SPI Master Variable Length Datasheet SPIMVL V 1.10

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

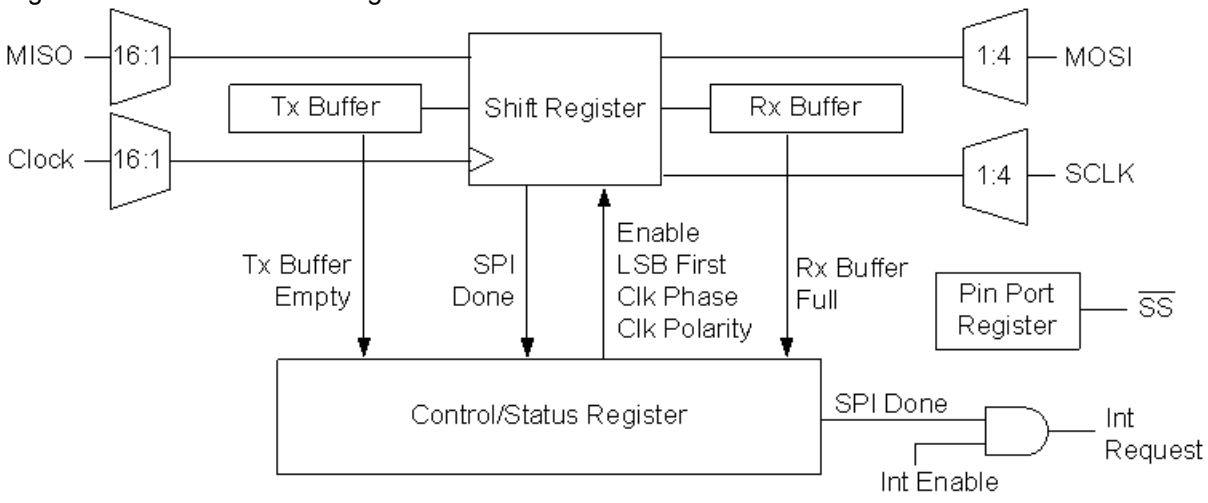
Resources	PSoC® Blocks			API Memory (Bytes)		Pins (per External I/O)
	Digital	Analog CT	Analog SC	Flash	RAM	
CY8C21x45, CY8C22x45, CY8C28x45, CY8C28xxx	2	0	0	93	0	3 - 4

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 Complete or TX Reg Empty conditions
- SPI Slave devices can be independently selected
- Selectable Data Length – from 9 to 16 bits

The SPI Master Variable Length User Module (SPIMVL) is a Serial Peripheral Interconnect Master that can be configured with a 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 clocking modes. Controlled by user-supplied software, the slave select signal can be configured to control one or more SPI Slave devices. The SPIMVL PSoC blocks have selectable routing for the input and output signals, and programmable interrupt-driven control.

Figure 1. SPIMVL Block Diagram



Functional Description

SPIMVL is a user module that implements a Serial Peripheral Interconnect Master. Since it supports a 9- to16-bit variable data length, it uses pairs of transmit (Tx) and receive (Rx) buffers, and the 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 and the SPIMVL User Module firmware Application Programming Interface (API) routines. Initialization includes setting the LSB First configuration, the SPI transmit and receive clocking modes, and the data length. SPI modes 0, 1, 2, and 3 are supported. The SPI master and all slaves must be set with the same clock 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	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	

The active low slave select signal, ~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 and receive clock. It is one-half the clock rate of the input clock signal. The effective transmit or 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 SPIMVL 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 SPIMVL User Module is enabled for operation when the SPI enable bit is set in the Control0 register LSB block using an API routine. All slave select signals should be asserted high.

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

The MSB and LSB of 2-byte word to be transmitted are written to the Tx Buffer registers of MSB and LSB SPIMVL digital blocks respectively. This clears the Tx Buffer Empty status bit in the Control0 register of

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

For each bit of the data word in the Shift registers, the SCLK output signal will be generated, the data in the Shift registers will be shifted to the MOSI output, and the input data from the slave SPI will be shifted into the Shift registers from the MISO input. The specific timing of the SCLK, MOSI, and MISO signals will be 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 registers to the Rx Buffer registers, and the Tx Buffer registers data is transferred to the Shift registers. The Rx Buffer Full and the SPI Done status bits are set. The SPI Done status bit will cause an interrupt to trigger, if interrupts are enabled.

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 will be set.

If a pending word of data has been preloaded into the Tx Buffer registers, then this word will restart the SPI state machine and transmission will commence immediately.

Two methods can be deployed to control the slave select signal after each data word 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 word of data is transmitted. The word transport protocol used for multi-word 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 non-interrupt level will require 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.

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

DC and AC Electrical Characteristics

Table 2. SPIMVL DC and AC Electrical Characteristics

Parameter	Conditions and Notes	Typical	Limit	Units
F_{max}	Maximum bit rate	4.1[1]	--	Mbps

1. 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. See the device datasheet for maximum values for your device.

Placement

SPIMVL maps onto two PSoC blocks which considered here as a architectural ensemble. SPIMVL may be placed into any pair of the Digital Communications blocks on the same row.

Pin Port register bit(s) will need 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

SPIMVL is clocked by one of the available sources. The global I/O buses 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 SPIMVL 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 one of the available 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 SPIMVL 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 Type

This option determines when an interrupt will be generated for the SPIMVL LSB block.

Option	Description
TX Reg Empty	Causes an interrupt to be generated as soon as the data has been transferred from the Data register to the Shift register. This option maximizes the output of the transmitter. It allows a byte to be loaded while the previous byte is being sent.
SPI Complete	Delays the interrupt until the last bit is shifted out of the Shift register. This option is useful when it is important to know when the character has been completely sent.

ClockSync

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), 32 kHz, 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 this setting when a 24 MHz (SysClk/1) clock is required. 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 this setting when the 48 MHz (SysClk*2) input is selected. Use when unsynchronized inputs are required. 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.

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. 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 SPIMVL_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 SPIMVL 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 prior to 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 will do so in the future.

For Large Memory Model devices, it is also the caller's responsibility to perserve 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.

SPIMVL_Start

Description:

Sets the mode configuration of the SPI interface and enables the SPIMVL module by setting the proper bits in the Control0 register. Upon setting mode configuration this function reconnects the SPIMVL blocks to I/O rows depending on wConfiguration input parameter value. For SPIMVL_MSB_FIRST mode actual connections are - MISO - to LSB block, SCK and MOSI - to MSB

block, for SPIMVL_LSB_FIRST mode actual connections are - MISO - to MSB block, SCK and MOSI - to LSB block.

Prior to 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  SPIMVL_Start(BYTE bConfiguration)
```

Assembly:

```
mov   A,SPIMVL_MODE_2 | SPIMVL_LSB_FIRST
lcall  SPIMVL_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.

Symbolic Name	Value
SPIMVL_MODE_0	0x00
SPIMVL_MODE_1	0x02
SPIMVL_MODE_2	0x04
SPIMVL_MODE_3	0x06
SPIMVL_LSB_FIRST	0x80
SPIMVL_MSB_FIRST	0x00

Return Value:

None

Side Effects:

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

SPIMVL_Stop

Description:

Disables the SPIMVL module by clearing the enable bit in the Control0 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  SPIMVL_Stop(void)
```

Assembly:

```
lcall  SPIMVL_Stop
```

Parameters:

None

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section

SPIMVL_EnableInt

Description:

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

C Prototype:

```
void  SPIMVL_EnableInt(void)
```

Assembly:

```
lcall  SPIMVL_EnableInt
```

Parameters:

None

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section

SPIMVL_DisableInt

Description:

Disables the SPIMVL interrupt on the SPI Done condition.

C Prototype:

```
void  SPIMVL_DisableInt(void)
```

Assembly:

```
lcall  SPIMVL_DisableInt
```

Parameters:

None

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section

SPIMVL_ClearInt

Description:

Clear posted SPIMVL interrupt.

C Prototype:

```
void  SPIMVL_ClearInt(void);
```

Assembly:

```
lcall  SPIMVL_ClearInt
```

Parameters:

None

Return Value:

None

Side Effects:

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

SPIMVL_SendTxData

Description:

Initiates the SPI transmission to a slave SPI device. Just prior to 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  SPIMVL_SendTxData(WORD wSPIMVLData)
```

Assembly:

```
mov   X, [bSPIMVLData]
mov   A, [bSPIMVLData+1]
lcall  SPIMVL_SendTxData
```

Parameters:

WORD wSPIMVLData: Data to be sent to the SPI slave device. It is passed in A and X registers.

Return Value:

None

Side Effects:

See Note ** at the beginning of the API section

SPIMVL_wReadRxData

Description:

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

C Prototype:

```
WORD  SPIMVL_wReadRxData(void)
```

Assembly:

```
lcall  SPIMVL_wReadRxData
mov   [bRxData], X
mov   [bRxData+1], A
```

Parameters:

None

Return Value:

Data word received from the slave SPI and returned in the A and X registers.

Side Effects:

See Note ** at the beginning of the API section

SPIMVL_bReadStatus

Description:

Reads and returns the current SPIMVL Control0/Status register.

C Prototype:

```
BYTE  SPIMVL_bReadStatus(void)
```

Assembly:

```
lcall  SPIMVL_bReadStatus
and   A, SPIMVL_DONE | SPIMVL_RX_BUFFER_FULL
jnz   SPIMVLCompleteGetRxData
```

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.

SPIMVL Status Masks	Value
SPIMVL_SPI_COMPLETE	0x20
SPIMVL_RX_OVERRUN_ERROR	0x40
SPIMVL_TX_BUFFER_EMPTY	0x10
SPIMVL_RX_BUFFER_FULL	0x08

Side Effects:

The status bits are cleared after this function is called. See Note ** at the beginning of the API section.

Sample Firmware Source Code

In the following C and Assembly sample code, The SPI Master transmits a two-byte data word which is stored in RAM and incremented continuously every main loop cycle. The SPI Master receives the echo packet from the slave and puts it into other word variable.

```
//
// DESCRIPTION:
//
//  This sample shows how to transmit a two-byte data word which is stored in RAM and
//  incremented continuously every main loop cycle.
//
// OVERVIEW:
//
//  The MISO input of SPIMVL can be routed to any pin.
//  The MOSI output of SPIMVL can be routed to any pin.
//  The SCLK output of SPIMVL can be routed to any pin.
//  The SS output of SPIMVL can be routed to any pin.
//  In this example: the MISO input is routed to P0[0],
//                   the MOSI output is routed to P0[4]
//                   the SCLK output is routed to P0[2]
//                   the SS output is routed to P0[6]
//
//The following changes need to be made to the default settings in the Device Editor:
//
//   1. Select SPIMVL user module.
//   2. The User Module will occupy the space in dedicated system resources.
//   3. Rename User Module's instance name to SPIMVL.
//   4. Set SPIMVL's Clock Parameter to VC2.
//   5. Set SPIMVL's MISO Parameter to Row_0_Input_0.
//   6. Set SPIMVL's MOSI Parameter to Row_0_Output_0.
//   7. Set SPIMVL's SCLK Parameter to Row_0_Output_2.
//   8. Set SPIMVL's InterruptType Parameter to SPI Complete.
//   9. Set SPIMVL's ClockSync Parameter to Sync to SysClk.
//   10.Set SPIMVL's Data Length Parameter to 9 Bit.
//   11.Set SPIMVL's Invert MISO Parameter to Normal.
//   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_Output_0 and connect Row_0_Output_0 to GlobalOutEven_4.
//   15.Select GlobalOutEven_4 for P0[4] 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 SPIMVL.
//
// 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
//  ------------------------------------------------------------------------
//  SPIMVL    Name               SPIMVL               UM's instance name
//            Clock              VC2
//            MISO               Row_0_Input_0
//            MOSI               Row_0_Output_0
//            SCLK               Row_0_Output_2
//            InterruptType   SPI Complete
//            ClockSync          Sync to SysClk
//            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)
{
    WORD wDataTx; // data to transmit from
    WORD wDataRx; // data to receive into
    WORD i;
    SPIMVL_Start(SPIMVL_MODE_0|SPIMVL_MSB_FIRST);

    M8C_EnableGInt;

    PRT0DR |= 0x40; // forse SS high first

    while(1)
    {
        PRT0DR &= ~0x40;                        // set SS low before transaction
        while( !(SPIMVL_bReadStatus() & SPIMVL_TX_BUFFER_EMPTY) );
                                                // check for empty Tx buffer
        SPIMVL_SendTxData(wDataTx);             // send data word
        while( !(SPIMVL_bReadStatus() & SPIMVL_SPI_COMPLETE) );
                                               // ensure transaction completed
        wDataRx = SPIMVL_wReadRxData();         // read received data from Rx
buffer        wDataTx++;                             // increment data before transmit
        PRT0DR |= 0x40;                         // disassert SS signal
        for(i=0;i<65000;i++);                   // delay
    }
}
```

The equivalent code written in Assembly is:

```
;
;  This sample shows how to transmit a two-byte data word which is stored in RAM and
; incremented continuously every main loop cycle.
;
; OVERVIEW:
;
;  The MISO input of SPIMVL can be routed to any pin.
```

```
;  The MOSI output of SPIMVL can be routed to any pin.
;  The SCLK output of SPIMVL can be routed to any pin.
;  The SS output of SPIMVL can be routed to any pin.
;  In this example: the MISO input is routed to P0[0],
;                   the MOSI output is routed to P0[4]
;                   the SCLK output is routed to P0[2]
;                   the SS output is routed to P0[6]
;
;The following changes need to be made to the default settings in the Device Editor:
;
;   1. Select SPIMVL user module.
;   2. The User Module will occupy the space in dedicated system resources.
;   3. Rename User Module's instance name to SPIMVL.
;   4. Set SPIMVL's Clock Parameter to VC2.
;   5. Set SPIMVL's MISO Parameter to Row_0_Input_0.
;   6. Set SPIMVL's MOSI Parameter to Row_0_Output_0.
;   7. Set SPIMVL's SCLK Parameter to Row_0_Output_2.
;   8. Set SPIMVL's InterruptType Parameter to SPI Complete.
;   9. Set SPIMVL's ClockSync Parameter to Sync to SysClk.
;   10.Set SPIMVL's Data Length Parameter to 9 Bit.
;   11.Set SPIMVL's Invert MISO Parameter to Normal.
;   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_Output_0 and connect Row_0_Output_0 to GlobalOutEven_4.
;   15.Select GlobalOutEven_4 for P0[4] 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 SPIMVL.
;
; 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
;  ------------------------------------------------------------------------------
;  SPIMVL    Name               SPIMVL               UM's instance name
;            Clock              VC2
;            MISO               Row_0_Input_0
;            MOSI               Row_0_Output_0
;            SCLK               Row_0_Output_2
;            InterruptType   SPI Complete
;            ClockSync          Sync to SysClk
;            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)
 wDataTx: blk 2  ; data to transmit from
 wDataRx: blk 2  ; data to receive into

area text (ROM, REL)

_main:
    mov A, (SPIMVL_MODE_0|SPIMVL_MSB_FIRST)
    lcall SPIMVL_Start

    M8C_EnableGInt
    or reg[PRT0DR], 0x40     ; forse SS high first

.nextLoop:
       and reg[PRT0DR], ~0x40   ; set SS low before transaction

.WaitTx:
    lcall SPIMVL_bReadStatus
    and A, SPIMVL_TX_BUFFER_EMPTY
    jz .WaitTx               ; check for empty Tx buffer

    mov A, [wDataTx+1]
    mov X, [wDataTx]
    lcall SPIMVL_SendTxData  ; send data word

.WaitComplete:
    lcall SPIMVL_bReadStatus
    and A, SPIMVL_SPI_COMPLETE
    jz .WaitComplete         ; ensure transaction completed

    lcall SPIMVL_wReadRxData ; read received data from Rx buffer
    mov [wDataRx+1], A
    mov [wDataRx], X

    inc [wDataTx+1]          ; increment data before transmit
    adc [wDataTx], 0x00

    or reg[PRT0DR], 0x40     ; disassert SS signal

    jmp .nextLoop
```

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 SPIMVL, Register: FunctionMSB First Configuration:

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

This register defines the personality of this Digital Communications Type ‘A’ Block to be a SPIMVL User Module thus some bits are set accordingly to the user module properties.

Table 5. Block SPIMVL, Register: Input MSB First Configuration:

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

Table 6. Block SPIMVL, Register: Input LSB First Configuration:

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

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

Table 7. Block SPIMVL, Register: Output MSB First Configuration:

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

Table 8. Block SPIMVL, Register: Output LSB First Configuration:

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

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 9. Block SPIMVL, Shift Register: DR0

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

Shift Register is a SPI Shift register.

Table 10. Block SPIMVL, TX Data Buffer Register: DR1

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

Tx Buffer Register: Data written to this buffer will be transferred to the Shift register when the PSoC block is enabled.

Table 11. Block SPIMVL, RX Data Buffer Register: DR2

Bit	7	6	5	4	3	2	1	0
LSB	RX Buffer Register							
MSB	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 12. Block SPIMVL, Control0 Register: CR0

Bit	7	6	5	4	3	2	1	0
LSB	LSB First	RX Overrun Error	SPI_ COMPLETE	TX Buffer Empty	Rx Buffer Full	Clock Phase	Clock Polarity	SPIMVL Enable
MSB	LSB First	RX Overrun Error	SPI_ COMPLETE	TX Buffer Empty	Rx Buffer Full	Clock Phase	Clock Polarity	SPIMVL 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.

SPIMVL Enable enables the SPIMVL PSoC block when set.

Table 13. Block SPIMVL, Control1 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	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.
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.