SPI Slave Variable Length Datasheet SPISVL V 1.10

Copyright © 2009-2014 Cypress Semiconductor Corporation. All Rights Reserved.

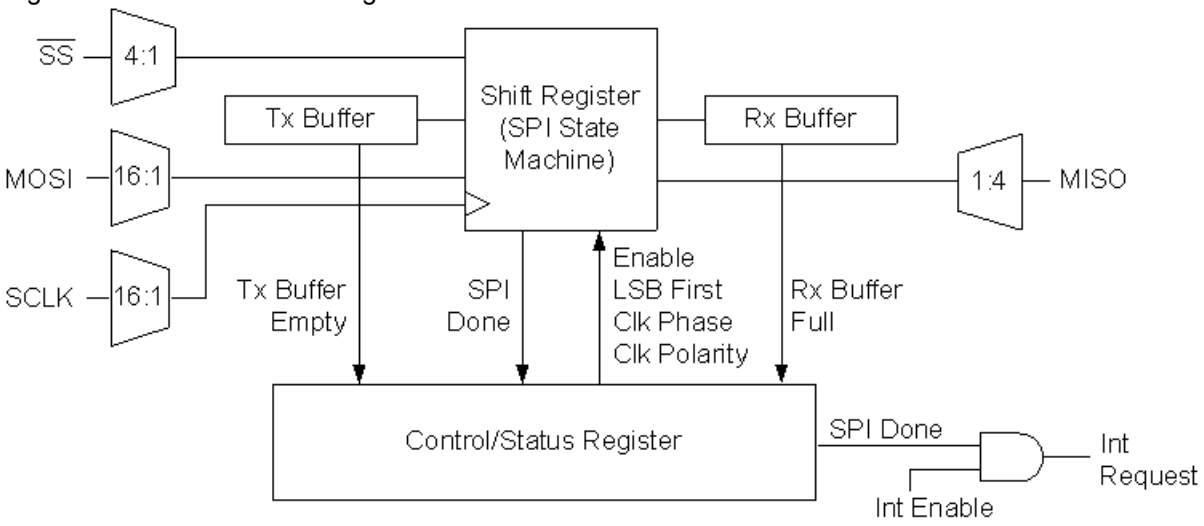
Resources	PSoC® Blocks			API Memory (Bytes)		Pins (per External I/O)
	Digital	Analog CT	Analog SC	Flash	RAM	
CY8C21x45, CY8C22x45, CY8C28x45, CY8C28xxx						
	2	0	0	142	1	4

Features and Overview

- Supports serial peripheral interconnect slave (SPIS) 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 Complete or TX Reg Empty conditions
- SS may be firmware controlled
- Selectable data length - 9 to 16 bits

The SPISVL User Module is a SPIS that can be configured as variable data length. It performs full duplex synchronous data transfers with arbitrary data length between 9 and 16 bits. SCLK phase, SCLK polarity, and LSB First can be specified to accommodate most SPI protocols. The SPISVL PSoC blocks have 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. SPISVL Block Diagram



Functional Description

SPISVL is a user module that implements a SPIS. Since it supports the 9-16 bit data length, it uses pairs of Tx Buffer, Rx Buffer, Control0, Control1, and Shift registers of two digital communication type PSoC blocks and one or more Pin Port registers.

The Control0 and Control1 registers are initialized and configured using the Device Editor or the SPISVL User Module firmware API routines. Initialization includes setting the LSB First configuration, the SPI receive/transmission clocking modes, and the data length. SPI modes 0, 1, 2, and 3 are supported. Both the SPI master (SPIM) and SPIS must be set with the same clock mode and bit configuration for proper communication. The SPI modes are defined in the following table.

Table 1. SPI Modes

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

The SCLK input signal is the SPI transmit/receive clock generated by the SPIM. 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 SPIM.

The MISO output signal is the Master-In-Slave-Out data signal that transmits the data from the Shift registers to the SPIM device. Typically, the MISO signals of multiple slaves are tied together. Each slave tristates its respective MISO signal until its Slave Select signal is asserted. This user module does not tristate the MISO output signal. You can easily add this functionality to the user module, if required.

The SPISVL hardware receives data from the Master SPI device on the MOSI signal and simultaneously transmits data to the SPIM 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 SPISVL User Module is enabled for operation when the SPI Enable bit is set in the Control0 register of the LSB block using an API routine.

Data to be transmitted to the SPIM is written to the Tx Buffer registers. This clears the Tx Buffer Empty status bit of LSB block.

After the falling edge of the Slave Select signal, the data is transferred from the Tx Buffer registers to the Shift registers. The first bit of the data word to be transmitted is then asserted on the MISO output signal. Another data word to transmit can be written to the Tx Buffer registers at this time. Upon completion of the transmission of the current word, this data can be sent to the SPIM.

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

After all of the bits have been transmitted and simultaneously received, the received data is transferred from the Shift registers to the Rx Buffer registers and the Tx Buffer registers are transferred to the Shift registers. The Rx Buffer Full and the SPI Done status bits are set. If the interrupt is enabled, the SPI Done status bit triggers it. This interrupt can be used to alert the software that a byte of data has been received or that a byte was successfully transmitted.

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

If the SPI Done interrupt condition is not used to retrieve the data word from the Rx Buffer registers, the Control0 register should be polled to monitor the Rx Buffer Full status bit. The received data must be read from the Rx Buffer registers before the next data word is fully received or the Overrun Error status bit is set.

The SPI Done bit should be monitored to determine when to disable the SPISVL User Module. This guarantees that all of the clocking signals are completed between the SPIM and SPIS devices.

If interrupts are used, the SPISVL status register must be read 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 SPISVL interrupts is masked.

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

SPISVL maps to two PSoC blocks, which are considered here as an architectural ensemble. SPISVL may be placed in any pair of Digital Communications blocks on the same row.

Parameters and Resources

SCLK

SPISVL is clocked by the SPIM generated SCLK signal. This signal can be routed from one of 15 possible sources. High, low, global I/O buses, analog comparator buses, or another PSoC block can be specified as supplying the SCLK input. This clock defines the effective bit transfer rate. By default SCLK is synchronized to SysClk. If SCLK is not synchronized or synchronized to SysClk*2, then a direct write to the ClockSync bitfield of the Output Register should be used.

MOSI

The MOSI input signal can be routed from one of 15 possible sources. High, low, global I/O buses, analog comparator buses, or another PSoC block can be specified as supplying the MOSI input.

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

~SS

The Slave Select input signal can be routed from one of four possible global inputs. In addition, the ~SS may also be firmware controlled.

MISO

The MISO output signal 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.

Interrupt Type

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, "SPI Complete," 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. 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.

InvertMOSI

This parameter gives you the option to invert the MOSI input.

DataLength

This parameter sets the data length of the SPI communication protocol.

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. Select whether an Interrupt API is to be generated, 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 may 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 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 registers 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 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.

Note that the instance name of the user module replaces "SPISVL" word in constants and function names. For example, if you named the user module SPISVL1 when you placed it, the symbolic name of the first mode is SPISVL_1_MODE_0, the SPISVL_Start function is available as SPISVL_1_Start.

The following is the list of SPISVL supplied API functions.

SPISVL_Start

Description:

Sets the mode configuration of the SPI interface and enables the SPISVL module by setting the proper bits in the Control0 register. Before calling this function, all of the slave select signals should be asserted high to deselect connected SPIS devices. This should be done in a user-supplied routine.

C Prototype:

```
void  SPISVL_Start(BYTE bConfiguration)
```

Assembly:

```
mov   A,SPISVL_MODE_2 | SPISVL_LSB_FIRST
lcall  SPISVL_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.

Symbolic Name	Value
SPISVL_MODE_0	0x00
SPISVL_MODE_1	0x02

Symbolic Name	Value
SPISVL_MODE_2	0x04
SPISVL_MODE_3	0x06
SPISVL_LSB_FIRST	0X80
SPISVL_MSB_FIRST	0X00

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section.

SPISVL_Stop

Description:

Disables the SPISVL module by clearing the enable bit in the Control0 register.

C Prototype:

```
void SPISVL_Stop(void)
```

Assembly:

```
lcall  SPISVL_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section.

SPISVL_EnableInt

Description:

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

C Prototype:

```
void SPISVL_EnableInt(void)
```

Assembly:

```
lcall  SPISVL_EnableInt
```

Parameters:

None

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section.

SPISVL_DisableInt

Description:

Disables the SPISVL interrupt on the SPI Done condition.

C Prototype:

```
void SPISVL_DisableInt(void)
```

Assembly:

```
lcall  SPISVL_DisableInt
```

Parameters:

None

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section.

SPISVL_ClearInt

Description:

Clear posted SPISVL interrupt.

C Prototype:

```
void  SPISVL_ClearInt(void);
```

Assembly:

```
lcall  SPISVL_ClearInt
```

Parameters:

None

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section.

SPISVL_SetupTxData

Description:

Writes the data byte to transmit to the SPIM into the Tx Buffer register.

C Prototype:

```
void SPISVL_SetupTxData(WORD wTxData)
```

Assembly:

```
mov   X, [wTxData]
mov   A, [wTxData+1]
lcall  SPISVL_SetupTxData
```

Parameters:

bTxData: Data to be sent to the SPIM device and passed in the Accumulator.

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section.

SPISVL_wReadRxData

Description:

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

C Prototype:

```
WORD SPISVL_wReadRxData(void)
```

Assembly:

```
lcall  SPISVL_wReadRxData
mov   [wRxData], X
mov   [wRxData+1], A
```

Parameters:

None

Return Value:

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

Side Effects:

See Note ** at the beginning of the API section.

SPISVL_bReadStatus

Description:

Reads and returns the current SPISVL Control/Status register.

C Prototype:

```
BYTE SPISVL_bReadStatus(void)
```

Assembly:

```
lcall  SPISVL_bReadStatus
and   A, SPISVL_COMPLETE | SPISVL_RX_BUFFER_FULL
jnz   SPIVL_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.

SPIM Status Masks	Value
SPISVL_COMPLETE	0x20
SPISVL_RX_OVERRUN_ERROR	0x40
SPISVL_TX_BUFFER_EMPTY	0x10
SPISVL_RX_BUFFER_FULL	0x08

Side Effects:

The status bits are cleared after this function is called. You can alter the A and X registers by this function.

SPISVL_DisableSS

Description:

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

C Prototype:

```
void SPISVL_DisableSS(void)
```

Assembly:

```
lcall SPISVL_DisableSS
```

Parameters:

None

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section.

SPISVL_EnableSS

Description:

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

C Prototype:

```
void SPISVL_EnableSS(void)
```

Assembly:

```
lcall SPISVL_EnableSS
```

Parameters:

None

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section.

Sample Firmware Source Code

This example shows how to create an SPI loop back. It echoes the data received from the SPIM.

```
//
//  This sample shows how to resive a two-byte data word which is stored in RAM.
//
// OVERVIEW:
//
//  The MISO output of SPISVL can be routed to any pin.
//  The MOSI input of SPISVL can be routed to any pin.
//  The SCLK input of SPISVL can be routed to any pin.
//  The SS input of SPISVL can be routed to any pin.
//  In this example: the MISO output is routed to P0[2],
//                   the MOSI input is routed to P0[0]
//                   the SCLK intput is routed to P0[3]
//                   the SS input is routed to P0[1]
//
//The following changes need to be made to the default settings in the Device Editor:
//
//   1. Select SPISVL user module.
//   2. The User Module will occupy the space in dedicated system resources.
//   3. Rename User Module's instance name to SPISVL.
//   4. Set SPISVL's SCLK Parameter to Row_0_Input_3.
//   5. Set SPISVL's MOSI Parameter to Row_0_Input_0.
//   8. Set SPISVL's Slave Select Input Parameter to Row_0_Input_1.
//   6. Set SPISVL's MISO Parameter to Row_0_Output_2.
//   7. Set SPISVL's InterruptType Parameter to SPI Complete.
//   8. Set SPISVL's Data Length Parameter to 9 Bit.
//   9. Set SPISVL's Invert MOSI Parameter to Normal.
//   10.Click on Row_0_Input_3 and connect Row_0_Input_3 to GlobalInEven_3.
//   11.Select GlobalInEven_3 for P0[3] in the Pinout.
//   12.Click on Row_0_Input_0 and connect Row_0_Input_0 to GlobalInEven_0.
//   13.Select GlobalInEven_0 for P0[0] in the Pinout.
//   14.Click on Row_0_Input_1 and connect Row_0_Input_1 to GlobalInEven_1.
//   15.Select GlobalInEven_1 for P0[1] in the Pinout.
//   16.Click on Row_0_Output_2 and connect Row_0_Output_2 to GlobalOutEven_2.
//   17.Select GlobalOutEven_2 for P0[2] in the Pinout.
//
//
// CONFIGURATION DETAILS:
//
//  1. The UM's instance name must be shortened to SPISVL.
```

```
//
// PROJECT SETTINGS:
//
//     CPU_Colck  = SysClk/1       System clock is set to 24MHz
//     VC1=SysClk/N  = 16 (default)
//     VC2=VC1/N  = 16 (default)
//
// USER MODULE PARAMETER SETTINGS:
//
//  ------------------------------------------------------------------------------
//  UM        Parameter          Value                Comments
//  ------------------------------------------------------------------------------
//  SPISVL    Name               SPISVL               UM's instance name
//            SCLK               Row_0_Input_3
//            MOSI               Row_0_Input_0
//            Slave Select Input Row_0_Input_1
//            MISO               Row_0_Output_2
//            InterruptType   SPI Complete
//            Data Length     9 Bit
//            Invert MISO     Normal
//
//  ------------------------------------------------------------------------------

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

void main(void)
{
    int wData;
    SPISVL_Start(SPISVL_MODE_0|SPISVL_MSB_FIRST);
     M8C_EnableGInt;
     while(1)
    {
        while( !(SPISVL_bReadStatus() & SPISVL_RX_BUFFER_FULL) );
                                          // wait for received data
        wData = SPISVL_wReadRxData();      // read received data from the Rx buffer

        while( !(SPISVL_bReadStatus() & SPISVL_TX_BUFFER_EMPTY) );
                                         // check for Tx buffer empty
        SPISVL_SetupTxData(wData);        // send echo packet back
         while( !(SPISVL_bReadStatus() & SPISVL_SPI_COMPLETE) );
                                        // ensure transaction is complete
    }
}
```

The following the same code in Assembly:

```
;
;  This sample shows how to resive a two-byte data word which is stored in RAM end send
; echo packet back.
;
; OVERVIEW:
;
;  The MISO output of SPISVL can be routed to any pin.
```

```
;  The MOSI input of SPISVL can be routed to any pin.
;  The SCLK input of SPISVL can be routed to any pin.
;  The SS input of SPISVL can be routed to any pin.
;  In this example: the MISO output is routed to P0[2],
;                   the MOSI input is routed to P0[0]
;                   the SCLK intput is routed to P0[3]
;                   the SS input is routed to P0[1]
;
;The following changes need to be made to the default settings in the Device Editor:
;
;   1. Select SPISVL user module.
;   2. The User Module will occupy the space in dedicated system resources.
;   3. Rename User Module's instance name to SPISVL.
;   4. Set SPISVL's SCLK Parameter to Row_0_Input_3.
;   5. Set SPISVL's MOSI Parameter to Row_0_Input_0.
;   8. Set SPISVL's Slave Select Input Parameter to Row_0_Input_1.
;   6. Set SPISVL's MISO Parameter to Row_0_Output_2.
;   7. Set SPISVL's InterruptType Parameter to SPI Complete.
;   8. Set SPISVL's Data Length Parameter to 9 Bit.
;   9. Set SPISVL's Invert MOSI Parameter to Normal.
;   10.Click on Row_0_Input_3 and connect Row_0_Input_3 to GlobalInEven_3.
;   11.Select GlobalInEven_3 for P0[3] in the Pinout.
;   12.Click on Row_0_Input_0 and connect Row_0_Input_0 to GlobalInEven_0.
;   13.Select GlobalInEven_0 for P0[0] in the Pinout.
;   14.Click on Row_0_Input_1 and connect Row_0_Input_1 to GlobalInEven_1.
;   15.Select GlobalInEven_1 for P0[1] in the Pinout.
;   16.Click on Row_0_Output_2 and connect Row_0_Output_2 to GlobalOutEven_2.
;   17.Select GlobalOutEven_2 for P0[2] in the Pinout.
;
;
; CONFIGURATION DETAILS:
;
;  1. The UM's instance name must be shortened to SPISVL.
;
; PROJECT SETTINGS:
;
;     CPU_Colck  = SysClk/1       System clock is set to 24MHz
;     VC1=SysClk/N  = 16 (default)
;     VC2=VC1/N  = 16 (default)
;
; USER MODULE PARAMETER SETTINGS:
;
;  ------------------------------------------------------------------------------
;  UM        Parameter          Value                Comments
;  ------------------------------------------------------------------------------
;  SPISVL    Name               SPISVL               UM's instance name
;            SCLK               Row_0_Input_3
;            MOSI               Row_0_Input_0
;            Slave Select Input Row_0_Input_1
;            MISO               Row_0_Output_2
;            InterruptType   SPI Complete
;            Data Length     9 Bit
;            Invert MISO     Normal
;
;  ------------------------------------------------------------------------------
```

```
; Code begins here

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

area bss(RAM, REL)
 wData: blk 2

area text (ROM, REL)

_main:

    mov A, (SPISVL_MODE_0|SPISVL_MSB_FIRST)
    lcall SPISVL_Start

    M8C_EnableGInt

.nextLoop:

.WaitRx:
    lcall SPISVL_bReadStatus
    and A, SPISVL_RX_BUFFER_FULL
    jz .WaitRx ; wait for received data

    lcall SPISVL_wReadRxData;; read received data from the Rx buffer
    mov  [wData], X
    mov  [wData+1], A
    push A
    push X

.WaitTx:
    lcall SPISVL_bReadStatus
    and A, SPISVL_TX_BUFFER_EMPTY
    jz .WaitTx; check for Tx buffer empty

    pop X
    pop A

    lcall SPISVL_SetupTxData        ; send echo packet back

.WaitComplete:
    lcall SPISVL_bReadStatus
    and A, SPISVL_SPI_COMPLETE      ; ensure transaction is complete
    jz .WaitComplete
jmp .nextLoop
```

Configuration Registers

This section describes the Digital Communication Type A PSoC block registers used to configure this user module. Only the parameterized symbols are explained.

Table 3. Block SPIMVL, Register: FunctionMSB First Configuration:

Bit	7	6	5	4	3	2	1	0
LSB	InvertMOSI	0	0	Interrupt Type	1	1	1	0
MSB	0	0	0	0	1	1	1	0

Table 4. Block SPIMVL, Register: FunctionLSB First Configuration:

Bit	7	6	5	4	3	2	1	0
LSB	0	0	0	Interrupt Type	1	1	1	0
MSB	InvertMOSI	0	0	0	1	1	1	0

This register defines the characteristics required by Digital Communications Blocks to be a SPISVL User Module.

Table 5. Block SPISVL: Register InputMSB First Configuration:

Bit	7	6	5	4	3	2	1	0
LSB	MOSI				SCLK			
MSB					SCLK			

Table 6. Block SPISVL: Register InputLSB First Configuration:

Bit	7	6	5	4	3	2	1	0
LSB					SCLK			
MSB	MOSI				SCLK			

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

Table 7. Block SPISVL: Register OutputMSB First Configuration:

Bit	7	6	5	4	3	2	1	0
LSB	0	0	0	~SS				
MSB	0	0	0			MISO		

Table 8. Block SPISVL: Register OutputLSB First Configuration:

Bit	7	6	5	4	3	2	1	0
LSB				~SS		MISO		
MSB								

~SS is the Slave Select input signal. MISO is the Master-In-Slave-Out output signal. Both are set using the Device Editor during parameter selection.

Table 9. Block SPISVL: Shift Register DR0

Bit	7	6	5	4	3	2	1	0
LSB	Shift Register							
MSB	Shift Register							

Table 10. Block SPISVL: TX Data Buffer Registers DR1

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

Tx Buffer Registers: Data written to this buffer is transferred to the Shift registers when the PSoC block is enabled.

Table 11. Block SPISVL: RX Data Buffer Register DR2

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

RX Buffer Registers: Data received in the Shift registers is transferred to these registers after completion of the SPI transmit cycle.

Table 12. Block SPISVL: Control Register CR0

Bit	7	6	5	4	3	2	1	0
LSB	LSB First	RX Overrun Error	SPI Done	TX Buffer Empty	Rx Buffer Full	Clock Phase	Clock Polarity	SPIS Enable
MSB	LSB First	RX Overrun Error	SPI Done	TX Buffer Empty	Rx Buffer Full	Clock Phase	Clock Polarity	SPIS 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 SPI transmit/receive cycle is complete.

Tx Buffer Empty is a flag that indicates that the Tx Buffer is empty.

Rx Buffer Full is 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.

SPIS Enable enables the SPIS PSoC block when set.

Table 13. Block SPISVL: Control Register CR1

Bit	7	6	5	4	3	2	1	0
LSB	1	1	0	SPI Length				
MSB	1	0	0	SPI Length				

SPI Length Specifies the SPI length in chain mode.

Version History

Version	Originator	Description
1.0	DHA	Initial version
1.0.b	DHA	Updated MISO parameter description in this user module datasheet.
1.10	HPHA	Corrected method of clearing posted interrupts.

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 © 2009-2014 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.