffer Empty - a flag that indicates an empty TX buffer.

RX Buffer Full - a flag that indicates that the Shift register received a byte of data.

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.

Enable - enables the SPI PSoC block when set.

Version History

Version	Originator	Description
2.5	DHA	P1[0] was set as SPI CLK pin for CY7C64315 and CY7C64316 PSoC devices.
2.5.b	DHA	Added note to ensure that the clock polarity and phase for the selected mode match the settings used for any attached SPI device.
2.5.c	DHA	Updated list of supported devices in the user module datasheet.
2.5.d	HPHA	Added AN51234 reference in Getting Started section.

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 © 2002-2015 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.