CHAPTER 12

SPI Communication

The Serial Peripheral Interface (SPI) allows the PIC32 to communicate with other devices at high speeds. Numerous devices use SPI as their primary mode of communication, such as RAM, flash, SD cards, ADCs, DACs, and accelerometers. Unlike the UART, SPI communication is synchronous: the "master" device on an SPI bus creates a separate clock signal that dictates the timing of communication. The devices do not have to be configured in advance to share the same bit rate, and any clock frequency can be used, provided it is within the capabilities of the chips. High speeds are possible; for example, the SPI interface of the STMicroelectronics LSM303D accelerometer/magnetometer, a device considered in this chapter, supports up to 10 MHz clock signals.

12.1 Overview

SPI is a master-slave architecture, and an SPI bus has one master device and one or more slaves. A minimal SPI bus consists of three wires (in addition to GND): Master Output Slave Input (MOSI), carrying data from the master to the slave(s); Master Input Slave Output (MISO), carrying data from the slave(s) to the master; and the master's clock output, which clocks the data transfers, one bit per clock pulse. Each SPI device on the bus correspondingly has three pins: Serial Data Out (SDO), Serial Data In (SDI), and System Clock (SCK). The MOSI line is connected to the master's SDO pin and the slaves' SDI pins, and the MISO line is connected to the master's SDI pin and the slaves' SDO pins. All slaves' SCK pins are inputs, connected to the master's SCK output (Figure 12.1).

Each of the PIC32's three SPI peripherals can either be a master or a slave. We typically think of the PIC32 acting as the master.

If there is more than one slave on the bus, then the master controls which slave is active by using an active-low slave select ($\overline{\text{SS}}$) line, one per slave. Only one slave-select line can have a low signal at a time, and the line which is low indicates which slave is active. Unselected slaves must let their SDO output float at high impedance, effectively disconnected from the MISO line, and they should ignore data on the MOSI line. In Figure 12.1, there are two slaves, and therefore two output slave-select lines from the master $\overline{\text{SS1}}$ and $\overline{\text{SS2}}$, and one slave-select input line for each slave. Thus, adding more slaves to the bus means adding more wires.

Embedded Computing and Mechatronics with the PIC32 Microcontroller. http://dx.doi.org/10.1016/B978-0-12-420165-1.00012-3

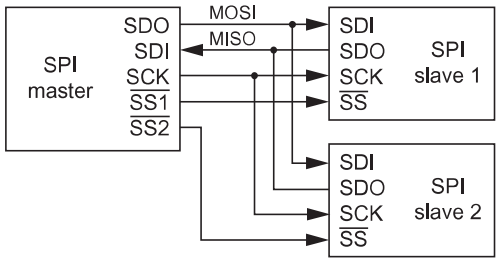

Figure 12.1
An SPI master connected to two slave devices. Arrows indicate data direction. A PIC32 SPI peripheral can either be a master or a slave. Only one slave select line can be active (low) at a time.

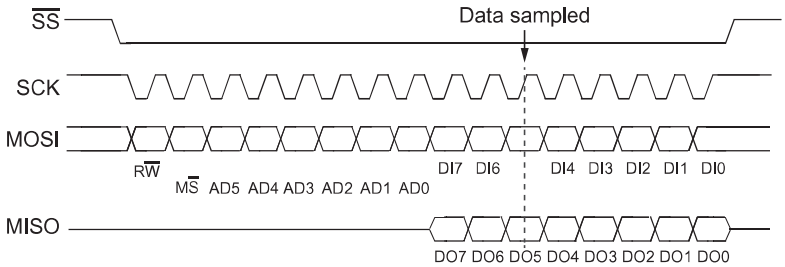

Figure 12.2
Timing for an SPI transaction with a slave LSM303D accelerometer/magnetometer. The LSM303D is selected when $\overline{SS}$ is driven low. On the falling edge of the master's SCK, both the master (controlling MOSI with its SDO pin) and the slave (controlling MISO with its SDO pin) transition to the next bit of their signal. On the rising edge of SCK, the devices read the data, the master reading MISO with its SDI pin and the slave reading MOSI with its SDI pin. In this example, the master first sends 8 bits, the first of which ($\overline{RW}$) determines whether it will be reading data from the LSM303D or writing data to it. The bits AD0 to AD5 determine the address of the LSM303D register that the master is accessing. Finally, if the operation is a read from the LSM303D, the LSM303D puts 8 bits DO0 to DO7 on the MISO line; otherwise the master sends 8 bits on the MOSI line to write to the LSM303D.

Compare this to the (typically lower speed) I^2C (Chapter 13) and CAN (Chapter 19) buses, which have a fixed number of wires regardless of the number of devices on them.

The master initiates all communication by creating a square wave on the clock line. Reading from and writing to the slave occur simultaneously, one bit per clock pulse. Transfers occur in groups of 8, 16, or 32 bits, depending on the slave. Figure 12.2, taken from the LSM303D data sheet, depicts the signals for a typical SPI transaction. The timing of the data bits relative to the clock signal are settable using SFRs to match the behavior of other devices on the SPI bus; see Section 12.2 for more information.

In addition to the basic SPI communication described above, the PIC32 SPI module has other modes of operations, which we do not cover in detail here. For example, a framing mode allows for streaming data from supported devices. By default, the SPI peripheral has only a single input and single output buffer. The PIC32 also has an enhanced buffer mode, which provides FIFOs for queuing data waiting to be transferred or processed.

12.2 Details

Each of the three SPI peripherals, SPI2 to SPI4, has four pins associated with it: SCKx, SDIx, SDOx, and $\overline{\text{SSx}}$, where x is 2 to 4. When the SPI peripheral is a slave, it can be configured so that it only receives and transmits data when the input $\overline{\text{SSx}}$ is low (e.g., when there is more than one slave on the bus). When the SPI peripheral is a master, it can be configured to drive the $\overline{\text{SSx}}$ automatically when communicating with a single slave. If there are multiple slaves on the bus, however, other digital outputs can be used to control the multiple slave select pins of the slaves.

Each SPI peripheral uses four SFRs, SPIxCON, SPIxSTAT, SPIxBUF, and SPIxBRG, x = 2 to 4. Many of the settings are related to the alternative operation modes that we do not discuss. SFRs and fields that we omit may be safely left at their default values for typical applications. All default bits are zero, except for one read-only bit in SPIxSTAT.

SPIxCON This register contains the main control options for the SPI peripheral.

SPIxCON⟨28⟩ or SPIxCONbits.MSSEN: Master slave select enable. If set, the SPI master will assert (drive low) the slave select pin $\overline{\text{SSx}}$ prior to sending an 8-, 16-, or 32-bit transmission and de-assert (drive high) after sending the data. Some devices require you to toggle the slave select after a complete multi-word transaction; in this case, you should clear SPIxCONbits.MSSEN to zero (the default) and use any digital output as the slave select.

SPIxCON⟨15⟩ or SPIxCONbits.ON: Set to enable the SPI peripheral.

SPIxCON⟨11:10⟩ or SPIxCONbits.MODE32 (bit 11) and SPIxCONbits.MODE16 (bit 10): Determines the communication width.

- 0b1X (SPIxCONbits.MODE32 = 1): 32 bits of data sent per transfer.
- 0b01 (SPIxCONbits.MODE32 = 0 and SPIxCONbits.MODE16 = 1): 16 bits of data sent per transfer.
- 0b00 (SPIxCONbits.MODE32 = SPIxCONbits.MODE16 = 0): 8 bits of data sent per transfer.

SPIxCON⟨9⟩ or SPIxCONbits.SMP: determines when, relative to the clock pulses, the master samples input data. Should be set to match the slave device's specifications.

- 1 Sample at end of a clock pulse.
- 0 Sample in the middle of a clock pulse.

SPIxCON⟨8⟩ or SPIxCONbits.CKE: The clock signal can be configured as either active high or active low by setting SPIxCONbits.CKP (see below). This bit, SPIxCONbits.CKE, determines whether the master changes the current output bit on the edge when the clock transitions from active to idle or idle to active (see SPIxCONbits.CKP, below). You should choose this bit based on what the slave device expects.

1 The output data changes when the clock transitions from active to idle.
0 The output data changes when the clock transitions from idle to active.

SPIxCON⟨7⟩ or SPIxCONbits.SSEN: This slave select enable bit determines whether the $\overline{\text{SSx}}$ pin is used in slave mode.

1 The $\overline{\text{SSx}}$ pin must be low for this slave to be selected on the SPI bus.
0 The $\overline{\text{SSx}}$ pin is not used, and is available to be used with another peripheral.

SPIxCON⟨6⟩ or SPIxCONbits.CKP: The clock signal can be configured as being active high or active low. Chosen in conjunction with SPIxCONbits.CKE, above, this setting should match the expectations of the slave device.

1 The clock is idle when high, active when low.
0 The clock is idle when low, active when high.

SPIxCON⟨5⟩ or SPIxCONbits.MSTEN: Master enable. Usually the PIC32 operates as the master, meaning that it controls the clock and hence when and how fast data is transferred, in which case this bit should be set to 1. To use the SPI peripheral as a slave, this bit should be cleared to 0.

1 The SPI peripheral is the master.
0 The SPI peripheral is the slave.

SPIxSTAT The status of the SPI peripheral.

SPIxSTAT⟨11⟩ or SPIxSTATbits.SPIBUSY: When set, indicates that the SPI peripheral is busy transferring data. You should not access the SPI buffer SPIxBUF when the peripheral is busy.

SPIxSTAT⟨6⟩ or SPIxSTATbits.SPIROV: Set to indicate that an overflow has occurred, which happens when the receive buffer is full and another data word is received. This bit should be cleared in software. The SPI peripheral can only receive data when this bit is clear.

SPIxSTAT⟨1⟩ or SPIxSTATbits.SPITXBF: SPI transmit buffer full. Set by hardware when you write to SPIxBUF. Cleared by hardware after the data you wrote is transferred into the transmit buffer SPIxTXB, a non-memory-mapped buffer. When this bit is clear you can write to SPIxBUF.

SPIxSTAT⟨0⟩ or SPIxSTATbits.SPIRXBF: SPI receive buffer full. Set by hardware when data is received into the SPI receive buffer indicating that SPIxBUF can be read. Cleared when you read the data via SPIxBUF.

SPIxBUF Used to both read and write data over SPI. When, as the master, you write to SPIxBUF, the data is actually stored in a transmit buffer SPIxTXB, and the SPI peripheral generates a clock signal and sends the data over the SDOx pin. Meanwhile, in response to

the clock signal, the slave sends data to the SDIx pin, where it is stored in a receive buffer SPIxRXB, which you do not have direct access to. To access this received data, you do a read from SPIxBUF. Therefore, perhaps unintuitively, after executing the C code

```
SPI1BUF = data1;
data2 = SPI1BUF;
```

`data1` and `data2` will not be identical! `data1` is sent data, and `data2` is received data. To avoid buffer overflow errors, every time you write to SPIxBUF you should also read from SPIxBUF, even if you do not need the data. Additionally, since the slave can only send data when it receives a clock signal, which is only generated by the master when you write data, as a master you must write to SPIxBUF before getting new data from the slave.

SPIxBRG Determines the SPI clock frequency. Only the lowest 12 bits are used. To calculate the appropriate value for SPIxBRG use the tables provided in the Reference Manual or the following formula:

$$\text{SPIxBRG} = \frac{F_{PB}}{2F_{sck}} - 1, \tag{12.1}$$

where F_{PB} is the peripheral bus clock frequency (80 MHz for the NU32 board) and F_{sck} is the desired clock frequency. SPI can operate at relatively high frequencies, in the MHz range. The master dictates the clock frequency and the slave reads the clock; therefore a slave device never needs to configure a clock frequency.

The interrupt vector for SPIx is `_SPI_x_VECTOR`, where x is 2 to 4. An interrupt can be generated by an SPI fault, SPI RX conditions, and SPI TX conditions. For SPI2, the interrupt flag status bits are IFS1bits.SPI2EIF (error), IFS1bits.SPI2RXIF (RX), and IFS1bits.SPI2TXIF (TX); the enable control bits are IEC1bits.SPI2EIE (error), IEC1bits.SPI2RXIE (RX), and IEC1bits.SPI2TXIE (TX); and the priority and subpriority bits are IPC7bits.SPI2IP and IPC7bits.SPI2IS. For SPI3 and SPI4, the bits are named similarly, replacing SPI2 with SPIx (x = 3 or 4), and with SPI3's flag status bits in IFS0, enable control bits in IEC0, and priority in IPC6; and SPI4's in IFS1, IEC1, and IPC8.

The sample code in this chapter does not include interrupts, but TX and RX interrupt conditions can be selected using SPIxCONbits.STXISEL and SPIxCONbits.SRXISEL. See the Reference Manual.

12.3 Sample Code

12.3.1 Loopback

The first example uses two SPI peripherals to allow the PIC32 to communicate with itself over SPI. Although practically useless, the code serves as vehicle for understanding the basic

configuration of both an SPI master and an SPI slave. Both master (SPI4) and slave (SPI3) are configured to send 16 bits per transfer. Notice that the SPI master sets a clock frequency, whereas the slave does not. The program prompts the user to enter two 16-bit hexadecimal numbers. The first number is sent by the master and the second by the slave. The code then reads from the SPI ports and prints the results.

Remember, one bit of data is transferred in each direction per clock cycle. When the slave writes to its SPI buffer, the data is not actually sent until the master generates a clock signal. Both master and slave use the clock to send and receive the data. As sending and receiving happen simultaneously, the master should always read from SPI4BUF after writing to SPI4BUF.

Code Sample 12.1 spi_loop.c**. SPI Loopback Example.**

```
#include "NU32.h"  // constants, funcs for startup and UART
// Demonstrates spi by using two spi peripherals on the same PIC32,
// one is the master, the other is the slave
// SPI4 will be the master, SPI3 the slave.
// connect
// SDO4 -> SDI3 (pin F5 -> pin D2)
// SDI4 -> SDO3 (pin F4 -> pin D3)
// SCK4 -> SCK3 (pin B14 -> pin D1)

int main(void) {
  char buf[100] = {};
  // setup NU32 LED's and buttons
  NU32_Startup();

  // Master - SPI4, pins are: SDI4(F4), SDO4(F5), SCK4(B14), SS4(B8); not connected)
  // since the pic is just starting, we know that SPI is off. We rely on defaults here
  SPI4BUF;                  // clear the rx buffer by reading from it
  SPI4BRG = 0x4;            // baud rate to 8 MHz [SPI4BRG = (80000000/(2*desired))-1]
  SPI4STATbits.SPIROV = 0;  // clear the overflow bit
  SPI4CONbits.MODE32 = 0;   // use 16 bit mode
  SPI4CONbits.MODE16 = 1;
  SPI4CONbits.MSTEN = 1;    // master operation
  SPI4CONbits.ON = 1;       // turn on spi 4

  // Slave - SPI3, pins are: SDI3(D2), SDO3(D3), SCK3(D1), SS3(D9; not connected)
  SPI3BUF;                  // clear the rx buffer
  SPI3STATbits.SPIROV = 0;  // clear the overflow
  SPI3CONbits.MODE32 = 0;   // use 16 bit mode
  SPI3CONbits.MODE16 = 1;
  SPI3CONbits.MSTEN = 0;    // slave mode
  SPI3CONbits.ON = 1;       // turn spi on.  Note: in slave mode you do not set baud

  while(1) {
    unsigned short master = 0, slave = 0;
    unsigned short rmaster = 0, rslave = 0;
    NU32_WriteUART3("Enter two 16-bit hex words (lowercase) to send from ");
    NU32_WriteUART3("master and slave (i.e., 0xd13f 0xb075): \r\n");
    NU32_ReadUART3(buf, sizeof(buf));
    sscanf(buf,"%04hx %04hx",&master, &slave);
    // have the slave write its data to its SPI buffer
```

```
    // (note, the data will not be sent until the master performs a write)
    SPI3BUF = slave;
    // now the master performs a write
    SPI4BUF = master;
    // wait until the master receives the data
    while(!SPI4STATbits.SPIRBF) {
      ; // could check SPI3STAT instead; slave receives data same time as master
    }
    // receive the data
    rmaster = SPI4BUF;
    rslave = SPI3BUF;
    sprintf(buf,"Master sent 0x%04x,  Slave sent 0x%04x\r\n", master, slave);
    NU32_WriteUART3(buf);
    sprintf(buf," Slave read 0x%04x, Master read 0x%04x\r\n",rslave,rmaster);
    NU32_WriteUART3(buf);
  }
  return 0;
}
```

12.3.2 SRAM

One use for SPI is to add external RAM to the PIC32. For example, the Microchip 23K256 256 kbit (32 KB) static random-access memory (SRAM) has an SPI interface. The data sheet describes its communication protocol. The SRAM has three modes of operation: byte, page, and sequential. Byte operation allows reading or writing a single byte of RAM. In page mode, you can access one 32-byte page of RAM at a time. Finally, sequential mode allows writing or reading sequences of bytes, ignoring page boundaries. The example code we provide uses sequential mode.

The SRAM chip requires the use of slave select (called chip select $\overline{\text{CS}}$ on the 23K256). When this signal drops low, the SRAM knows that data or commands are about to be sent, and when $\overline{\text{CS}}$ becomes high, the SRAM knows that communication is finished. We control $\overline{\text{CS}}$ using a normal digital output pin, as its state is only changed after several bytes are sent, not after every byte is sent to the device (which is what the automatic slave select enable feature of the SPI peripheral would do). After wiring the chip according to Figure 12.3 you can run the

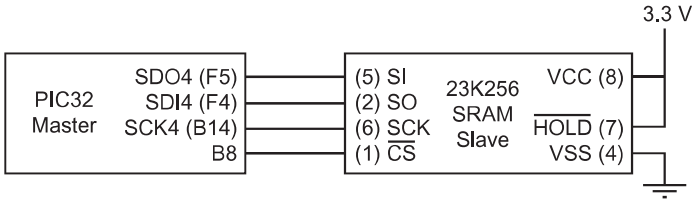

Figure 12.3
SRAM circuit diagram. SRAM pin numbers are given in parentheses.

following sample code, which writes data to the SRAM and reads it back, sending the results over UART to your computer.

The main function used by the sample code is `spi_io`, which writes a byte to the SPI port and reads the result. Every operation uses this command to communicate with the SRAM, since every write requires a read and vice versa. After configuring the SRAM to use sequential mode, the example reads the status of the SRAM, writes some data to it, and reads it back.

After executing this sample code, comment out the write to RAM, recompile and execute the code. Notice that, if you do not power off the SRAM, the SRAM still contains the data from the previous write, no matter how long between writes. Dynamic RAM, or DRAM, on the other hand, must periodically have its bits rewritten or it loses the data.

Code Sample 12.2 `spi_ram.c`**. SPI SRAM Access.**

```
#include "NU32.h"       // constants, funcs for startup and UART
// Demonstrates spi by accessing external ram
// PIC is the master, ram is the slave
// Uses microchip 23K256 ram chip (see the data sheet for protocol details)
// SDO4 -> SI (pin F5 -> pin 5)
// SDI4 -> SO (pin F4 -> pin 2)
// SCK4 -> SCK (pin B14 -> pin 6)
// SS4 -> CS (pin B8 -> pin 1)
// Additional SRAM connections
// Vss (Pin 4) -> ground
// Vcc (Pin 8) -> 3.3 V
// Hold (pin 7) -> 3.3 V (we don't use the hold function)
//
// Only uses the SRAM's sequential mode
//
#define CS LATBbits.LATB8       // chip select pin

// send a byte via spi and return the response
unsigned char spi_io(unsigned char o) {
  SPI4BUF = o;
  while(!SPI4STATbits.SPIRBF) { // wait to receive the byte
    ;
  }
  return SPI4BUF;
}

// initialize spi4 and the ram module
void ram_init() {
  // set up the chip select pin as an output
  // the chip select pin is used by the sram to indicate
  // when a command is beginning (clear CS to low) and when it
  // is ending (set CS high)
  TRISBbits.TRISB8 = 0;
  CS = 1;

  // Master - SPI4, pins are: SDI4(F4), SDO4(F5), SCK4(F13).
  // we manually control SS4 as a digital output (F12)
  // since the pic is just starting, we know that spi is off. We rely on defaults here
```

```
  // setup spi4
  SPI4CON = 0;              // turn off the spi module and reset it
  SPI4BUF;                  // clear the rx buffer by reading from it
  SPI4BRG = 0x3;            // baud rate to 10 MHz [SPI4BRG = (80000000/(2*desired))-1]
  SPI4STATbits.SPIROV = 0;  // clear the overflow bit
  SPI4CONbits.CKE = 1;      // data changes when clock goes from hi to lo (since CKP is 0)
  SPI4CONbits.MSTEN = 1;    // master operation
  SPI4CONbits.ON = 1;       // turn on spi 4

                            // send a ram set status command.
  CS = 0;                   // enable the ram
  spi_io(0x01);             // ram write status
  spi_io(0x41);             // sequential mode (mode = 0b01), hold disabled (hold = 0)
  CS = 1;                   // finish the command
}

// write len bytes to the ram, starting at the address addr
void ram_write(unsigned short addr, const char data[], int len) {
  int i = 0;
  CS = 0;                         // enable the ram by lowering the chip select line
  spi_io(0x2);                    // sequential write operation
  spi_io((addr & 0xFF00) >> 8 ); // most significant byte of address
  spi_io(addr & 0x00FF);          // the least significant address byte
  for(i = 0; i < len; ++i) {
    spi_io(data[i]);
  }
  CS = 1;                         // raise the chip select line, ending communication
}

// read len bytes from ram, starting at the address addr
void ram_read(unsigned short addr, char data[], int len) {
  int i = 0;
  CS = 0;
  spi_io(0x3);                    // ram read operation
  spi_io((addr & 0xFF00) >> 8);  // most significant address byte
  spi_io(addr & 0x00FF);          // least significant address byte
  for(i = 0; i < len; ++i) {
    data[i] = spi_io(0);          // read in the data
  }
  CS = 1;
}

int main(void) {
  unsigned short addr1 = 0x1234;                  // the address for writing the ram
  char data[] = "Help, I'm stuck in the RAM!";    // the test message
  char read[] = "****************************";    // buffer for reading from ram
  char buf[100];                                  // buffer for comm. with the user
  unsigned char status;                           // used to verify we set the status
  NU32_Startup();   // cache on, interrupts on, LED/button init, UART init
  ram_init();

  // check the ram status
  CS = 0;
  spi_io(0x5);                                      // ram read status command
  status = spi_io(0);                               // the actual status
  CS = 1;

  sprintf(buf, "Status 0x%x\r\n",status);
  NU32_WriteUART3(buf);
```

```
    sprintf(buf,"Writing \"%s\" to ram at address 0x%x\r\n", data, addr1);
    NU32_WriteUART3(buf);
                                                    // write the data to the ram
    ram_write(addr1, data, strlen(data) + 1);       // +1, to send the '\0' character
    ram_read(addr1, read, strlen(data) + 1);        // read the data back
    sprintf(buf,"Read \"%s\" from ram at address 0x%x\r\n", read, addr1);
    NU32_WriteUART3(buf);

    while(1) {
      ;
    }
    return 0;
}
```

12.3.3 LSM303D Accelerometer/Magnetometer

The STMicroelectronics LSM303D accelerometer/magnetometer, depicted in Figure 12.4, is a sensor that combines a three-axis accelerometer, a three-axis magnetometer, and a temperature sensor.[1] As a surface mount component, the accelerometer is difficult to work with; therefore, we use it with a breakout board from Pololu. The combination of an accelerometer and magnetometer is ideal for creating an electronic compass: the accelerometer gives tilt parameters relative to the earth, providing a reference frame for the magnetometer readings. The PIC32 can control this device using either SPI or I^2C, another communication method discussed in Chapter 13.

The sample code consists of three files: `accel.h`, `spi_accel.c`, and `accel.c`. The header file `accel.h` provides a rudimentary interface to the accelerometer. The function prototypes in `accel.h` are implemented using SPI in `spi_accel.c`. Thus you can think of `accel.h` and `spi_accel.c` together as making an LSM303D interface library. (In Chapter 13, we will make another version of the library by using I^2C to implement the function prototypes.)

The SPI peripheral is set for 10 MHz operation and, as in `spi_ram.c`, `spi_io` encapsulates the write/read behavior. The `main` file that uses the LSM303D library is `accel.c`. This code reads and displays the sensor values approximately once per second.

You can test the accelerometer readings by tilting the board in various directions. The accelerometer will always read 1 g of acceleration in the downward direction. If you rotate the board you should see the magnetic field readings change. You can test the temperature sensor by blowing on it: the heat from your breath will cause the reading to temporarily increase. To use this device as a magnetic compass you must perform a calibration; STMicroelectronics application note AN3192 provides a guide.

[1] This chip uses microelectromechanical systems (MEMS) to provide you with so many sensors in such a small package.

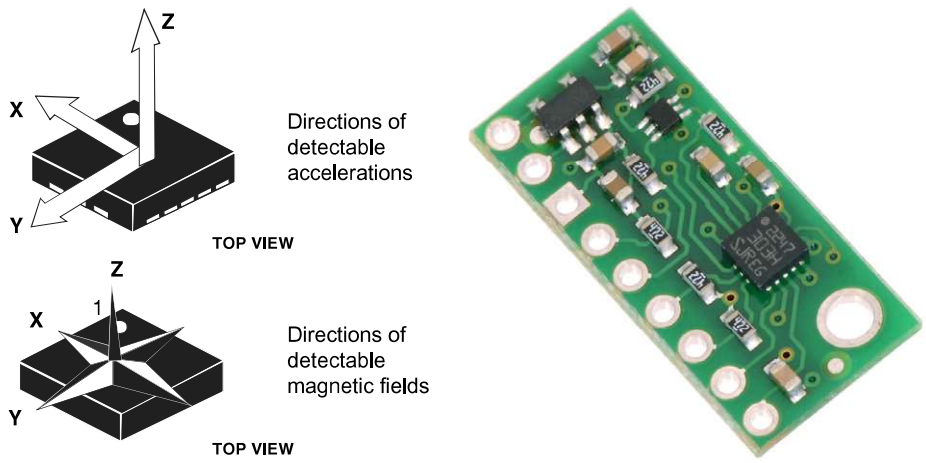

Figure 12.4
The LSM303D accelerometer on the Pololu breakout board. (Image of breakout board courtesy of Pololu Robotics and Electronics, pololu.com.)

Now that you are an expert in SPI, you can figure out the wiring yourself.

Code Sample 12.3 `accel.h`. **Header File Providing the Interface to the LSM303D Accelerometer/Magnetometer.**

```
#ifndef ACCEL__H__
#define ACCEL__H__
// Basic interface with an LSM303D accelerometer/compass.
// Used for both i2c and spi examples, but with different implementation (.c) files

                        // register addresses
#define CTRL1 0x20      // control register 1
#define CTRL5 0x24      // control register 5
#define CTRL7 0x26      // control register 7

#define OUT_X_L_A 0x28  // LSB of x-axis acceleration register.
                        // accel. registers are contiguous, this is the lowest address
#define OUT_X_L_M 0x08  // LSB of x-axis of magnetometer register

#define TEMP_OUT_L 0x05 // temperature sensor register

// read len bytes from the specified register into data[]
void acc_read_register(unsigned char reg, unsigned char data[], unsigned int len);

// write to the register
void acc_write_register(unsigned char reg, unsigned char data);

// initialize the accelerometer
void acc_setup();
#endif
```

Code Sample 12.4 accel.c. **Example Code that Reads the LSM303D and Prints the Results Over UART.**

```
#include "NU32.h" // constants, funcs for startup and UART
#include "accel.h"
// accelerometer/magnetometer example.  Prints the results from the sensor to the UART

int main() {
  char buffer[200];
  NU32_Startup(); // cache on, interrupts on, LED/button init, UART init
  acc_setup();

  short accels[3];   // accelerations for the 3 axes
  short mags[3];     // magnetometer readings for the 3 axes
  short temp;        // temperature reading
  while(1) {
    // read the accelerometer from all three axes
    // the accelerometer and the pic32 are both little endian by default
    // (the lowest address has the LSB)
    // the accelerations are 16-bit twos complement numbers, the same as a short
    acc_read_register(OUT_X_L_A, (unsigned char *)accels,6);

    // NOTE: the accelerometer is influenced by gravity,
    // meaning that, on earth, when stationary it measures gravity as a 1g acceleration
    // You could use this information to calibrate the readings into actual units
    sprintf(buffer,"x: %d y: %d z: %d\r\n",accels[0], accels[1], accels[2]);
    NU32_WriteUART3(buffer);

    // need to read all 6 bytes in one transaction to get an update.
    acc_read_register(OUT_X_L_M, (unsigned char *)mags, 6);

    sprintf(buffer, "xmag: %d ymag: %d zmag: %d \r\n",mags[0], mags[1], mags[2]);
    NU32_WriteUART3(buffer);

    // read the temperature data.  It's a right-justified 12-bit two's complement number
    acc_read_register(TEMP_OUT_L,(unsigned char *)&temp,2);
    sprintf(buffer,"temp: %d\r\n",temp);
    NU32_WriteUART3(buffer);

    //delay
   _CP0_SET_COUNT(0);
   while(_CP0_GET_COUNT() < 40000000) { ; }
  }
}
```

Code Sample 12.5 spi_accel.c. **Communicates with the LSM303D Accelerometer/Magnetometer Using SPI.**

```
#include "accel.h"
#include "NU32.h"
// interface with the LSM303D accelerometer/magnetometer using spi
// Wire GND to GND, VDD to 3.3V, Vin is disconnected (on Pololu breakout board)
// SDO4 (F5) ->  SDI (labeled SDA on Pololu board),
// SDI4 (F4) -> SDO
// SCK4 (B14) -> SCL
// RB8 -> CS
#define CS LATBbits.LATB8        // use RB8 as CS
```

```
// send a byte via spi and return the response
unsigned char spi_io(unsigned char o) {
  SPI4BUF = o;
  while(!SPI4STATbits.SPIRBF) { // wait to receive the byte
    ;
  }
  return SPI4BUF;
}

// read data from the accelerometer, given the starting register address.
// return the data in data
void acc_read_register(unsigned char reg, unsigned char data[], unsigned int len)
{
  unsigned int i;
  reg |= 0x80; // set the read bit (as per the accelerometer's protocol)
  if(len > 1) {
    reg |= 0x40; // set the address auto inc. bit (as per the accelerometer's protocol)
  }
  CS = 0;
  spi_io(reg);
  for(i = 0; i != len; ++i) {
    data[i] = spi_io(0); // read data from spi
  }
  CS = 1;
}

void acc_write_register(unsigned char reg, unsigned char data)
{
  CS = 0;               // bring CS low to activate SPI
  spi_io(reg);
  spi_io(data);
  CS = 1;               // complete the command
}

void acc_setup() {      // setup the accelerometer, using SPI 4
  TRISBbits.TRISB8 = 0;
  CS = 1;

  // Master - SPI4, pins are: SDI4(F4), SDO4(F5), SCK4(B14).
  // we manually control SS4 as a digital output (B8)
  // since the PIC is just starting, we know that spi is off. We rely on defaults here

  // setup SPI4
  SPI4CON = 0;              // turn off the SPI module and reset it
  SPI4BUF;                  // clear the rx buffer by reading from it
  SPI4BRG = 0x3;            // baud rate to 10MHz [SPI4BRG = (80000000/(2*desired))-1]
  SPI4STATbits.SPIROV = 0;  // clear the overflow bit
  SPI4CONbits.CKE = 1;      // data changes when clock goes from active to inactive
                            //    (high to low since CKP is 0)
  SPI4CONbits.MSTEN = 1;    // master operation
  SPI4CONbits.ON = 1;       // turn on SPI 4

  // set the accelerometer data rate to 1600 Hz. Do not update until we read values
  acc_write_register(CTRL1, 0xAF);

  // 50 Hz magnetometer, high resolution, temperature sensor on
  acc_write_register(CTRL5, 0xF0);

  // enable continuous reading of the magnetometer
  acc_write_register(CTRL7, 0x0);
}
```

12.4 Chapter Summary

- An SPI device can be either a master or a slave. Usually the PIC32 is configured as the master.
- Two-way communication requires at least three wires: a clock controlled by the master, master-output slave-input (MOSI), and master-input slave-output (MISO). Some slave devices also require their slave select pin to be actively controlled, even if they are the only slave on the bus. If there is more than one slave device on the SPI bus, then there must be one slave select line for every slave. The slave whose slave select line is held low by the master controls the MISO line.
- When the master performs a write, it generates a clock signal. This clock signal also signals the active slave to send data back to the master. Therefore, every write by the master should be followed by a read, even if you do not need the data.
- Master writes to the MOSI line are initiated by writing data to SPIxBUF. Received data is obtained by reading SPIxBUF, which actually gives you access to data in the receive buffer SPIxRXB.

12.5 Exercises

1. Why must you write to the SPI bus in order to read a value from the slave?
2. Is it possible to use only two wires (plus GND) if you need to only read or only write? Why or why not?
3. Write a program that receives bytes from the terminal emulator, sends the bytes by SPI to an external chip, receives bytes back from the external chip, and sends the received bytes to the terminal emulator for display. The program should also allow the user to toggle the $\overline{\text{SS}}$ line. Such a program may prove useful when working with an unfamiliar chip. Note: you can read and write hexadecimal numbers using the `%x` format specifier with `sscanf` and `sprintf`.

Further Reading

23A256/23K256 256K SPI bus low-power serial SRAM. (2011). Microchip Technology Inc.

AN3192 using LSM303DLH for a tilt compensated electronic compass. (2010). STMicroelectronics.

LSM303D ultra compact high performance e-compass 3D accelerometer and 3D magnetometer module. (2012). STMicroelectronics.