

Serial Peripheral Interface Communication

Jeremy Konz

Team 9: Safety Enhancement Features for the 21st Century Automobile
ECE 480: Capstone Design
Michigan State University

November 14, 2008

Contents

1	Executive Summary	1
2	Objective	2
3	Development	2
3.1	Basic Structure	2
3.2	The Microchip SPI Module	3
3.3	SPI Module Implementation in C	4
4	Issues	7
5	Conclusion	7

1 Executive Summary

In most modern circuitry it is inevitable that it will become necessary to incorporate more than a single chip in design. When more than a single piece of silicon is required, it is also essential that the several different components are able to communicate with each other. There are two main forms of communication, serial and parallel. In a parallel implementation, there are as many connections as there are bits that need to be sent and received. Parallel communication busses are extremely fast, however become extremely large and consume many of the valuable pins needed by microprocessors.

For the reason of space and pin consumption, serial communication is preferred over parallel when connecting two or more chips, consuming only two to four pins. A serial communication bus functions by sending 0's and 1's sequentially over one or two wires. In most fully featured modern microcontrollers (μC) it is standard to have a hardware module that controls the Serial Peripheral Interface (SPI) clock, receive buffer, and transmit buffer. However in some smaller or less featured μC 's it is required to write one's own SPI driver using typical I/O pins. The information is sampled by the receiver based on a serial clock when the active edge of the clock rises. This sample is then digitized into a single bit of information. In this manner it is possible to rate the speed of a serial bus based on the clock frequency associated with it. These frequencies can range from a couple KHz to 20MHz like the SPI bus that will be covered in this application note.

2 Objective

Upon completion of reading this document, one should be understanding of how a SPI bus is structured and how to maximize functionality in Microchip[©] hardware modules. This will be accomplished by first covering in detail the timing of the SPI bus, then the functionality of the hardware SPI module, and finally how to implement this system in the C programming language.

3 Development

3.1 Basic Structure

Before one can get going using the SPI bus to have communication between chips, it is first essential to understand some basics of how to configure the module. Figure 1 shows a very basic rundown of a single 8-bit transfer cycle. CKPOL stands for the clock polarity, and CKPHA refers to the clock being active high. MOSI/MISO refers to the serial in and serial out lines, and how they are being sampled on their respective pins. The $\overline{SSEL}$ (signal select) signal is called the $\overline{CS}$ in some Microchip[©] implementations and stands for chip select. When this signal is low, that chip is active. This is useful for connecting multiple different components to the same SPI bus, because when the $\overline{CS}$ signal is high the chip is disabled.

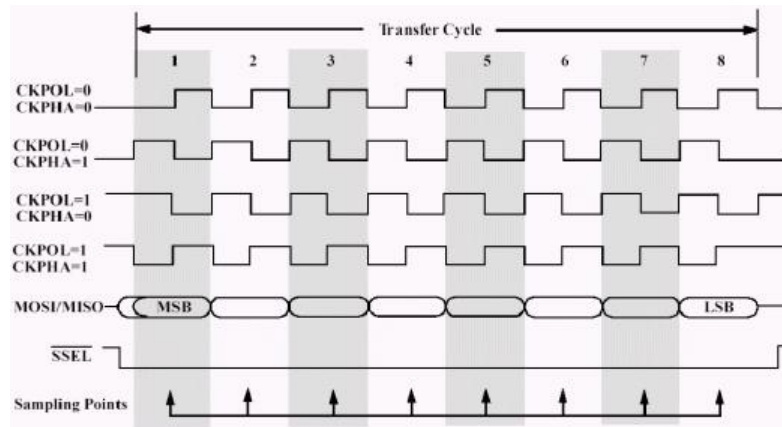


Figure 1: Diagram showing different timing modes of transmission. Taken from http://www.maxim-ic.com/appnotes.cfm/an_pk/3078

3.2 The Microchip SPI Module

In microcontrollers with a dedicated SPI module, manufacturers make implementation simpler by having dedicated read and write buffers connected directly to the input and output pins. When data is written to the transmit buffer, the module automatically begins functionality by outputting a clock frequency (directly related to the operating frequency of the μC) and on each active clock edge, shifts one bit onto the output line. During the shift out of information on the clock edge, the receive pin shifts one bit of information into the receive buffer. Should the receive buffer not be empty because it was not cleared by the program, an overflow flag will trigger and the new data will be discarded. This manner of shifting in at the same time as shifting out allows for extremely fast communication.

When one transmit/receive cycle is complete the module triggers an interrupt. The interrupt controls the main functionality of the SPI bus. At this point in time, the transmit buffer will be empty and the receive buffer will be full. This means that two things need to happen in order to continue operation, the receive buffer must be read and the transmit buffer needs to be filled. It is important to note that the transmit and receive buffers are encapsulated in a larger buffer called SPI1BUF. When information is assigned to SPI1BUF it


```
void SPI_INIT(void)
{
    IFS0bits.SPI1IF = 0;    //Clear the Interrupt Flag
    IEC0bits.SPI1IE = 0;    //Disable the Interrupt

    SPI1CONbits.DISSDO = 0; //Use SD01, not as I/O
    SPI1CONbits.MODE16 = 1; //Communication is word-wide (16 bits).
    SPI1CONbits.SMP = 0;    //Input Data is sampled at the middle of data output time.
    SPI1CONbits.CKE = 0;    //Output data changes on idle to active clock state
    SPI1CONbits.CKP = 0;    //Active state is a clock high
    SPI1CONbits.MSTEN = 1;  //Master Mode Enabled
    SPI1STATbits.SPIEN = 1; //Enable SPI Module
    SPI1BUF = 0x0000;       //Write junk data to start the module

    IFS0bits.SPI1IF = 0;    //Clear the Interrupt Flag
    IEC0bits.SPI1IE = 1;    //Enable the Interrupt
}
```

Figure 3: SPI module initialization.

by the manufacturer. In this example the μ C in use is the dsPIC30F3014, and therefore it is necessary to have this line at the top of the *.c file:

```
#include <p30f3014.h>
```

After the correct file is included which is provided by the manufacturer, it is necessary to set all of the control register bits so that the SPI module functions as one might expect it to. The following code is assigning values to the control registers associated with the module, as specified by the manufacturer in the reference manual for the dsPIC30F3014. The function in figure 3, SPI_INIT was not provided by the manufacturer, it was written to improve readability of code.

```
void _ISRFAST _SPI1Interrupt(void)
{
if (SPI1STATbits.SPIRBF) //as long as the read buffer is actually full
{
rxinfo = SPI1BUF; //read input buffer
}

if (SPI1STATbits.SPIRBF) //as long as the transmit buffer is actually empty
{
SPI1BUF = txinfo; //write to output buffer
}

/* Insert action code here */

IFS0bits.SPI1IF = 0; //Clear the Interrupt Flag
}
```

Figure 4: SPI module initialization.

With the module initialized and functioning, the next step is to prepare the interrupt so that with each execution of the interrupt, both the receive buffer will be emptied and the transmit buffer will be filled. Of course one must be sure the buffers are full and empty, respectively, before performing these operations. This will vary by application, and only a rough outline is given in figure 4. Remember that when SPI1BUF is read, the information is really being read from the Rx buffer. And when SPI1BUF is assigned, the information is really being assigned to the Tx buffer.

With the initialization and the interrupt written, all that is left is deciding what to do with the data you want to transmit and receive. It is ideal that the SPI bus can be managed by interrupts because this allows for other sections of code to execute while transmission is taking place in hardware.

4 Issues

There is a few pitfalls to watch out for when attempting to implement a SPI communication bus. One of these issues is wiring. It is important to make sure that the serial input on the μC is connected to the serial output of the device with which communications is occurring. Also, many devices such as external EEPROM's have pins that can disable the device selectively (for use in larger systems), it is important to take note of pins that are not being used as well as pins that are in use.

The next area to observe is to be sure that the interrupt is actually occurring as one might expect. It is sometimes difficult to observe phenomena that is not executing as planned on a microcontroller. Implementing either visual notifications on error (perhaps with an LED), or using an in-circuit debugger is essential. Should the interrupt occur too often this can hinder operation of other code, and likewise if it does not occur often enough the SPI bus will not be as active as expected.

5 Conclusion

Serial communication between components is a vital part to reducing size of circuitry, and is used in almost every gadget in todays world. Knowing how to take advantage of bus systems such as SPI is a vital part in becoming a well rounded embedded systems programmer.