1300 Henley Court
Pullman, WA 99163
509.334.6306
www.store.digilent.com

Lab 4c: Communications – SPI Serial Protocols

Revised April 25, 2017
This manual applies to Unit 4, Lab 4c.

1 Objectives

1. Learn how to configure an SPI channel on the PIC32MX370 processor.
2. Learn the difference between software and hardware handshaking.
3. Learn how to communicate with an SPI FLASH memory device.

2 Basic Knowledge

1. How to configure I/O pins on a Microchip PIC32 PPS microprocessor.
2. How to configure the Analog Discovery 2 to display logic traces.
3. How to implement code reuse that integrates previously developed processor code into new application projects.

3 Equipment List

3.1 Hardware

1. Basys MX3 trainer board
2. Workstation computer running Windows 10 or higher, MAC OS, or Linux
3. 2 Standard USB A to micro-B cables

In addition, we suggest the following instruments:

4. Analog Discovery 2

3.2 Software

The following programs must be installed on your development work station:

1. Microchip MPLAB X® v3.35 or higher
2. PLIB Peripheral Library
3. XC32 Cross Compiler
4. WaveForms 2015 (if using the Analog Discovery 2)
5. PuTTY Terminal Emulation

4 Project Takeaways

1. Understanding of requirements and implementations of synchronous communications.
2. Understanding of the SPI protocol.
3. How to generate instruction sets for controlling SPI devices.

5 Fundamental Concepts

Serial Peripheral Interface (SPI) is a master-slave interface bus commonly used to send data between microcontrollers and small peripherals such as analog-to-digital converters, instrumentation sensors, and solid state memory devices. It uses a separate clock, send and receive data lines, and a device select signal. The PIC32MX370 has built-in hardware circuits to support two SPI channels. Since SPI and I^{2}C have been used in similar applications, there is frequently a comparison of the two protocols, such as presented in Reference 4.

5.1 Software Handshaking

We can see the application of hardware handshaking for synchronizing data transfers for both the asynchronous UART that used start and stop bits, and synchronous I^{2}C communications that uses the ACK bit. Software handshaking involves exchanging data that indicates the status of slave devices. An example of software handshaking with parallel I/O is polling the LCD busy flag. When using the UART, there is the XON/XOFF handshaking that is used for information flow control.

We will see that the flash memory device used on the Basys MX3 board requires command strings to place the device in different operating modes and to read device internal registers for determining status. Since SPI is a synchronous communications protocol, the clock signal manages the hardware element of the device synchronization.

5.2 SPI Communications

The SPI serial protocol is capable of higher data rates than I^{2}C because it can generally operate at higher clock rates, and is not limited to 8-bits per word. Although I^{2}C requires only two wires (thus conserving processor pins), rather than four wires required by SPI, I^{2}C has bandwidth overhead due to the time required for device selection by sending the ID as a serial byte. Unlike I^{2}C, SPI has no device acknowledge capability.

SPI is a full-duplex synchronous serial communications bus protocol developed by Motorola and has become a de facto standard that has not been adopted by any national or international standards organizations. As with the I^{2}C protocol, the SPI bus implements a master-slave communications scheme where the master device alone controls the data exchange with slave devices. The master device has exclusive control of the serial clock (SCK) signal that is used to clock the data to and from a slave device.

SPI requires four wires for full-duplex operation and supports only one master but multiple slave devices. The master writes data to the slave using the Master Out Slave In (MOSI) line. The slave devices are able to send data to the master over the Master In Slave Out (MISO) line. Other than the clock signal, all handshaking is handled by an explicit slave select (SS) or chip select (CS) signal.

Figures 5.1 and 5.2 illustrate the two common SPI bus connection configurations. Figure 5.1 shows that the multiple slave devices share SCK, MOSI, and MISO signals. For this connection configuration, the slave devices are explicitly selected by multiple dedicated SS microprocessor outputs. This is the more common SPI connection configuration.

Figure 5.2 shows a daisy-chain configuration where slave devices share both the SCK and SS signals, and the MOSI and MISO signals are routed through a series of slave devices. Using this configuration, data intended for the last device in the chain must be clocked through the preceding slave devices. Data that is to be read from the first slave device in the chain must also be clocked through the slave devices that follow it in the chain. This additional data transfer time is the cost of conserving processor I/O pins used for enabling slave devices. Due to the excessive data transfer time for systems with many slave devices, this configuration is seldom used.

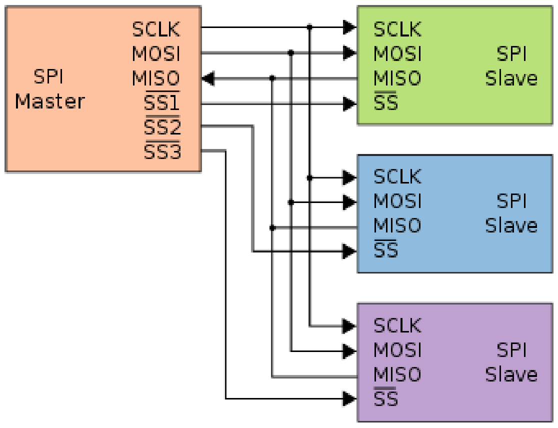


Figure 5.1. Parallel multiple slave SPI bus configuration with individual device select signals.

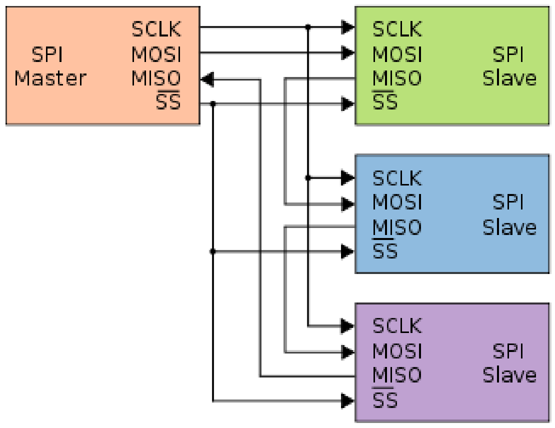


Figure 5.2. Daisy chain multiple slave SPI bus configuration with a common device select signal.

When the microprocessor is connected to a slave device that has both input and output capability using SPI, as data is clocked out of the master, data is also clocked in from the slave device. This results in efficient data transfers for some slave devices. The loosely defined SPI interface requires careful consideration of the slave clock-data timing, as well as using a microprocessor that can be configured to support various timing requirements. Figure 5.3 shows the clock-data timing for the four SPI operating modes. The clock polarity (CPOL) controls the idle level of the SCK output from the master. If CPOL is high, then the idle level of SCK is high. The clock phase (CPHA) specifies when the data is to be changed or written to MOSI by the master and to MISO by the slave device. For example, when CPHA is low, the data signal (MISO for master and MOSI for the slave) is sampled by the receiving device when SCK makes a transition from the idle level to the active level.

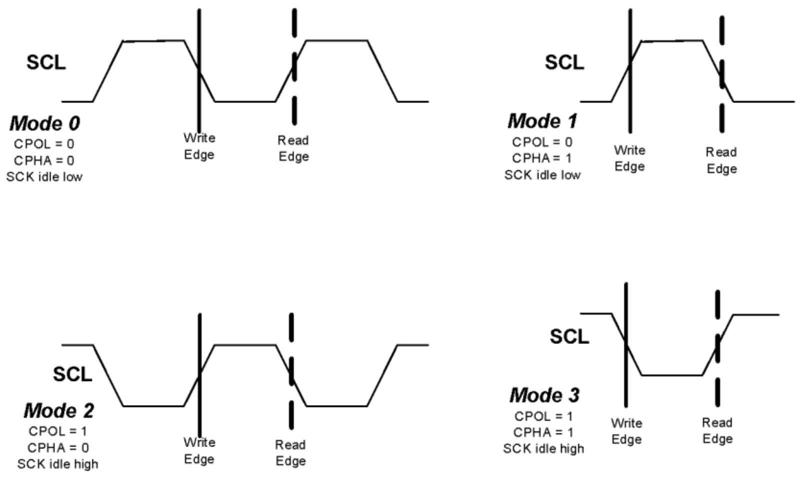
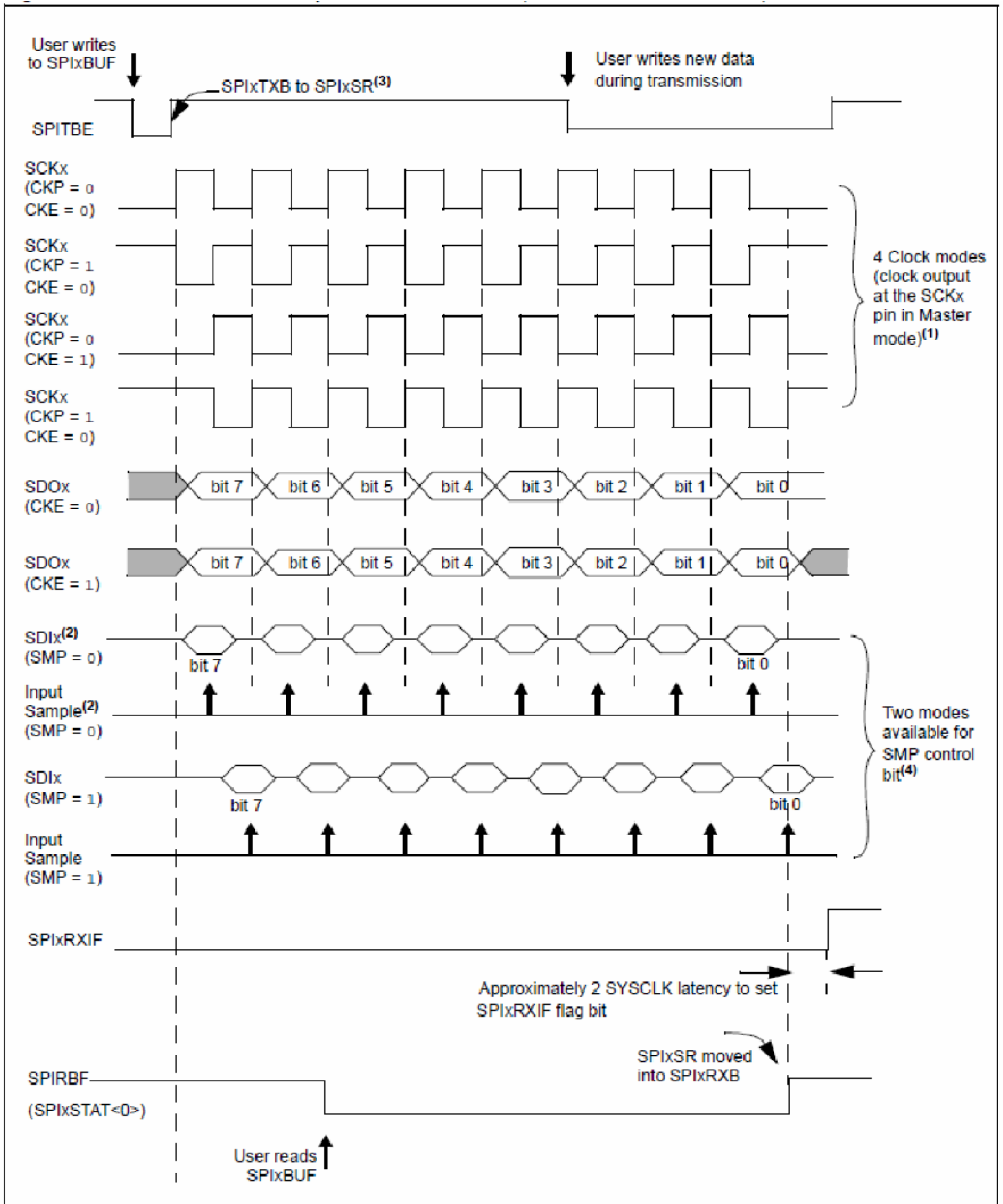


Figure 5.3. SPI timing modes.

Figure 5.4 shows the PIC32 settings for clock edge (CKE) and clock polarity (CKP). Table 5.1 provides the correlation between the conventional definitions of SPI mode CPHA and CPOL to those CKE and CKP settings. (Note: There is an apparent error in the Microchip table shown in Fig. 5.4. Both CKE and CKP should be equal to 1 for the fourth case of the SCK timing diagrams.) It is important to match the master processor operation to what the slave device is expecting. For some designs, it is possible for different slave devices to expect the master to operate in modes that are not the same. The PIC32 operating modes can be changed during program execution, but the modes should not be changed when the processor is actively clocking data on the SPI bus.

Table 5.1. SPI SCK operational modes.

SPI Mode	Active Level	Sample Transition	CPOL	CPHA	PIC32 CKP	PIC32 CKE
0 (or 0,0)	High	Idle to Active	0	0	0	1
1 (or 0,1)	High	Active to Idle	0	1	0	0
2 (or 1,0)	Low	Idle to Active	1	0	1	1
3 (or 1,1)	Low	Active to Idle	1	1	1	0



Note 1: Four SPI Clock modes are shown here to demonstrate the functionality of bits CKP (SPIxCON<6>) and CKE (SPIxCON<8>). Only one of the four modes can be chosen for operation.
2: The SDI and input samples shown here for two different values of the SMP bit (SPIxCON<9>) are strictly for demonstration purposes. Only one of the two configurations of the SMP bit can be chosen during operation.
3: If there are no pending transmissions, SPIxTXB is transferred to SPIxSR as soon as the user writes to SPIxBUF.
4: Operation for 8-bit mode shown. 16-bit and 32-bit modes are similar.

Figure 5.4. PIC32 SPI control setting to specify the sample timing (Reproduced from Microchip PIC32 Family Reference Guide Section 23, Fig. 23-7).

SPI master-slave communications can be operated in either simplex or full-duplex modes. Simplex communications occur when the slave device can only send or receive data. Regardless of which device, master or slave, is sending the data, the master always provides the SCK signal. Although the phase and sample timing can differ between devices using SPI, the data is always active high (a high level represents a logical 1.)

Data is sent and received by transferring the most significant bit (MSB) on the first clock pulse. We will define a transfer transaction as the exchange of data while the SS signal is continuously asserted in the active state (usually a low level.) For a specific transaction, if multiple bytes are to be transferred, the byte counter in the slave device is reset when the SS signal is asserted. There is no start and stop sequence or byte acknowledge like that used with I^{2}C. Data can be transferred as 8-, 16-, or 32-bit words. There is no limit to how much data can be transferred in a single transaction.

A data bit is shifted into the receiving device at the same time as the data bit is shifted out. Hence, once a word of data has been sent, the device has also received the next word. The SPI uses SCK clock edges to implement each bit transfer. At one SCK edge, each data sends a bit of data on the send line. The opposite clock edge a half of SCK clock cycle later, a data bit on the receive line is clocked into the receiving device. The specific clock edges are specified by the SPI mode of operation. The SCK signal can be asymmetrical as long as the period of the high or low state is greater than the inverse of two times the maximum data rate.

6 Problem Statement

You are to develop a software system that allows the PIC32MX370 to write an arbitrary number of 8-bit bytes to an arbitrary address location in the SPI flash memory device. You must be able to read back this stored data and determine if the data read matches the data written.

7 Background Information

7.1 PIC32MX370 SPI I/O

Figure 7.1 shows the signal connections to the S25FL132K flash memory IC. Table 7.1 lists the PIC32MX370 processor pins that these signals are connected to. The PIC32 processor pin for the SPI_CE signal is configured as a digital output pin, as shown in Listing B.1. To implement the wiring configuration shown in Fig. 7.1, processor I/O pins PORT F:2 and PORTF:7 are mapped to SPI1T and SPI1R respectively using the first two statements in Listing B.2 in Appendix B. The PIC32MX370 PORTF:6 has a fixed assignment to SCK1.

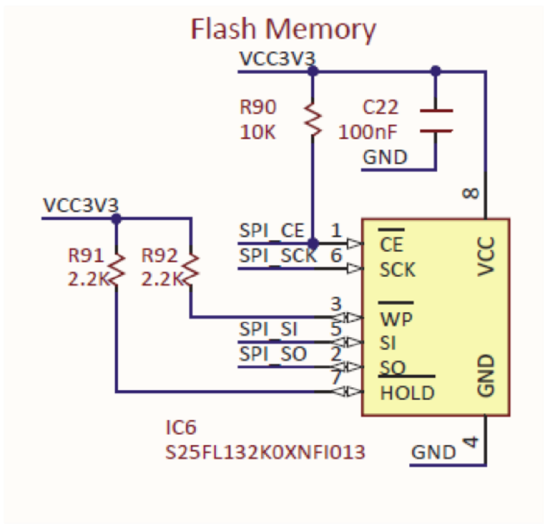


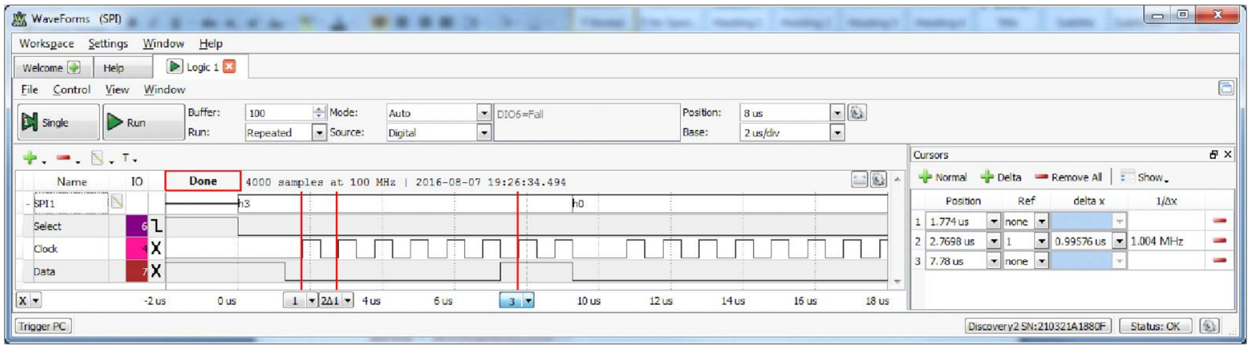
Figure 7.1. SPI flash memory IC schematic diagram.

Table 7.1. SPI Flash Memory to PIC32MX370 connection table.

Function	Flash Memory Pin	PIC32MX370 Pin
Chip Enable	SPI_CE – 1	PORT F Pin 8
Serial Clock	SPI_SCK – 6	PORT F Pin 6 – SCK1
Flash Serial Input	SPI_SI – 5	PORT F Pin 2 – SPI1R (MISO)
Flash Serial Output	SPI_SO – 2	PORT F Pin 7 – SPI1T (MOSI)

7.2 Interface with SPI Flash Memory

Using the SPI initialization shown in Listing B.2, the PIC32 processor is configured for SPI Mode 0 operation. Figure 7.2 shows that the bit output is held constant when the clock pulse makes a positive transition. Figure 7.2 also shows that the bit rate is 1 MHz.


Buffer: 100 Mode: Auto DIO6=Fall Position: 8 us
Run: Repeated Source: Digital Base: 2 us/div

Done 4000 samples at 100 MHz | 2016-08-07 19:26:34.494

	Position	Ref	delta x	1/Δx
1	1.774 us	none		
2	2.7698 us	1	0.99576 us	1.004 MHz
3	7.78 us	none		

Figure 7.2. SPI Bit timing for Mode 0 configuration.

As Fig. A.2 shows, the same SPIxSR serial shift buffer is used for input and output, thus requiring the common SCKx clock signal. Hence, when a byte of data is shifted out of the SDOx pin, data is also being shifted into the SPIxSR from the SDIx pin. In other words, to receive SPI serial data, you must send SPI serial data.

Listing B.3 shows the function that can be used to send, receive, or exchange a byte of data using SPI communications. In most cases, when the SPI master is sending header data to the slave, the master SPI ignores the data received on the SDIx pin. Similarly, as the master continues to send non-consequential data during a SPI slave read operation, the slave SPI discards the data sent to it. The SPI master clock must be generated for both sending and receiving.

7.2.1 SPI Flash Memory Software

We will use the S25FL132K SPI FLASH 132 MB memory device that is populated on the Basys MX3 processor board to demonstrate SPI device communications. The characteristics of this FLASH device are not representative of all control protocol used by SPI silicon devices. The FLASH memory device has command and status/configuration registers that can be accessed independently of the memory data.

Each device transaction is started with a command from the PIC32 processor. The command set of the S25FL132K Flash Memory is fully controlled through the SPI bus. Commands are initiated with the falling edge of Chip Select (CS#). The first byte of data clocked into the SI input provides the instruction code. Data on the SI input is sampled on the rising edge of clock with most significant bit (MSB) first.

Commands vary in length from a single byte to several bytes. Each command begins with an instruction code and may be followed by address bytes, a mode byte, read latency (dummy/don't care) cycles, or data bytes.

Commands are completed with the rising edge of edge CS#. Clock relative sequence diagrams for each command are included in the command descriptions. All read commands can be completed after any data bit. However, all commands that Write, Program, or Erase must complete on a byte boundary (CS# driven high after a full 8 bits have been clocked) otherwise the command will be ignored. This feature further protects the device from inadvertent writes. Additionally, while the memory is being programmed or erased, all commands except for Read Status Register and Suspend commands will be ignored until the program or erase cycle has completed. When the Status Register is being written, all commands except for Read Status Register will be ignored until the Status Register write operation has completed.

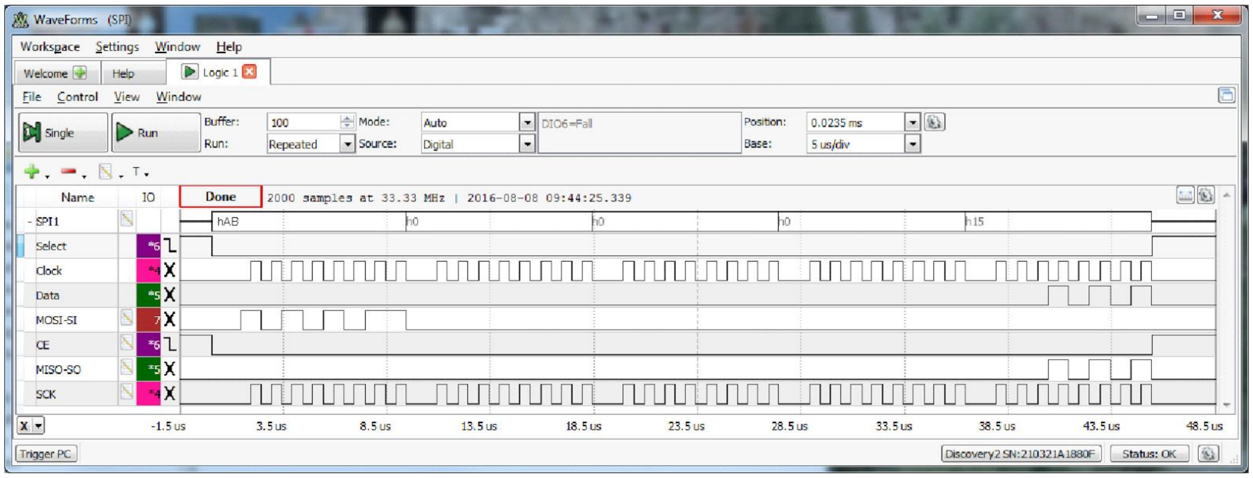
Since the data sheet for this Flash memory device is quite daunting, excerpts from that manufacturers data sheets are shown in Table 7.2 through 7.5 with the commands highlighted that can be used to implement basic FLASH memory management.

The highlighted commands in Table 7.2 allow for reading basic device identification parameters. Each command is sent to the device followed by zero to five read bytes. It is recommended that the "Release Power down/Device ID" command be sent as part of a Flash initialization process. Figure 7.3 shows the screen capture for the SPI transaction of this command. For the S25FL132K Flash part, the device ID is 21 or 0x15 as demonstrated in Fig. 7.3.

Table 7.2. Command Set (ID and Security Commands)[1]

Command Name	BYTE 1 (Instruction)	BYTE 2	BYTE 3	BYTE 4	BYTE 5	BYTE 6
Deep Power-down	B9h					
Release Power Down / Device ID	ABh	Dummy	Dummy	Dummy	Device ID (1)	
Manufacturer / Device ID (2)	90h	Dummy	Dummy	00h	Manufacturer	Device ID
JEDEC ID	9Fh	Manufacturer	Memory Type	Capacity		
Read SFDP Register / Read Unique ID Number	5Ah	00h	00h	A7-A0	Dummy	(D7-D0, …)
Read Security Registers (3)	48h	A23-A16	A15-A8	A7-A0	Dummy	(D7-D0, …)
Erase Security Registers (3)	44h	A23-A16	A15-A8	A7-A0		
Program Security Registers (3)	42h	A23-A16	A15-A8	A7-A0	D7-D0, …	

[1] S25FL132K and S25FL164K Data Sheet, http://www.mouser.com/ds/2/380/S25FL132K_164K_00-268210.pdf



Buffer: 100 Mode: Auto DIO6=Fall Position: 0.0235 ms
Run: Repeated Source: Digital Base: 5 us/div

Done 2000 samples at 33.33 MHz | 2016-08-08 09:44:25.339

Figure 7.3. Release Power Down/Read Device ID SPI transaction.

The flash memory status register, SR0, has two volatile bits, bits 0 and 1, that indicate the current operational status of the memory chip. When WEL (bit 1) is set high, the device can be written to erase or program. The Page Program command (0x02) allows from one byte to 256 bytes (a page) of data to be programmed at previously erased (0xFF) memory locations. A Write Enable command must be executed before the device will accept the Page Program Command (Status Register bit WEL= 1). The command is initiated by driving the CS# pin low then shifting the instruction code "02h," followed by a 24-bit address (A23-A0) and at least one data byte, into the SI pin. The CS# pin must be held low for the entire length of the command while data is being sent to the device. Refer to section 8 of the Flash Memory data sheet (Reference 5) for additional information.

Table 7.3. Command set (Configuration, Status, Erase, Program Commands)[2]

Command Name	**BYTE 1 (Instruction)**	**BYTE 2**	**BYTE 3**	**BYTE 4**	**BYTE 5**	**BYTE 6**
Read Status Register - 1	05h	SR1[7:0] (2)(4)				
Read Status Register - 2	35h	SR2[7:0] (2)(4)				
Read Status Register - 3	33h	SR3[7:0] (2)				
Write Enable	06h					
Write Disable	04h					
Write Status Registers	01h	SR1[7:0]	SR2[7:0]	SR3[7:0]		
Set Burst with Wrap	77h	Xxh	Xxh	Xxh	SR3[7:0] (3)	
Set Block / Pointer Protection (**S25FL132K / S25FL164K**)	39h	A23-A16	A15-A10, x, x	xxh		
Page Program	02h	A23-A16	A15-A8	A7-A0	D7-D0	

[2] S25FL116K, S25FL132K, S25FL164K, http://www.cypress.com/file/196886/download

Sector Erase (4 kB)	20h	A23-A16	A15-A8	A7-A0		
Block Erase (64 kB)	D8h	A23-A16	A15-A8	A7-A0		
Chip Erase	C7h / 60h					
Erase / Program Suspend	75h					
Erase / Program Resume	7Ah					

Table 7.4. Status Register 0 bit definitions.

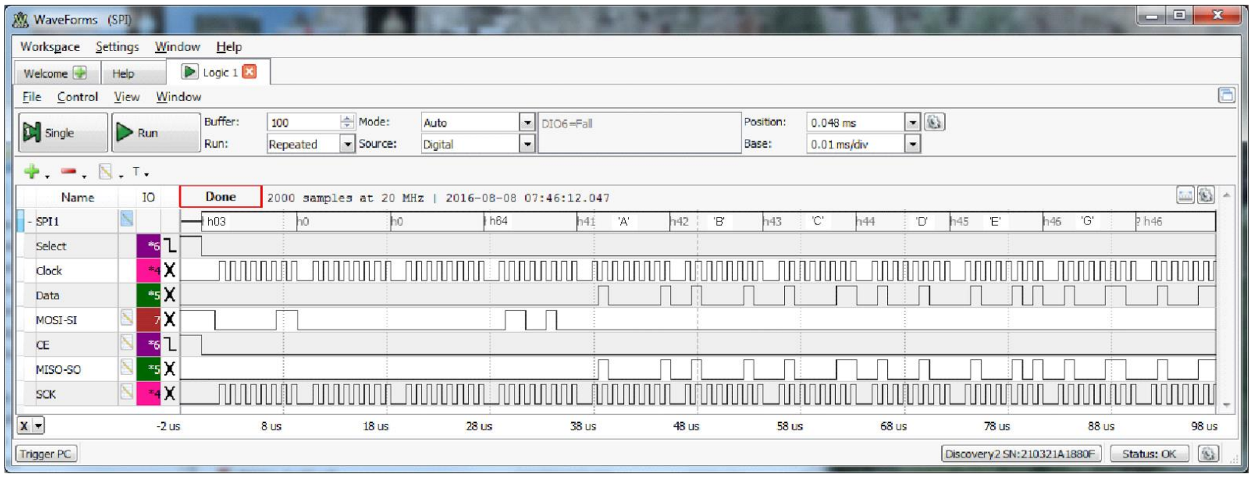
Bits	Field Name	Function	Type	Default State	Description
7	SRP0	Status Register Protect 0	Non-volatile and Volatile versions	0	0 = WP# input has no effect of Power Supply Lock Down mode 1 = WP# input can protect the Status Register or OTP Lock Down.
6	SEC	Sector / Block Protect		0	0 = BP2-BP0 protect 64 kB blocks 1 = BP2-BP0 protect 4 kB sectors
5	TB	Top / Bottom Protect		0	0 = BP2-BP0 protect from the Top down 1 = BP2-BP0 protect from the Bottom up
4	BP2	Block Protect Bits		0	000b = No protection
3	BP1			0	
2	BP0			0	
1	WEL	Write Enable Latch	Volatile, Read only	0	0 = Not Write Enabled, no embedded operation can start 1= Write Enabled, embedded operation can start
0	BUST	Embedded Operation Status	Volatile, Read only	0	0 = Not Bust, no embedded operation in progress 1 = Busy, embedded operation in progress

Any number of bytes can be read from the flash device starting at any address. As Table 7.5 illustrates, a read command (0x03) initializes the starting address. The read operation is terminated whenever the CS# pin is asserted high.

Table 7.5. Command Set (Read Commands).

Command Name	BYTE 1 (Instruction)	BYTE 2	BYTE 3	BYTE 4	BYTE 5	BYTE 6
Read Data	03h	A23-A16	A15-A8	A7-A0	(D7-D0, …)	
Fast Read	0Bh	A23-A16	A15-A8	A7-A0	Dummy	(D7-D0, …)
Fast Read Dual Output	3Bh	A23-A16	A15-A8	A7-A0	Dummy	(D7-D0, …) [1]
Fast Read Quad Output	6Bh	A23-A16	A15-A8	A7-A0	Dummy	(D7-D0, …) [3]

Fast Read Dual I/O	BBh	A23-A8 (2)	A7-A0, M7-M0 (2)	(D7-D0, …) (1)		
Fast Read Quad I/O	EBh	A23-A0, M7-M0 (4)	(x, x, x, x, D7-D0, …) (5)	(D7-D0, …) (3)		
Continuous Read Mode Reset (6)	FFh	FFh				


WaveForms (SPI)
Workspace Settings Window Help
Welcome Help Logic 1
File Control View Window
Single Run
Name IO
Done
SPI1 h03 h0 h0 h64 h41 'A' h42 'B' h43 'C' h44 'D' h45 'E' h46 'G' h46
Select
Clock
Data
MOSI-SI
CE
MISO-SO
SCK
-2 us 8 us 18 us 28 us 38 us 48 us 58 us 68 us 78 us 88 us 98 us
Trigger PC

Buffer: 100 Mode: Auto DIO6=Fall Position: 0.048 ms
Run: Repeated Source: Digital Base: 0.01 ms/div

2000 samples at 20 MHz | 2016-08-08 07:46:12.047

Discovery2 SN:210321A1880F Status: OK

Figure 7.4. SPI Flash Read Byte command and three-byte address followed by reading data 'A', 'B', 'C', etc.

8 Lab 4c

8.1 Requirements

1. The PIC32MX370 UART channel 4 must operate at 38400 BAUD with no parity.
2. Two buffers must be created of size 1024 bytes.
3. Generate a 550-byte data set using the following code:

   #define nBytes 550

   ```
   for(i=0; i<nBytes; i++)              // Initialize array for FLASH write
   {
     wrBuffer[i] = (BYTE) ('A' + i);
   }
   ```

4. Whenever the push button, BTNR, is pressed, 550 bytes of data must be written to the flash starting at address 100 (0x64). The following sequence must be executed:
   a. 4K bytes of flash memory must be erased starting at address 0.
   b. You must verify that the memory space that will be programmed has been erased (data value set to 0xFF).
   c. The preset data stream must be programmed into the flash memory at the starting address.
   d. A message is sent to the workstation terminal stating that the programming has completed.
5. All bytes in a receive buffer must be initialized to 0.
6. 550 bytes of flash memory starting at address 100 must be read and each byte compared to the original data set.
7. If all the bytes compare, the message “No memory error!” is sent to the terminal.

8. If any byte does not compare, the message "Error at address ###: ## written - ## read" where ''###" refers to the specific values.
9. Steps 4 through 8 must be repeated once each second.

8.2 Design Phase

1. Develop a data flow diagram for the software components needed for the requirements of Lab 4c.
2. Schematic diagrams: Provide a block diagram of the equipment used for Lab 4c.
3. Flow diagrams: Provide a complete software control flow diagram for lab 4c.

8.3 Construction Phase

1. Create a new project named Lab4c.
2. Add C program files for configuring the processor and initializing the I/O for the switches and LEDs on Basys MX3 board.
3. And C program files developed for previous labs that provide an interface to the UART.
4. Develop a file containing the all SPI functions listed Appendix B.
5. Using the primitive control function listed Appendix B, complete the requirements for Lab4c using the call graphs provided in Fig. 8.1 and Fig. 8.2.

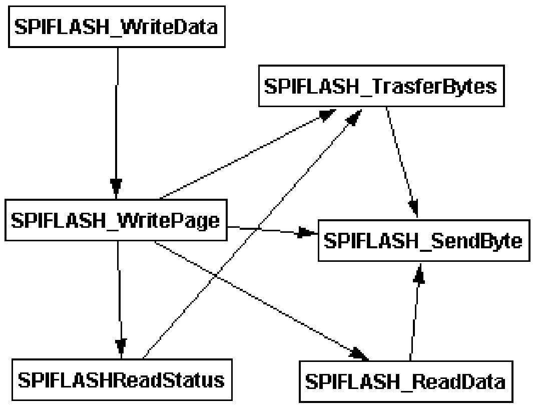


Figure 8.1. Call map for SPI FLash Write Page.

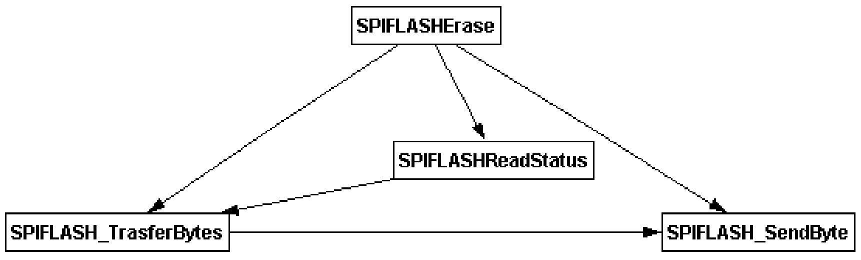


Figure 8.2. Call map for SPI Flash Erase.

6. Download the completed functional project to the PIC32MX370 processor and test by running the project.

8.4 Testing

1. Executing the program must result in the message "No memory error!" being sent to the terminal once each second.

8.5 SPI Design Challenge

Replace the code in Listing B.7 for the function "SPIFLASH_WritePage" that uses software-intensive "for" loops with one that uses DMA.

9 Questions

1. Why does SPI normally have higher data transfer rates?
2. Is it necessary for the SPI SCK signal to have a 50% duty cycle?
3. Can text data be sent using SPI communications?
4. Can SPI master be configured for only receiving data without transmitting data?
5. How many slave devices can be connected to the SPI bus?
6. What will happen to the SPI communications is an interrupt occurs in the middle of a transmission?

10 References

1. Embedded Computing and Mechatronics with the PIC32 Microcontroller, 1st Edition, by Kevin Lynch (Author), Nicholas Marchuk (Author), Matthew Elwin (Author), https://www.amazon.com/Embedded-Computing-Mechatronics-PIC32-Microcontroller/dp/0124201652
2. "PIC32MX330/350/370/430/4450/470 32 Bit Microcontroller Datasheet (60001185E)", http://ww1.microchip.com/downloads/en/DeviceDoc/60001185E.pdf
3. "Overview and Use of the PICmicro Serial Peripheral Interface", http://ww1.microchip.com/downloads/en/devicedoc/spi.pdf
4. Introduction to I²C and SPI protocols, http://www.byteparadigm.com/applications/introduction-to-i2c-and-spi-protocols/
5. 16 Mbit (2 Mbyte), 32 Mbit (4 Mbyte), 64 Mbit (8 Mbyte) 3.0 V NVM, http://www.cypress.com/file/196886/download
6. S25FL132K and S25FL164K Data Sheet, http://www.mouser.com/ds/2/380/S25FL132K_164K_00-268210.pdf
7. Implementing File I/O Functions Using Microchip's Memory Disk Drive File System Library, http://ww1.microchip.com/downloads/en/AppNotes/01045b.pdf

Appendix A: Lab 4 Parts Configuration

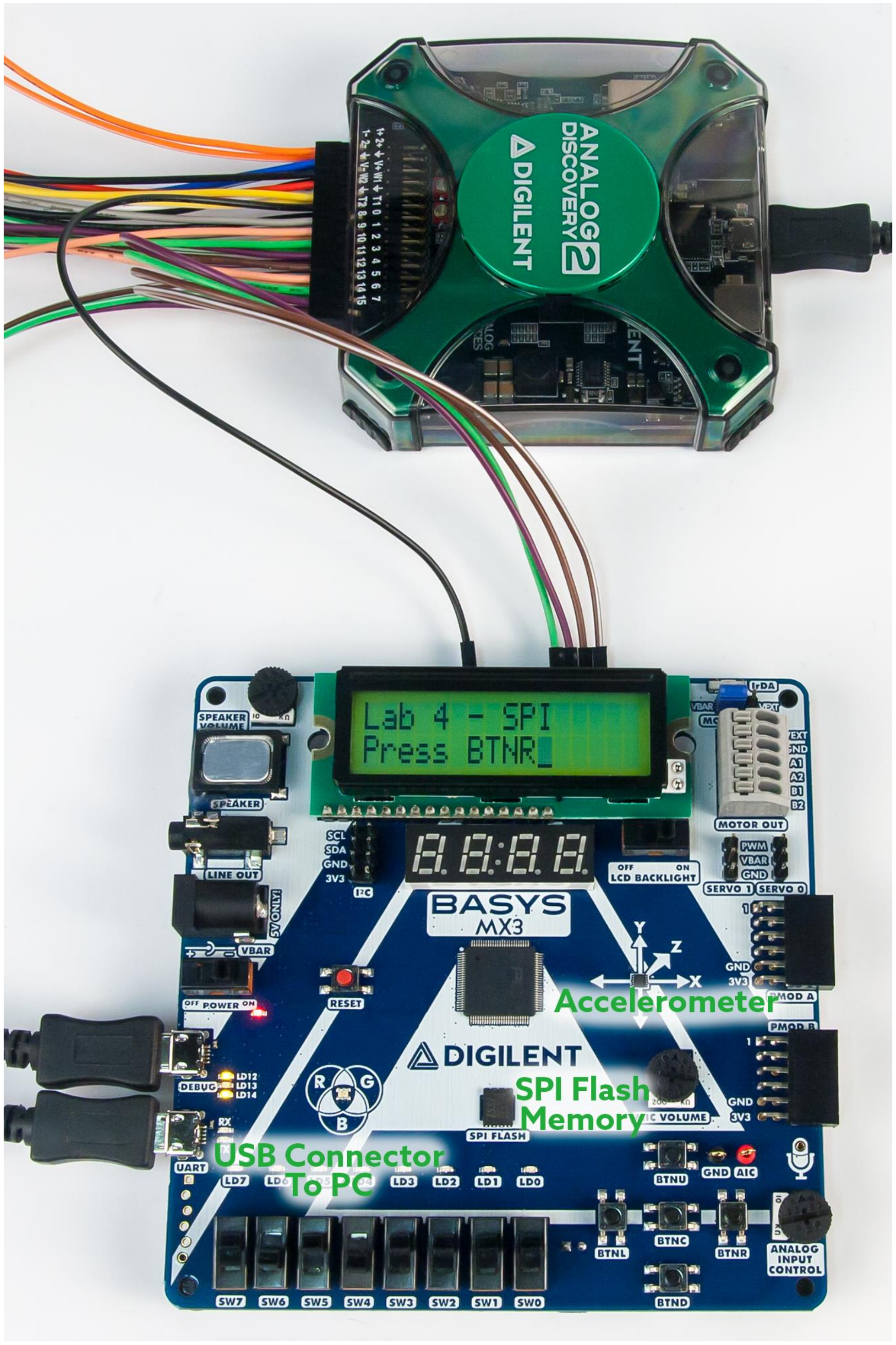


Figure A.1. Unit 4 hardware and instrumentation configuration.

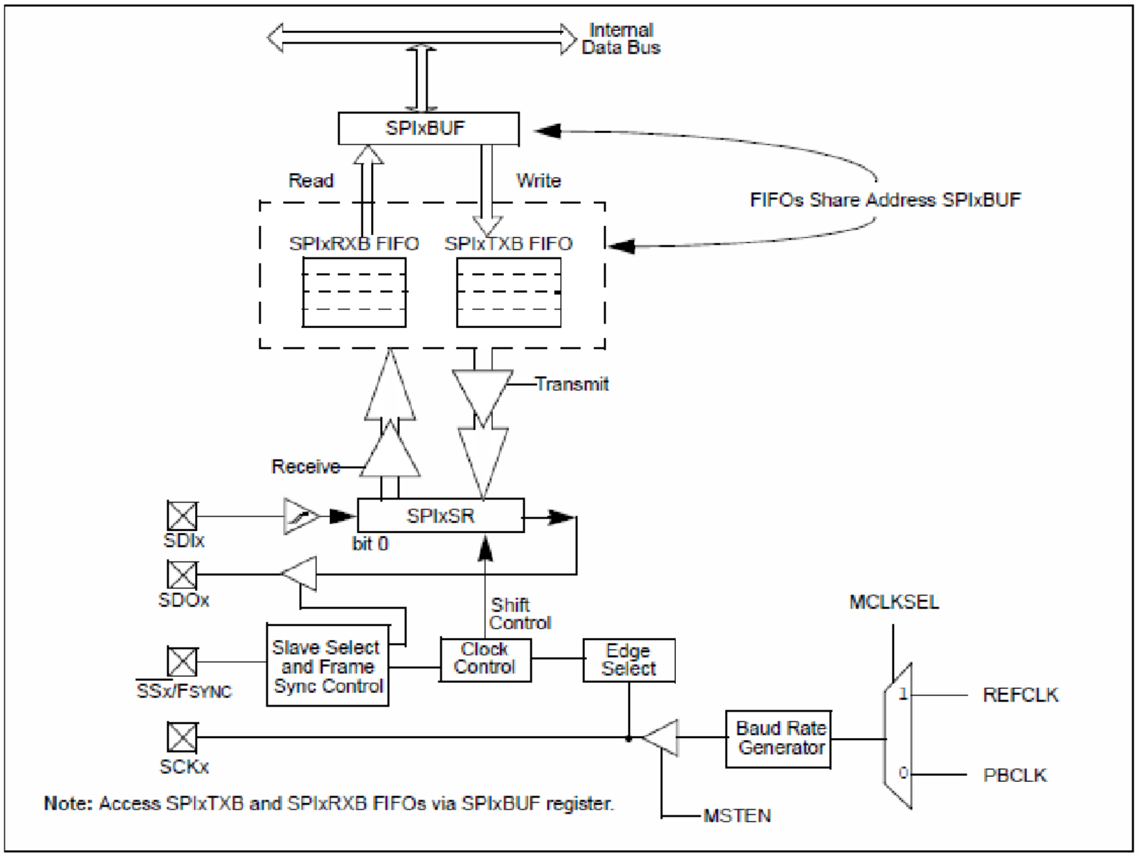


Figure A.2. SPI block diagram from the Microchip PIC32MX370 data sheet.

Appendix B: Allocating a Heap in MPLAB X

Listing B.1. Flash Memory SPI Initialization

```
#define SPIFLASH_SEL              LATFbits.LATF8
#define SPIFLASH_CS_TRIS          TRISFbits.TRISF8
unsigned int flash_mem_init(void)
{
unsigned int bitrate;

    bitrate = init_SPI1();
    SPIFLASH_CS_TRIS = 0;       // Set FLASH select as output
    SPIFLASH_SEL = 1;           // Reset FLASH chip select
    return bitrate;             // Has no meaning at this time
}
```

Listing B.2. SPI Channel 1 initialization for Master Mode

```
#define MAX_SPI_CLK_FREQ          1000000
#define GetPeripheralClock()     (GetSystemClock()/8)
unsigned int init_SPI1(void)
{
unsigned int pbFreq;
unsigned int SPI_Clk_Freq;
unsigned int berg_val;

// Map PPS pins to SDI1
   SDI1R = 0x0F;       // Map SDI1 to RF7 - Input (MISO))
   RPF2R = 0x08;       // MAP SDO1 to RF2 - Output (MOSI))
//  SCK1 has fixed assignment to RF6

// Initialize the direction of the SPI interface signals.  The device SS
// is not assigned in this initialization since the pin assignment can be
// hardware dependent.
   SDI1_TRIS = 1;      // FLASH SO
   SDO1_TRIS = 0;      // FLASH SI
   SCK1_TRIS = 0;      // FLASH SCK
   SPI1_SDO = 0;
   SPI1_SCK = 0;

// Peripheral Bus Frequency = System Clock / PB Divider
// PB Frequency can be maximum 40 MHz *
   pbFreq = GetPeripheralClock();
// Compute proper BERG value for specified SPI bit rate
   berg_val = SpiBrgVal(pbFreq, MAX_SPI_CLK_FREQ);
// Compute actual SPI bit rate
   SPI_Clk_Freq = pbFreq / ( 2 * (berg_val+1) );

// Enable SPI1, Set to Master Mode & Set CKE bit 1 for SPI MODE 0
// Serial output data changes on transition from Active clock state to Idle
// clock state. Idle clock state is low.
   SPI1BRG = berg_val;         // Set SPI bit rate
   SPI1CONbits.MSTEN = 1;      // SPI Master enable
   SPI1CONbits.CKE = 1;        // Set for SPI Mode 0
   SPI1CONbits.ON = 1;         // Enable SPI1
   return SPI_Clk_Freq;
}
```

Listing B.3. SPI Channel 1 Byte Transaction.

```
BYTE spiXfer(BYTE data_out)
{
   SpiChnPutC(1, data_out);
   DelayUs(1);                    // Not required – testing only
   return SpiChnGetC(1);
}
```

Listing B.4. FLASH Transfer Bytes

```
void SPIFLASH_TrasferBytes(int nBytes, unsigned char *pbRdData,
                          unsigned char *pbWrData)
{
   int i;
   SPIFLASH_SEL = 0;                      // Activate CS
   for(i = 0; i< nBytes; i++)
   {
       SPIFLASH_SendByte(pbWrData[i]); // Write byte to SPI FLash
       pbRdData[i] = SPI1BUF;
   }
   SPIFLASH_SEL = 1; // Deactivate CS
}
```

Listing B.5. FLASH Release Power Down and read Device ID

```
void SPIFLASH_ReleasePowerDownGetDeviceID(BYTE *rd)
{
BYTE wr[5] = {0};

   spi_wr[0] = SPIFLASH_CMD_PWRDWN_DEVID;
   spi_wr[1] = 0;
   spi_wr[2] = 0;
   spi_wr[3] = 0;
   spi_wr[4] = 0;
   SPIFLASH_TrasferBytes(5, rd, wr);
}
```

Listing B.6. SPI Write data to Flash

```
int SPIFLASH_WriteData(int nBytes, BYTE *pbWrData, unsigned int flashAddr)
{
int error = 0;
unsigned int pageAddrS;      // Page start programming address
unsigned int pageAddrE;      // Page end programming address
unsigned int pageAddrL;      // Last Page end programming address
unsigned int pageBytes;      // Bytes to program in current page
unsigned int nPages;
BYTE *dataPtr;               // Updated data array pointer

   dataPtr = pbWrData;
   pageAddrS = flashAddr;
   pageAddrL = pageAddrS + nBytes;

// Write data one page at a time until error or write is complete.
   while((pageAddrS < pageAddrL) && !error)
   {
       pageAddrE = (pageAddrS & 0xFFFFFF00) + SPIFLASH_PAGE_SIZE;
       if(pageAddrE >= pageAddrL)
       {
           pageAddrE = pageAddrL;
       }
       pageBytes = pageAddrE - pageAddrS;
// Program one full page (256 bytes) or partial page within page boundaries
       error |= SPIFLASH_WritePage(pageBytes, dataPtr, pageAddrS);
       pageAddrS = pageAddrE;
       dataPtr += pageBytes;
   }

   return error;
}
```

Listing B.7. SPI Write Page data to Flash

```
static int SPIFLASH_WritePage(int nBytes, unsigned char *pbWrData,
                             unsigned int data_addr)
{
BYTE wr_hdr[5] = {0};      // Used for command and address
BYTE rd_hdr[5] = {0};
BYTE tmpBuffer1[256];      // Read buffer for checking page erased
unsigned int pageAddr;     // Start write address
int error = 0;             // Process error
BYTE chk_flag = 0xFF;      // Check erased flag
int i;                     // General index
BYTE FLASHStatus;          // Flash memory busy flag

// Test page constraint
   if(((data_addr & 0x000000FF) + nBytes) > 256)
   {
       error = 1;  // Write beyond page boundary
   }
   else
   {

// Read existing page contents
       SPIFLASH_ReadData(nBytes, tmpBuffer1, data_addr);

// Check for memory erased
       chk_flag = 0xFF;
       for(i=0; i<nBytes; i++)
       {
           chk_flag &= tmpBuffer1[i];
       }
       if(chk_flag != 0xFF)
       {
           error = 1;
           return error;
       }

       wr_hdr[0] = SPIFLASH_CMD_WREN;  // Unlock FLASH for writing
       SPIFLASH_TrasferBytes(1, rd_hdr, wr_hdr);

// Poll Status Register 1 until WEL bit is set and BUSY flag is reset.
       do {
           FLASHStatus = SPIFLASHReadStatus();
       } while(FLASHStatus != SPIFLASH_WEL_STATUS_BIT);

// Setup write header
       wr_hdr[0] = SPIFLASH_CMD_WRITE;
       wr_hdr[1] = (BYTE) (data_addr >> 16);
       wr_hdr[2] = (BYTE) (data_addr >> 8);
       wr_hdr[3] = (BYTE) (data_addr);
       wr_hdr[4] = 0;

       SPIFLASH_SEL = 0;                   // Activate CS
       for(i=0; i<4; i++)                  // SPI Write command and address
       {
           SPIFLASH_SendByte(wr_hdr[i]);
```

```
            rd_hdr[0] = SPI1BUF;            // Ignore returned bytes
        }
        for(i=0; i<nBytes; i++)             // Write data
        {
            SPIFLASH_SendByte(*pbWrData++);
            rd_hdr[0] = SPI1BUF;            // Ignore returned bytes
        }
        SPIFLASH_SEL = 1;                   // deactivate CS

        wr_hdr[0] = SPIFLASH_CMD_WRDI;      // Disable SPI FLash write

        SPIFLASH_TrasferBytes(1, rd_hdr, wr_hdr);
        do {
            FLASHStatus = SPIFLASHReadStatus();
        } while(FLASHStatus != 0);  // Wait until write is complete

    }
    return error;
}
```