# Lab 8 – Serial Peripheral Interface and the SD card

## Objectives

The objectives for this lab are to:

1. communicate to an external device using a serial bus
2. use a externally pre-compiled module in your project
3. use a *typedef* and *struct* to declare a new datatype for your program

## Background

This lab focuses on using a serial bus like the SPI bus to communicate with a device such as the Secure Digital (SD) card reader. In Lab 7 you saw how to communicate with the LCD display using a parallel bus, and saw that these type of interfaces can require a lot of wiring. The various serial buses such as SPI, I2C, etc., trade the complexity of wiring and a simpler interface for higher bandwidth/throughput.

The modern SD cards used in electronics such as MP3 players, digital cameras and cell phones are quite miraculous. They have extremely high storage densities and transfer speeds. Compression technologies such as JPEG, PNG, and MPEG have extended the capabilities of these devices for storing data. A typical three minute MP3 file, sampled at 128Kbps, can take less than 4MB. This means even a modest (by this week's standards) 1GB memory card can hold over 250 songs, roughly the equivalent of audio 10 CDs. You remember audio CDs, right? You've *heard of* audio CDs, right? (If not, ask your parents to show you some. They're not a myth).

SD cards are an extension of an earlier memory card technology called MultiMediaCard (MMC). Although MMC cards still exist they have largely been replaced by SD cards. There are many variations of SD cards; full size, mini size, and micro size (which is the type we will be using). There are also SDHC (high capacity) cards with capacities larger than 4GB, and the cards have support for digital rights management (DRM) embedded in them. SD cards support three transfers modes: one-bit SD, four-bit SD, and SPI. Not all microSD cards support SPI, but the ones we will work with do. The SD interface specifications were originally not publicly available, but after the details were reverse-engineered for the non-DRM sections of the card some specs were made available for no charge.

The SPI interface has three or four wires. We will be using only three, hence we will be in three-wire mode. The wires are the clock, data out, and data in. The theory behind SPI is simple and kind of neat. On each clock cycle, the master (the device generating the clock) and the slave exchange one bit. Master sends data on its master-out-slave-in (MOSI) pin while the slave sends data on the master-in-slave-out (MISO) pin. Now, usually the master needs to tell the slave what's going on first, so the first byte or bytes the master sends is a control command for the slave. While the command is being sent,

the slave probably doesn't have any useful data to send back to the master, so the data coming on MISO is ignored. Similarly, when the slave is transmitting to the master, there normally isn't anything else that needs to be sent to the slave, so the bits on MOSI are ignored by the slave.

This “non-transfer” of data is important to you for a number of reasons. First, the SPI0 module of the C8051, like SPI modules in most other microcontrollers, handles the SPI clock and transfer of data in/out for you. To start an 8-bit transfer, all you need to do is write to the SPI0DAT SFR. This makes complete sense when you want to write to the slave, but what do you do when you want to *read*? Well, you also write to SPI0DAT. This obviously causes data to be transferred to the slave device, but simultaneously it causes data to be transferred from the slave device to the microcontroller. You wait for the data to be written, and then read from SPI0DAT. It now contains the data sent by the slave on the MISO pin.

Data on the SD card is organized in 512-byte blocks or *sectors*. When we want to read or write, we always transfer a sector. (Not so ironically, it's a similar process to sending data to the DAC.) We write a byte to SPI0DAT, wait for it to transfer, and repeat this 511 more times. The only real difference is that we need to tell the SD card which sector we wish to transfer. A command is sent prior to the read or write to inform the card. This is very similar to reading and writing to a memory address in RAM. The difference is we're doing the transfer in chunks, so we only specify the starting address for the read or write. After reading or writing a byte, the logic on the SD card increments the memory address so that the next byte transferred comes or goes to that address.

Like other modules within the C8051, the SPI interface can use interrupts to let us know when a byte has been transferred. The SPIF flag is set at the end of a transfer, and if interrupts are enabled then our ISR is called. In this instance, however, we'll use good old-fashioned polling. Why? Most of the time our program spends interfacing with the SD card will be when it is reading a sector. We take advantage of the high speed of our memory card, and so will transfer each sector in a very short amount of time. Since this time is much shorter compared to the sound output frequency, and since the sound output is already handled by the Timer 2 overflow, we can busy-wait the short amount of time needed for each SPI transfer. It also alleviates the overhead of the interrupt itself. If the interrupt is going off every few microseconds, we may be spending more time saving and restoring registers than doing anything else.

What this all boils down to is your code to transfer a sector will be something like:

```
assert CS on the SD card;
for (i = buflen; i != 0; --i ) {
    SPIF = 0;
    SPI0DAT = 0xff;
    while (SPIF == 0) {}
    *ptr = SPI0DAT;
    ++ptr;
}
deassert CS on the SD card;
```

The app note “Secure Digital Card Interface for the MSP430” by F. Foust at Michgan State University gives an excellent overview of the SPI protocol to communicate with an SD card. We use the code presented in that app note, heavily modified to work with our C8051. The protocols for interfacing to the SD card are not our focus for this lab; we will instead focus on the actual SPI interface and how to set configure it so that we can use the code for the SD card. All we need to know about this module is that it provides two procedures which you will call from your main program:

- **uint8 microSDinit( void )**: initializes the SD card. It is called after you have initialized the SPI interface, as it uses the bus to prepare the card for reading. It first calls **spi_set_divisor()** to set the SPI clock to the rate needed for initialization. If the initialization fails, it returns 0. Otherwise it calls **spi_set_divisor()** for full-speed operation and returns a 1.
- **uint8 microSDread( uint32 blockaddr, uint8 xdata *buffer )**: this is the primary procedure you will use – repeatedly. It is passed a sector number on the SD card and a pointer to a 512-byte buffer. It will attempt to read a sector from the SD card. If the read is successful it will return a non-zero value, otherwise it returns a zero.

## Pre-lab

Prior to the lab, you and your lab partner should turn in a “pre-lab” which answers the following questions. Without the answers to these questions, you will not be able to get this lab working correctly.

1. What SFR page contains the SPI SFRs?
2. When initializing the SD card, a series of clock pulses are sent on the SCK pin at 400KHz. What setting is necessary for SPI0CKR to set the data transfer rate to approximately 400KHz? To calculate this value use using the setting for SYSCLK in our project.
3. What is the maximum clock frequency you can get on SCK?
4. How is three-wire mode set on the C8051? (Note: you should do this after putting the SPI module into master mode, otherwise it may have no effect).
5. The SCK, MOSI and CS pins for the microSD card adapter need to be set to push-pull. What need to be done to P0MDOUT and P2MDOUT to accomplish this *and only this*?

## Lab work

Your goal for Lab 8 is to locate two things on your SD card:

1. a string which should be displayed on the LCD display
2. data for (you guessed it) a waveform to be sent to the DAC

During lab, you will be told a specific sector on your SD card from which you will read one sector. The sector will contains a null-terminated string, an unsigned short number, and a unsigned character array. You can create a *typedef struct* for this data:

```
typedef struct {
    char string[16];
    unsigned short len;
    unsigned char wavedata[512];
} SD_test_data;
```

This structure is larger than 512 bytes (a sector size), but the size of the wave data will not be 512 bytes so reading one sector will read everything you need for this test. You will read the specified sector from the SD card into a variable of this type, display the *string* on your LCD display and repeatedly send the *wavedata* to the DAC (just like was done in Labs 5 and 6). If your SPI code works correctly, you pass the test based on the results displayed on the LCD and DAC.

The microSD card adapter uses the following pin-out:

- SCK – P0.0 (SPI SCK signal)
- DO – P0.1 (SPI MISO signal)
- DI – P0.2 (SPI MOSI signal)
- CS – P2.0: This signal is active low. It is manually controlled in your code, by the **spi_cs_assert()** and **spi_cs_deassert()** procedures.
- CD – P2.1: The active level of this signal is left for you to determine. It is read in your **spicardpresent()** procedure.

The microSD card uses 3.3V for power. You should only connect this to the 3.3V from the C8051F120 board (pin A1). **Please, please, please, please, please, please, please DO NOT** connect it to the 5V power on your board. That would be very bad and would make me sad. Plus you will owe me a new microSD card.

As stated earlier, you will download **microsd.obj** from Sakai; it performs the communication with the SD card. Add it into the project as you have for other object files.

These are the procedures you need to write. Only two are called directly from the main module:

- **void spiinit( void )** – initializes the SPI module. It does not need to set the SPI clock speed.
- **uint8 spicardpresent( void )** – returns the status of the CD signal from the microSD card adapter. It should return true (non-zero) when there is a card in the adapter.

The remaining procedures are called from the SD card module (note, these procedures all have underscores in their names):

- **void spi_set_divisor( uint8 spd )** – sets the SPI clock speed. If *spd* is 0, the clock is set to approximately 400KHz; otherwise the clock is set to maximum speed.
- **void spi_cs_assert( void )** – asserts the CS chip select signal line (sets it low).
- **void spi_cs_deassert( void )** – de-asserts the CS chip select signal line (sets it high).

- **void spi_send_byte( uint8 input )** – sends one byte over the SPI bus.
- **uint8 spi_rcv_byte( void )** – receives one byte over the SPI bus.
- **void spi_rcv_buffer( uint16 len, uint8 xdata *buffer )** – sends multiple bytes (normally a SD card sector) over the SPI bus. The code for this procedure was given earlier.

You will also write your own main C module for this lab. It should perform the general initialization of the C8051, call the initialization procedures for each module (keyboard, DAC, LCD, and SPI). Also set the DAC rate to 11.025KHz and the DAC stereo for mono. Then it should go into an infinite loop that

1. waits for a microSD card to be inserted,
2. initialize the microSD card, and if not successful print a message on the LCD; otherwise
   (a) reads the data from the microSD card,
   (b) clears the LCD and displays the string,
   (c) while the microSD card is inserted, outputs the wave data to the DAC

When completed, turn in your **spi.c**, **spi.h** and **sd.h** files (yes, you need to make header files for the SPI and microSD modules which include a minimum of the function prototypes for each).