Getting Started with SPI in PSoC® 1

Author: Todd Dust

Associated Project: Yes
Associated Part Family: CY8C21x23, 21x34, 21x45, 22x45, 23x33, `24x23A, 24x33, 24x94, 27x43, 28xxx, 29x66
Software Version: PSoC® Designer™ 5.4
Related Application Notes: AN32200

AN51234 discusses Serial Peripheral Interface (SPI) and how PSoC® 1 handles SPI communications. After reading this document, you should have an understanding of how SPI works and how it is implemented in PSoC 1.

Contents

1 Introduction

Serial Peripheral Interface (SPI) is a common chip-to-chip serial communications standard developed by Motorola. SPI provides a simple way for ICs to communicate on the same printed circuit board (PCB).

The Cypress PSoC 1 device offers several choices for implementing SPI in a design. These choices come in the form of user modules (UMs) that are located in the PSoC Designer™ IDE.

This application note first explains the basics of SPI and then gives a brief overview of the SPI UMs and their associated APIs. In addition, it explores special SPI considerations, such as SPI modes, multislave systems, 16-bit transfers, and interbyte delay.

This application note assumes that you are familiar with the PSoC 1 device and the PSoC Designer IDE. If you already have an understanding of SPI basics, it is recommended that you go directly to the section SPI in PSoC 1. If you are looking for help in troubleshooting your SPI design, go to the Special SPI Considerations section.

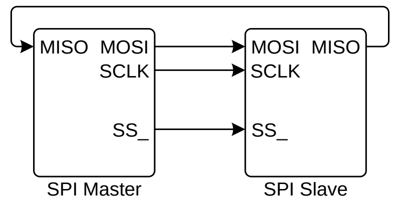
2 SPI Basics

2.1 The Physical Layer

SPI is a master/slave system with four I/O connections: MOSI, MISO, SS, and SCLK. Figure 1 shows the basic layout for a simple SPI bus.

- **MOSI:** The master out, slave in line, which carries data from the master to the slave
- **MISO:** The master in, slave out line, which carries data from the slave to the master
- **SS**: The slave select line, used to select and enable a slave on the bus
- **SCLK:** The clock signal that synchronizes the system

Figure 1. Two-Device Full-Duplex SPI System



There can be only one master on an SPI bus. The SPI master drives the clock signal at all times and is therefore in control of the bus at all times. Multiple slaves can exist on the same bus.

Both the SPI master and SPI slave modules act as shift registers. When the master wants to send a byte to the slave, it first loads a byte of data into its output shift register. The master then selects a slave as the destination. This is done by asserting the SS line associated with that slave. The SS line in an SPI system is active low, so asserting the line is accomplished by driving the signal low. The SCLK line is then enabled, and one bit of data is shifted out on the MOSI line with each clock pulse. The slave shifts in the bit of data on the MOSI line on the opposite clock edge.

In full-duplex mode, the slave shifts data out of the MISO line, and the master reads in the data. Some systems take advantage of this feature of SPI to increase the throughput of the system. For example, when the master sends a request for information to the slave, the slave can simultaneously send status information.

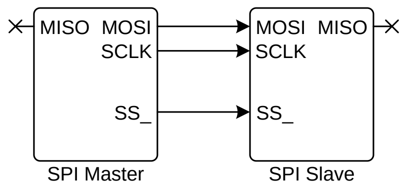
This means that for every pulse on the SCLK line, four actions take place:

- The master shifts out a bit on the MOSI line.
- The slave reads in a bit on the MOSI line.
- The slave shifts out a bit on the MISO line.
- The master reads in a bit on the MISO line.

Because the master always controls SCLK, it must initiate all clock pulses to read data from the slave. Reading data from a slave is often accomplished by shifting out “dummy” information to shift the data out of the slave into the master.

An SPI system can also be set up as a simplex system, as shown in Figure 2. In this configuration, data is sent only from the master to the slave. There are also SPI modules that use a half-duplex configuration, where data is shifted in each direction on a single line. In half-duplex systems, data can be sent in only one direction at time. The master still controls the SCLK in these configurations.

Figure 2. Two-Device Simplex SPI System Block Diagram



2.2 SPI Modes

Four modes of operation are defined for SPI systems. The numbering or naming of these modes may differ among vendors.

The modes are defined by the relationship between the clock and the sampling of the data. Most vendors use two parameters that the industry has accepted for these modes: CPOL and CPHA. CPOL is the clock polarity that determines whether the clock’s default state is high or low. CPHA is the clock phase that determines whether the data is sampled on the rising or falling edge of the clock. If the data is sampled on the rising edge of the clock, then the data is propagated, or shifted out, on the falling edge of the clock. You can mix and match these parameters to create four SPI modes.

Cypress defines the SPI modes as follows (see Table 1):

- Mode 0: CPHA = 0, CPOL = 0
- Mode 1: CPHA = 0, CPOL = 1

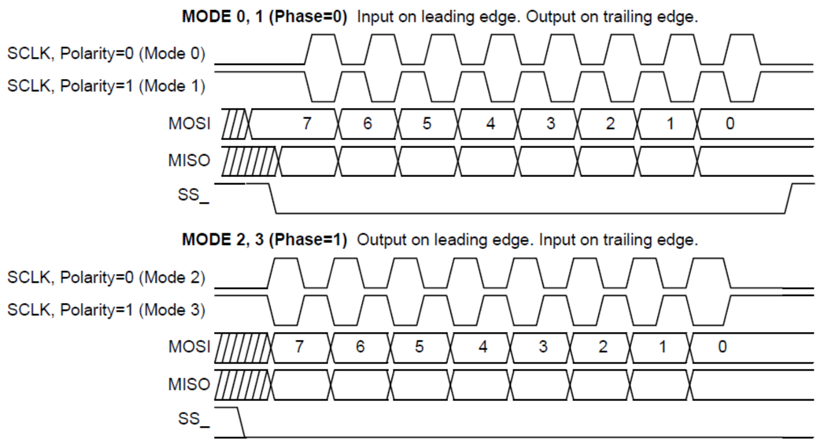
- Mode 2: CPHA = 1, CPOL = 0
- Mode 3: CPHA = 1, CPOL = 1

Cypress uses the terminology “leading edge” and “trailing edge” when defining these modes with respect to CPHA. For example, with CPOL set to 0, the default state of the clock is 0. With CPHA set to 0, you will sample data on the leading edge of SCLK that will be the rising edge of the clock as it transitions from 0 to 1 (mode 0). If CPHA were set to 1, you would sample on the trailing edge that would be the falling edge as the clock transitions from 1 to 0 (mode 2). Figure 3 depicts each mode.

Table 1. SPI Modes

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

Figure 3. SPI Timing by Mode

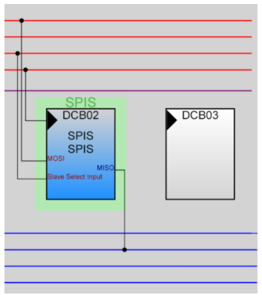

MODE 0, 1 (Phase=0) Input on leading edge. Output on trailing edge.
SCLK, Polarity=0 (Mode 0)
SCLK, Polarity=1 (Mode 1)
MOSI
7
6
5
4
3
2
1
0
MISO
SS_
MODE 2, 3 (Phase=1) Output on leading edge. Input on trailing edge.
SCLK, Polarity=0 (Mode 2)
SCLK, Polarity=1 (Mode 3)
MOSI
7
6
5
4
3
2
1
0
MISO
SS_

SPI provides a great solution for designers looking for serial communication with a simple hardware implementation and customizable protocol. However, SPI does not include any inherent error checking and uses more pins than other solutions such as I^{2}C. SPI can run at much faster rates than I^{2}C. These advantages and disadvantages must be considered when evaluating SPI as a possible serial communication solution.

3 SPI in PSoC 1

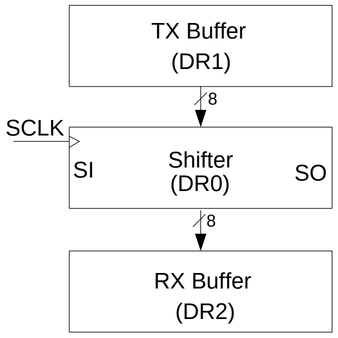
SPI is implemented on one of the digital communication blocks (DCBs) in PSoC 1 devices, as shown in Figure 4.

Figure 4. SPI in a DCB



When configured for SPI, the DCB's DR0 register acts as a shift register. All data is shifted in and out of this register. As shown in Figure 5 when a complete byte is received, the data is transferred from this shift register to the DR2 register, which acts as a one-byte RX buffer. In a similar fashion, DR1 acts as a one-byte TX buffer. All transmitted data is first loaded into this register. When the block wants to send data out, it loads the shift register (DR0) from this buffer.

Figure 5. SPI Functionality in Digital Blocks



The block generates three status signals to help determine the state of a transfer:

- **Byte Complete:** This status is triggered when a full byte of data has been clocked out. The status goes high after the edge of SCLK that shifts the last bit out. This status can also be used to trigger an interrupt.
- **RX REG FULL:** This status is triggered when the last bit of received data has been clocked in and the full received byte is loaded into the RX buffer. This status does not trigger an interrupt.
- **TX REG EMPTY:** This status is triggered when the TX register is empty. It also produces an interrupt. This status is useful for creating SPI masters with little to no interbyte latency.

Figure 6 and Figure 7 detail how the status bits are set for each mode.

Figure 6. Mode 0, 1 Status

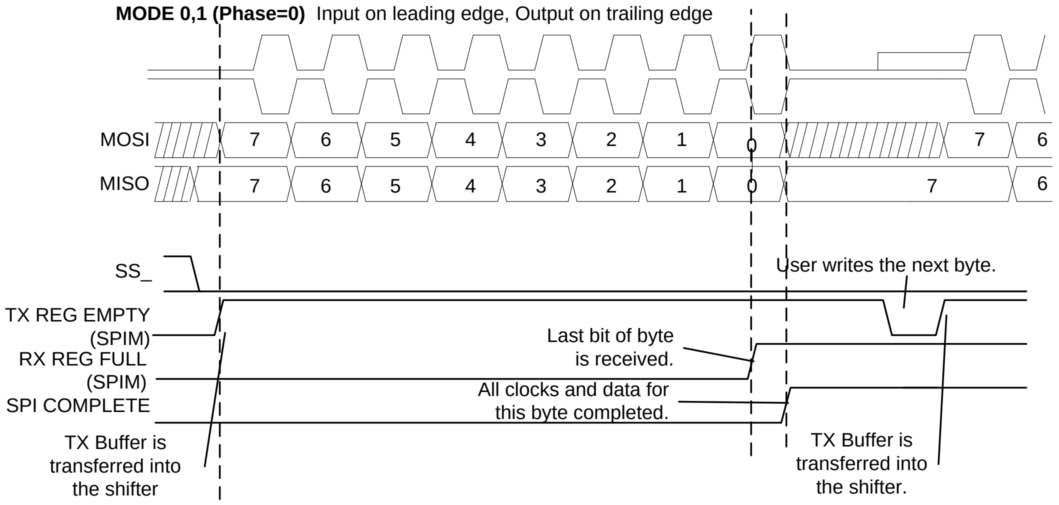
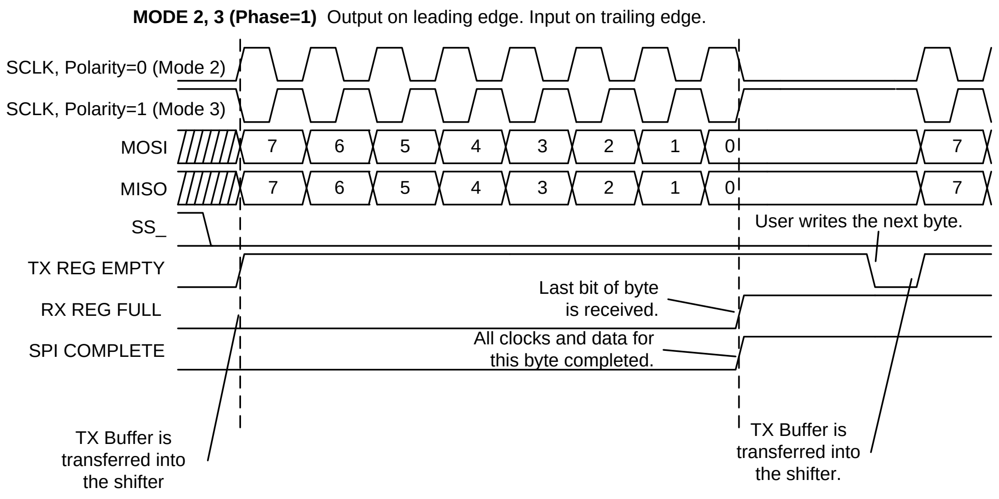

MODE 0,1 (Phase=0) Input on leading edge, Output on trailing edge
MOSI
MISO
7 6 5 4 3 2 1 0
SS_
TX REG EMPTY (SPIM)
RX REG FULL (SPIM)
SPI COMPLETE
User writes the next byte.
Last bit of byte is received.
All clocks and data for this byte completed.
TX Buffer is transferred into the shifter
TX Buffer is transferred into the shifter.

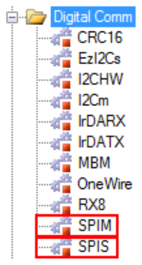
Figure 7. Mode 2, 3 Status


MODE 2, 3 (Phase=1) Output on leading edge. Input on trailing edge.
SCLK, Polarity=0 (Mode 2)
SCLK, Polarity=1 (Mode 3)
MOSI
MISO
7 6 5 4 3 2 1 0
SS_
TX REG EMPTY
RX REG FULL
SPI COMPLETE
User writes the next byte.
Last bit of byte is received.
All clocks and data for this byte completed.
TX Buffer is transferred into the shifter
TX Buffer is transferred into the shifter.

3.1 SPI User Modules

PSoC Designer offers two SPI UMs: one for an SPI master (SPIM) and one for an SPI slave (SPIS). Both of these modules are located in the **Digital Comm** category under **User Modules** in PSoC Designer, as shown in Figure 8.

Figure 8. SPI UMs



3.1.1 User Module Placement

The SPI UMs can be placed only in DCBs. In the chip view of PSoC Designer, these are the blocks that begin with the letters “DC” (for example, DCB02, DCB03, DCB12).

3.1.2 SPIM User Module

The SPIM UM configures the DCB as an SPI master.

Parameters

The SPIM UM has the following parameters to configure:

- **Clock:** This parameter selects the clock source for the digital block acting as the SPI master. The "AC Digital Block Specifications" section of each PSoC device datasheet lists a maximum value.

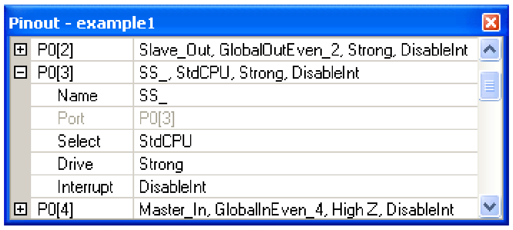
  Note The SCLK output signal has half the frequency of the input clock signal's frequency.
- **MISO:** This parameter selects the source of the data input signal from the SPI slave, if communication is full duplex.
- **MOSI:** This parameter selects the destination of the data output signal to the SPI slave.
- **SCLK:** This parameter selects the destination of the output data clock signal. Because of the clock division that occurs in the UM, the actual SCLK frequency is half the frequency of the "Clock" parameter selection.
- **Interrupt Mode:** This parameter selects one of the two interrupt generation conditions. TxRegEmpty mode generates an interrupt when the byte in the TX buffer has been moved over to the shift register for transmission. TxComplete mode generates an interrupt when the data for transmission is completely shifted out of the shift register.

  Note SPIS_EnableInt or SPIM_EnableInt must be called and global interrupts must be enabled in the software for the interrupts to be generated.
- **ClockSync:** This parameter selects one of the four possible clock synchronizing options. These options dictate when the digital block clocks in data. The most common selection is "Sync to SySCLK." For more details, see application note AN32200.
- **InvertMISO:** This parameter allows the input data to be inverted before being processed by the SPIM UM.

  1. Note The SPIM UM does not have a parameter for an SS_ output. A separate port pin must be assigned as the SS_ output and controlled in the software.

  An SS signal is created in the chip view of PSoC Designer by designating the port pin of choice as "StdCPU" with a "Strong" drive. Figure 9 provides an example.

Figure 9. Example SS_ GPIO Configuration


Pinout - example1

⊞ P0[2]	Slave_Out, GlobalOutEven_2, Strong, DisableInt
⊟ P0[3]	SS_, StdCPU, Strong, DisableInt
Name	SS_
Port	P0[3]
Select	StdCPU
Drive	Strong
Interrupt	DisableInt
⊞ P0[4]	Master_In, GlobalInEven_4, High Z, DisableInt

API

The SPIM UM provides the following APIs to read and write data.

`SPIM_SendTxData(data);`

This API loads data into the TX buffer. The status must be checked before calling this function to ensure that the previous data in the buffer does not get overwritten. Code to do so appears as follows.

```
if(SPIM_bReadStatus()&SPIM_SPIM_TX_BUFFER_EMPTY)
{
          SPIM_SendTxData(0x00);
}
```

`SPIM_bReadRxData();`

This API directly reads data out of the RX buffer. Again, the status must be checked before calling this API to ensure that there is valid data in the buffer.

```
if(SPIM_bReadStatus()&SPIM_SPIM_RX_BUFFER_FULL)
{
```

```
            SPIM_bReadRxData(0x00);
}
```

3.1.3 SPIS User Module

The SPIS UM configures the DCB as an SPI slave.

Parameters

The SPIS UM has the following parameters to configure:

- **SCLK:** This parameter selects the source of the input clock from the SPI master. The "AC Digital Block Specifications" section of each PSoC device datasheet lists a maximum value.
- **MOSI:** This parameter selects the source of the data input signal from the SPI master.
- **Slave Select Input:** This parameter selects the source of the SS_ input signal from the SPI master.
- **MISO:** This parameter selects the destination of the data output signal to the SPI master, if the communication is to be full duplex.
- **Interrupt Mode:** This parameter selects one of the two interrupt generation conditions. It has the same functionality as in the SPIM UM.
- **Invert MOSI:** This parameter allows the input data to be inverted before being processed by the SPIS UM.

Note The SPIS UM does not have a clock input other than the SCLK. This is because the SPIS digital block must be clocked by the SCLK of the SPI master to synchronize properly.

The SPIS and SPIM UM datasheets discuss these parameters in detail.

API

The SPIS UM provides the following APIs to read and write data.

`SPIS_SetupTxData(bTxData);`

This API loads data into the TX buffer. When the master asserts the SS line or starts clocking, the data in the TX buffer is loaded into the shift register. Similar to with SPIM, the status must be checked before calling this API to ensure that previous data in the TX buffer is not overwritten.

```
if(SPIS_bReadStatus()&SPIS_SPIS_TX_BUFFER_EMPTY)
{
            SPIS_SetupTxData(0x00);
}
```

`SPIS_bReadRxData();`

This API is exactly the same as the SPIM API. It reads data out of the RX buffer. Again, the status needs to be checked before reading data out of this buffer.

```
if(SPIS_bReadStatus()&SPIS_SPIS_RX_BUFFER_FULL)

{

SPIS_bReadRxData(0x00);

}
```

3.2 I/O Routing

The SPIM UM requires three signals to be routed: two outputs (one for SCLK, one for MOSI) and one input (MISO, if full duplex). If the SPI master needs to supply slave select (SS_), an additional output must be reserved.

The SPIS UM requires three inputs (SCLK, MOSI, SS_) and has one optional output (MISO).

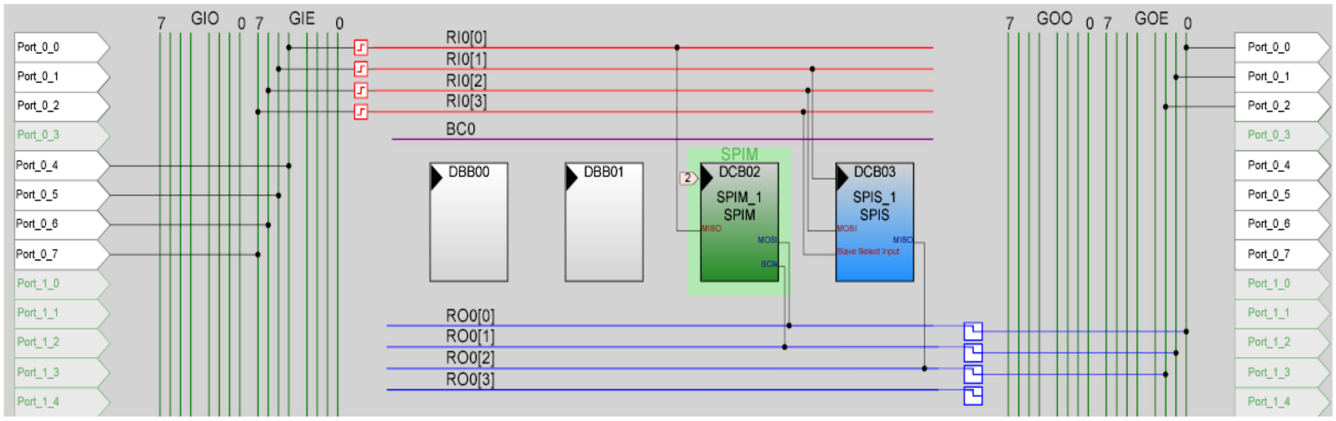
Figure 10 shows the routing of a "loopback" system that uses both the SPIM and SPIS UMs on a single PSoC device. To create this system, you add wires externally to connect the SPI master and slave together.

This connection scheme is as follows:

- P0[0] to P0[6] MOSI
- P0[1] to P0[5] SCLK

- P0[2] to P0[4] MISO
- P0[3] to P0[7] SS

Figure 10. Example PSoC Designer Routing of a “Loopback” System



4 Special SPI Considerations

On the surface, an SPI system appears relatively straightforward. This makes it especially frustrating when the circuit is built, the software is written, and nothing happens. A communication failure between master and slave can have many different sources, and this section is intended to uncover the slight degree of difference that can cause such a breakdown. Also covered are other special considerations such as communicating to multiple slaves with one master and communicating with devices with greater than 8-bit SPI interfaces.

4.1 SPI Modes

Master and slave devices must operate in the same mode. If the correct modes are not chosen, errors can range from intermittently corrupted data to absolute communication failure. The SPI UMs in PSoC Designer are configurable for four different SPI modes. The mode is set when calling the SPIM_Start or SPIS_Start API functions. For a complete description of the SPI modes, see the "SPI Mode Timing" section in the *PSoC Technical Reference Manual* or the previous section on SPI Modes.

Note Different manufacturers have different definitions for SPI mode numbering. If there is any question regarding the correct modes between a master and slave, you must observe the SCLK, MOSI, and MISO lines with an oscilloscope and experimentally verify them.

4.2 Device Differences

Different PSoC devices have different capabilities. This creates differences in the way SPI can be implemented on these parts.

4.2.1 Device Clock Frequency Limitations

While most PSoC parts have the same maximum SPIM input clock frequency and SPIS SCLK frequency, there are some differences between device families. In addition, the V_{DD} level affects the maximum clock frequencies. Because of these differences, you need to consult the "AC Digital Block Specifications" section of the device datasheet for the proper SCLK and input clock frequencies.

4.2.2 CY8C27x43 Device Shift Register Limitations

When the CY8C27x43 family of PSoC is used as a "link" in a "daisy chain" of multislave SPI configurations, it is operating as a shift register. This device needs special attention when used in this capacity. If no special action is taken in firmware, it shifts through seven of the eight bits contained in a transmitted byte. See the Errata Summary in the CY8C8C27x43 Family datasheet for more information.

The problem occurs when the CY8C27x43 SPIS receives data but has no data to transmit in its TX register. The workaround involves reading the received SPI data with an interrupt service routine and writing it back to the TX register, as shown in the following code. See the Errata Summary in the CY8C8C27x43 Family datasheet for more information regarding the workaround.

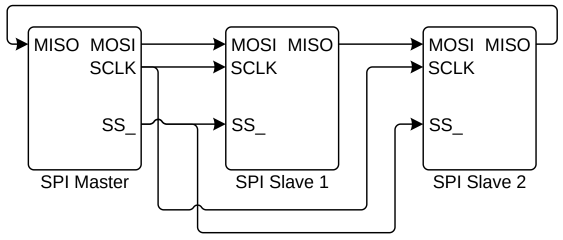
```
SPIS_Int:
// Read the received byte
mov A, reg[SPIS_RX_REGISTER]
// Write it to the TX register
mov reg[SPIS_TX_REGISTER], A
// Clear SPI Complete status
mov A, reg[SPIS_CTRL_REGISTER]
reti
```

4.3 Multiple Slave System

4.3.1 Daisy Chaining Slave Devices

One method of connecting multiple slaves to one master is to daisy chain the slaves. In this method, the output of one slave connects to the input of another; see Figure 11.

Figure 11. Daisy-Chained Multiple Slave System



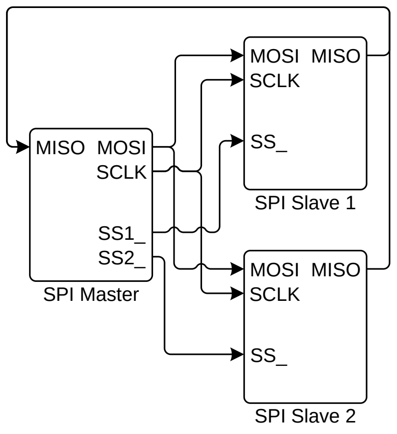
This system relies on the fact that most SPI slave devices (including Cypress's) behave as shift registers. Shift registers hold a number of bytes of data, and once every clock cycle they "shift" all bits to one place, left or right. After the master brings SS_ low, it clocks data through the first slave device, which propagates the data to additional slaves. For example, if the two slaves shown in Figure 11 are 8-bit devices, a total of 16 bits must be clocked out by the master for the first 8 bits to reach SPI slave 2.

The advantage of the daisy chain method is that the data output of one SPI device connects directly to the input of another device, and there is no chance of output contention. Furthermore, only one SS_ line is needed to control all SPI slaves. The disadvantages of the daisy chain method are that all SPI slaves must be written to if only one needs to be accessed, and one extra byte must be written by the master to transmit through each additional slave in the chain.

4.3.2 Slave Devices in Parallel

Another method of controlling multiple SPI slave devices is to place the data lines in parallel, as shown in Figure 12. In this figure, the SPI master must provide a separate SS_ line to each of the SPI slaves to select the slave to which the master talks. This scenario is allowed because the SPI slaves ignore any SCLK and MOSI transitions unless the SS_ line is driven low.

Figure 12. Parallel Multiple-Slave System



Also note in Figure 12 the data outputs of all SPI slaves that are tied into a single point at the SPI master input. Ensure that the outputs of two slaves are not driven at the same time. The SPIS UM does not automatically set the MISO pin to the high impedance drive mode.

To prevent bus contention, set the output of each slave to a high impedance state whenever the slave is not selected. When a particular slave is selected and the master requests data from it, the output of the slave must be set to a strong drive before it transmits data. To accomplish this in PSoC Designer, modify in the software the port drive mode registers corresponding to the MISO output to switch between high impedance and strong drive as follows.

```
/*Change MISO on P0.1 from Strong to High-z*/
PRT0DR &= ~ 0x01; /*Set the Drive to low*/
PRT0DM2 |= 0x01; /*Set Drive Mode to Slow Strong*/
PRT0DM1       |= 0x01; /*Set Drive Mode to Open Drain*/
PRT0DM0 &= ~0x01; /* Set Drive Mode to High-z*/
```

```
/*Change MISO on P0.1 from High-z to Strong*/
PRT0DR &= ~ 0x01; /*Set the Drive to low*/
PRT0DM2  &= ~0x01; /*Set Drive Mode to Pullup*/
PRT0DM1       &= ~0x01; /*Set Drive Mode to Pulldown*/
       PRT0DM0 |= 0x01; /* Set Drive Mode to Strong*/
```

This mechanism uses software and therefore has latency. This latency has to be compensated on the master side by adding a delay between the time the SS_ is asserted and the data is transmitted.

The advantage of the parallel device method is that the extra bytes are not needed to clock data to all devices. The disadvantages are the danger of signal contention between devices and the extra pin requirement for each slave select line.

4.4 Communicating With 16-Bit Slaves

Many SPI slave devices require a total of 16 data bits to be clocked in, where the first byte is a command byte and the second byte is a data byte. This section describes how to communicate with a 16-bit slave with the 8-bit SPIM module.

The key to using the 8-bit SPIM module with 16-bit slaves is proper use of the SS_signal. To write a total of 16 bits, the SS_signal must be asserted low in software, and the SPIM_SendTxData API function must be called twice before deasserting SS_. The first instance of SPIM_SendTxData sends the higher order byte, and the second instance sends the lower order byte.

Make sure with firmware that the TX buffer is not overwritten by the second byte. This can be carried out using the TxRegEmpty mode of interrupt generation to automatically insert the second byte when the first is moved to the shift register. The SPI master's status can also be polled using the SPIM_bReadStatus API.

Reading from a 16-bit slave device is handled in a similar fashion to writing. The SS_ signal must first be asserted low and a total of 16 bits must be clocked out of the slave before deasserting SS_.

Note that the SPIM_bReadRxData API function reads only data from the master's RX buffer and does not actually clock data out of the slave. To clock data out of the slave, the SPIM_SendTxData API function needs to be called to shift data out of the slave and into the master's RX buffer. For a total of 16 bits to be read, the master calls SPIM_SendTxData, polls the status until data is available, calls SPIM_bReadRxData to retrieve the higher order byte followed by SPIM_SendTxData, polls the status again, and finally calls SPIM_bReadRxData to retrieve the lower order byte. SS_ must be asserted low during this whole sequence of commands.

```
SPIM_SendTxData(0xff);

while(!(SPIM_bReadStatus() & SPIM_SPIM_RX_BUFFER_FULL));

cRXData = SPIM_bReadRxData();

SPIM_SendTxData(0xff);

while(!(SPIM_bReadStatus() & SPIM_SPIM_RX_BUFFER_FULL));

       cRXData1 = SPIM_bReadRxData();
```

4.4.1 9- to 16-Bit Transfers in the 28xxx, 22x45, and 21x45

These devices have enhanced digital communication blocks (DCCs) that allow for the chaining of SPIM and SPIS UMs to achieve up to 16-bit automatic transfers. This chaining is easily accomplished by using the SPIMVL and SPISVL UMs.

4.5 Voltage Levels

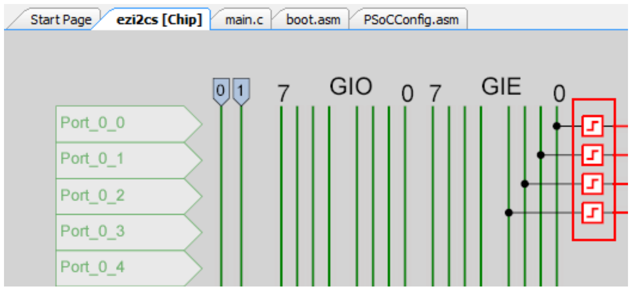
For two or more devices to communicate properly, they must be signaling at the same voltage levels. This means that when device A signals logic high, device B must interpret it as logic high. When two devices have different supply voltages, problems can occur in signaling. Each PSoC device interprets input voltage as described in the "DC General Purpose I/O" section of the device family's datasheet. The ground voltages (V_{SS}) of the devices must be the same.

The easiest way to ensure that voltages are signaled properly is to design your system such that the same voltage rails supply both devices. If this cannot be done, then the specifications for V_{IH}, V_{OH}, V_{IL}, and V_{OL} for the devices must be considered. For most PSoC devices, the GPIO V_{IH} is compatible with outputs from 5 V and 3.3 V V_{DD}. For other V_{OH} levels, a level translation is needed.

4.6 Input Synchronization

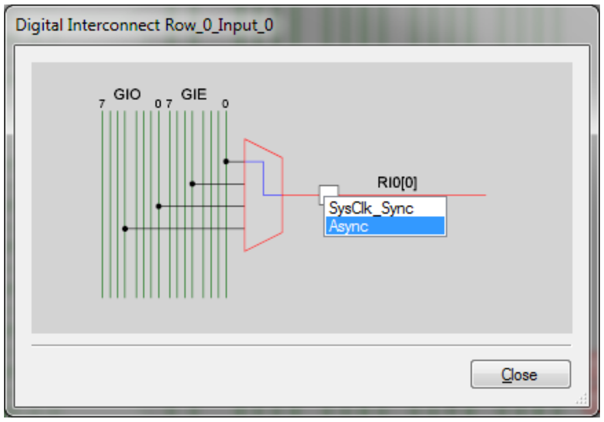
When operating SPI at frequencies greater than 1 MHz, it is recommended not to synchronize SPI inputs to the PSoC SysClk. By default, the inputs to the PSoC are always synchronized, as shown in Figure 13.

Figure 13. Synchronized Input



The symbols boxed in red on the right side of the figure indicate that the inputs are synchronized. To unsynchronize the inputs, click on the ⎍ symbol. When the window shown in Figure 14 appears, click on the same symbol and select **Async**.

Figure 14. Unsynchronizing an Input



For SPIS operation, the SS, MOSI, and SLCK signals should be set to **Async**.

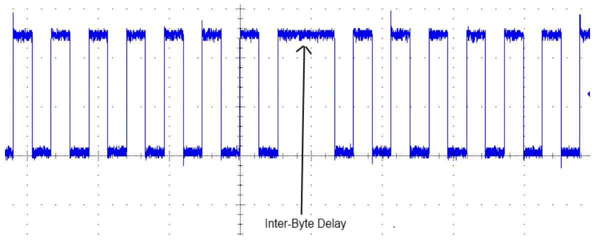
For SPIM operation, the MISO signal should be set to **Async**.

If the signals are left synchronized to **SysClk,** then the inputs signals will be delayed, which will affect system performance.

4.7 Interbyte Latency

When operating as a master, it is often desirable to have little to no latency between bytes of data that are sent to the slave. This latency, shown in Figure 15, is caused by instruction delays between calling `SPIM_SendTxData(0xff);`

Figure 15. Interbyte Delay



To overcome this issue, it is recommended to use the TxBufferEmpty interrupt. Inside the interrupt, load the next byte in the buffer. Also inside the interrupt, read the status by calling `SPIM_bReadStatus()`.

If the status is not read inside the interrupt, the status signal will remain high and the interrupt will not fire again.

4.8 Reading Data From a Slave

The SPIM_bReadRxData() API does not read data out of a slave device. It only reads data out of the master's RX register. Data needs to be present in this register before calling SPIM_bReadRxData().

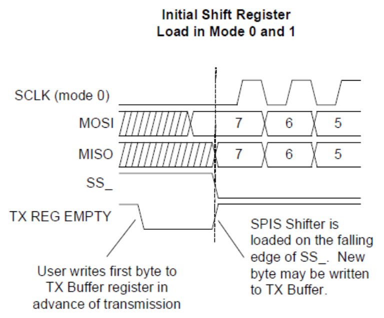
To read data out of the slave, a byte of data needs to be shifted into the slave. The only way to do this is by calling SPIM_SendTxData(0x00). Oftentimes, a value of 0x00 or 0xff is written out to the slave, which is called a "dummy transfer." As the "dummy" data is shifted into the slave, it shifts data out of the slave to the master.

4.9 Queuing Data in a Slave

Most SPI applications call for data to be sent back from the slave to the master. In this case, it is important to know when data is transferred from the slave TX buffer to the shift register. This section explains this operation for the different modes.

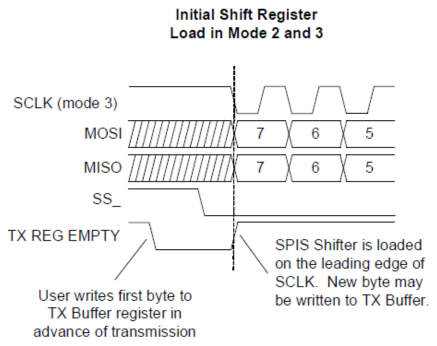
First, you need to know when data is initially loaded into the shift register from the TX buffer. In modes 0 and 1, the assertion of SS transfers the contents from the TX buffer into the shift register, as shown in Figure 16.

Figure 16. Initial Shifter Load for Mode 0 and 1


Initial Shift Register
Load in Mode 0 and 1
SCLK (mode 0)
MOSI
MISO
SS_
TX REG EMPTY
7
6
5
7
6
5
User writes first byte to TX Buffer register in advance of transmission
SPIS Shifter is loaded on the falling edge of SS_. New byte may be written to TX Buffer.

For modes 2 and 3, the data is loaded on the leading edge of SCLK, as shown in Figure 17.

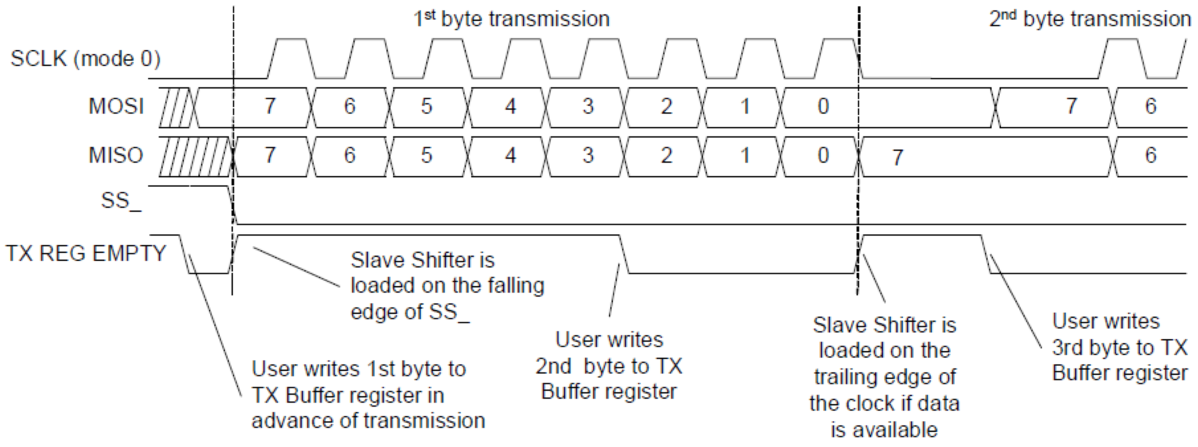
Figure 17. Initial Shifter Load for Modes 2 and 3


Initial Shift Register
Load in Mode 2 and 3
SCLK (mode 3)
MOSI
MISO
SS_
TX REG EMPTY
7
6
5
7
6
5
User writes first byte to TX Buffer register in advance of transmission
SPIS Shifter is loaded on the leading edge of SCLK. New byte may be written to TX Buffer.

After the initial load, it is important to understand how subsequent bytes are loaded into the shift register. The behavior of this loading differs among the different modes and differs depending on whether SS is kept low or toggled between bytes.

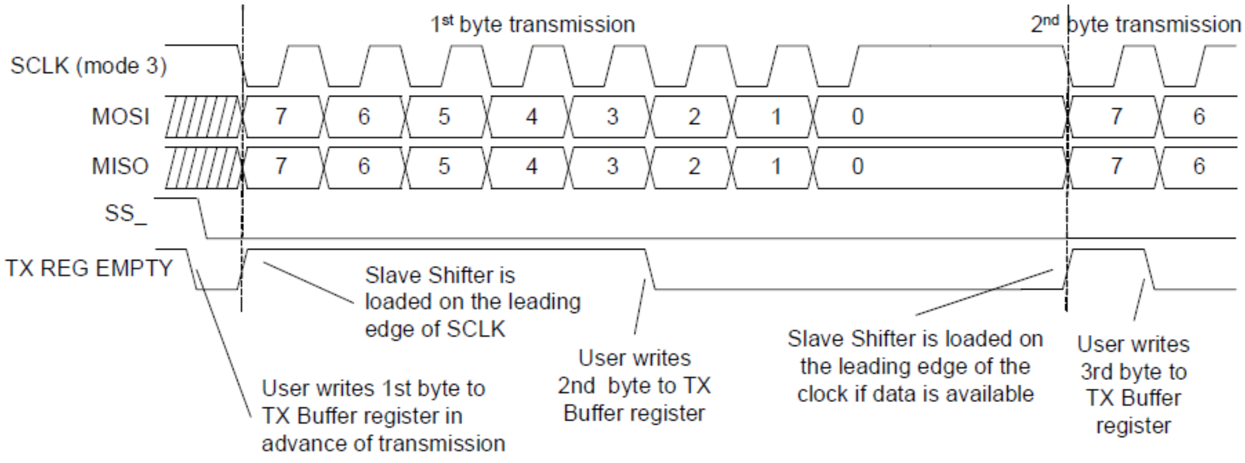
In modes 0 and 1, if SS is kept low, the shifter is loaded on the trailing edge of the last clock, as shown in Figure 18.

Figure 18. Shifter Loads in Mode 0 and 1 Without SS


1st byte transmission
2nd byte transmission
SCLK (mode 0)
MOSI
MISO
SS_
TX REG EMPTY
Slave Shifter is loaded on the falling edge of SS_
User writes 1st byte to TX Buffer register in advance of transmission
User writes 2nd byte to TX Buffer register
Slave Shifter is loaded on the trailing edge of the clock if data is available
User writes 3rd byte to TX Buffer register

If SS is toggled between bytes in modes 0 and 1, the next byte is not transferred from the TX buffer into the shifter register until SS is asserted, as shown in Figure 16. Note that if there is data in the TX buffer and SS is not released before the trailing edge of the clock, data will be transferred from the TX buffer to the shifter. In modes 2 and 3, the shifter is always loaded on the leading edge of the clock, as shown in Figure 17, regardless of the SS line. Figure 19 provides a complete view.

Figure 19. Shifter Loads in Modes 2 and 3


1st byte transmission
2nd byte transmission
SCLK (mode 3)
MOSI
MISO
SS_
TX REG EMPTY
Slave Shifter is loaded on the leading edge of SCLK
User writes 1st byte to TX Buffer register in advance of transmission
User writes 2nd byte to TX Buffer register
Slave Shifter is loaded on the leading edge of the clock if data is available
User writes 3rd byte to TX Buffer register

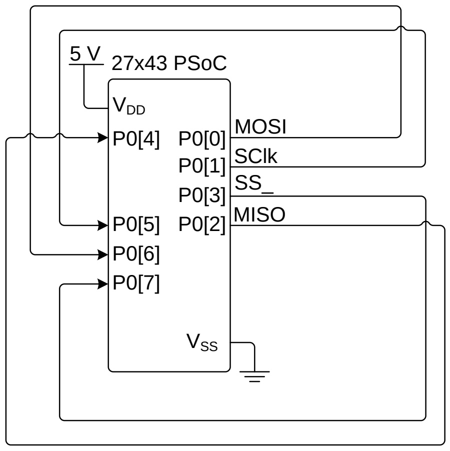
To send correct data back to the master at the specific time, use the information presented to determine when to load the TX buffer.

5 Example Projects

Four example projects are included with this application note.

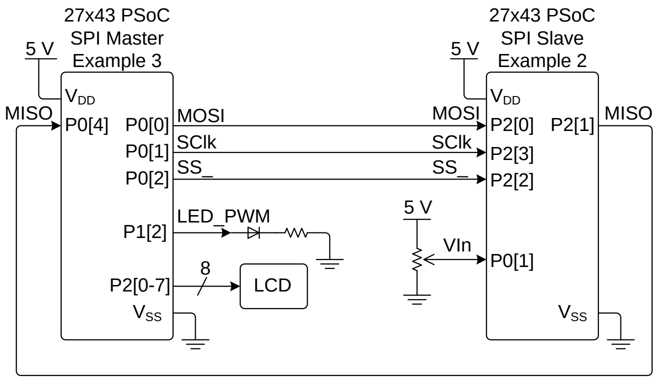
Example 1, shown in Figure 20, is the loopback example that contains both an SPI master and an SPI slave running on the same PSoC device. It can be observed running on a CY8C27x43 on-chip debug (OCD) PSoC device attached to a CY3250 in-circuit emulator. Figure 10 shows the internal routing of Example 1.

Figure 20. Physical Configuration for Example 1



Examples 2 and 3, shown in Figure 21, demonstrate an SPI master and slave system implemented on two separate PSoC devices. Example 2, the slave, measures an analog signal from pin P0[1] and communicates the 8-bit measurement through SPI.

Figure 21. Physical Configuration for Examples 2 and 3



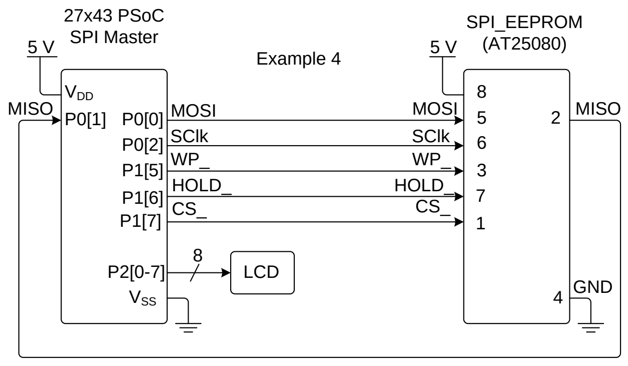
Example 3, the master, polls the slave through SPI and then updates the duty cycle of the PWM output to P1[2], which controls the brightness of the LED. Example 3 also displays the SPIM UM's status to an attached LCD. These examples may be run on separate CY8C27x43 PSoC devices in CY3210-EVAL1 boards.

For proper functioning of these examples, it is important for both PSoC devices to be operating at the same supply voltage and ground potential. If two separate EVAL1 boards are used, the grounds of the two boards must be connected.

In all examples, the SPI master and slave are connected in the same full-duplex manner as shown in Figure 1.

In Example 4, shown in Figure 22, an SPI communication is set up between PSoC 1 and an SPI EEPROM (AT25080A), with the SPIM module in PSoC 1 as the master and the external EEPROM as the SPI slave. An array of 32 bytes is written to the EEPROM starting from location 0 to 31, and the same array is read back from the EEPROM.

Figure 22. Physical Configuration for Example 4



6 Reference Documents

- *PSoC Technical Reference Manual*, available at **Help > Documents** in PSoC Designer or at http://www.cypress.com/?rID=34621
- Errata Summary provided in the Cy8C27x43 family datasheet.

About the Author

Name:	Todd Dust
Title:	Applications Engineer Senior
Background:	BSEE Seattle Pacific University

Document History

Document Title: AN51234 – Getting Started with SPI in PSoC® 1

Document Number: 001-51234

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2653106	MAXK/PYRS	02/03/2009	New Application Note
*A	3150325	MAXK	01/21/2011	Updated firmware to PD 5.1, added user module API details, updated title and abstract
*B	3441073	TDU	11/17/2011	Updated Structure to match AN50987. Did significant rewrite of all sections.
*C	3817347	DESH	11/20/2012	Updated firmware to PD 5.3. Added “example project 4” for interfacing PSoC 1 with external SPI EEPROM. Updated Figure 6 and Figure 22. Updated template.
*D	4297936	ASRI	03/04/2014	Updated firmware to PD 5.4 Minor content updates throughout the document
*E	4682694	DIMA	03/10/2015	Removed reference to Errata document and added reference to Errata Summary provided in device datasheet. Sunset update
*F	5688181	AESATMP8	04/19/2017	Updated logo and Copyright.
*G	6396088	DIMA	11/28/2018	Updated template

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

Cypress Developer Community

Community | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709