SD card read/write Using PIC18F4550

In this tutorial, we will be learning how to program the PIC18F4550 to perform the raw SD read/write functions on an SD card. To implement this hobby project, we will be using the popular SPI protocol and the software UART to display the output.

Microchip's PIC18F4550 is a small microcontroller in terms of its functionalities as well as the memory space. To design a code to implement the raw SD read/write using this microcontroller requires a few considerations. In our previous tutorial, we've discussed the process of SD card interfacing with microcontrollers.

To begin with, it only has a 32KB program memory and 2048 bytes of data memory. One of the features of SD card is it has a limited memory space is available when coding the program.
We're going to use UART for debugging the SD card. The Receive pin(Rx) of the UART and Serial Data Out (SDO) pin of SPI are sharing the same port pin of PIC18F4550 and here the software UART is used instead of a hardware interrupt.

Interfacing

The first step is to interface the hardware with the software using the microchip's library function, OpenSPI( ). The OpenLab uses RB4 as chip select(CS) or the slave select(SS) pin instead of the usual RA5 pin. The direction of RA5 must be set as output and keep it as low in order to apply this change. The RB4 pin must be set as an output pin and should be used as the chip select pin.

You can choose the CS pin as you desire. The SPI pins RB4 (CS), RB1(SCK) and RB0(SDI) are analog pins by default. So the pins configured to digital pins. Set the SPI at mode(0,0) and in the 8-bit asynchronous mode with the input sampled at the middle of the data out. The directions of the SPI pins apart from the adopted RB4 pin are set in

```
void soft_hard_init()
{
    unsigned char sync_mode=0;
    unsigned char bus_mode=0;
    unsigned char smp_phase=0;

    CloseSPI();

    ADCON1 = 0X0F;   //Make RB4, RB1, RB0 Pins Digital
    CMCON |= 7;

    TRISA5 = 0; //Work around for a hardware adjustment in Open Lab
    LATA5 = 0;
    TRISB4 = 0; //Chip Select

    sync_mode = SPI_FOSC_64; // 8 Bit Asynchronous Mode
    bus_mode = MODE_00; // Mode(0,0)
    smp_phase = SMPMID; //Input data sample at middle of data out
    OpenSPI(sync_mode, bus_mode, smp_phase);
}
```

Transmission

Next, we must create the basic functions to carry out the transmission and reception of data and command with the SD card. The function WriteSPI( ) is used to transmit command or data to SD card. The function ReadSPI( ) is used to read every datum or response that is received from the SD card. While receiving a response from the SD card, it is important to make sure that the chip select pin must be low prior to reading.

```
unsigned char response()
{
    unsigned char buff;
    LATB4 = 0;          //CS low
    buff=ReadSPI();
    return buff;
}
```

Reception

There is an exception while using Microchip's ReadSPI( ) here.

```
unsigned char ReadSPI( void )
{
```

```
  SSPBUF = 0x00;           // initiate bus cycle
  //while ( !SSPSTATbits.BF );                 // wait until cycle complete
  while(!PIR1bits.SSPIF);  // wait until cycle complete
  return ( SSPBUF );       // return with byte read
}
```

The above code shows the ReadSPI( ) microchip library function. In line 6, 0x00 byte write to the SSPBUF register to initiate a bus cycle. The microcontroller generates the bus cycle only after the buffer is filled. Writing 0x00 does not fill the buffer. Rather, it clears the buffer. A 0xFF write to SSPBUF register in order to generate a bus cycle before the arriving of response. So we need to use the modified version of the ReadSPI() function in our code.

The command has a fixed length of 6 bytes. A dummy byte (0xFF) is sent in order to generate the bus cycle.

```
void Command(char cmd, unsigned long int arg, char CRCbits)
{
    unsigned char temp;
    WriteSPI(0xff);  //Dummy Byte
    WriteSPI(cmd);
    WriteSPI((unsigned char) (arg >> 24));
    WriteSPI((unsigned char) (arg >> 16));
    WriteSPI((unsigned char) (arg >> 8));
    WriteSPI((unsigned char)  arg);
    WriteSPI(CRCbits);
}
```

First, transmitting the argument ‘ cmd’, which contains a single byte hexadecimal command number. The variable ‘arg’ contains the 4-byte argument, It transmitted as 8 bits starting from the most significant bit (MSB) at a time and finally sents the CRC bit. Before starting the initialization process, we must create a function called dummy_clocks( ) and another function called proceed( ). The purpose of dummy_clocks( ) is to send bus clocks to the SD card.

```
void dummy_clocks(int n)
{
    int i = 0;

    for(i=0;i<n;i++)
    {
        LATB4 = 1; //CS High
        WriteSPI(0XFF); //Dummy Byte
        LATB4 = 0; //CS Low
```

Set the chip select to high to send each dummy byte. Now, the purpose of proceed( ) is to send bus clocks at instances where the buffer of the microcontroller is empty during the execution time.

```
void proceed()
{
    LATB4 = 0; //CS Low
    WriteSPI(0XFF); // Give Time For SD_CARD To Proceed
    LATB4 = 1; //CS High
}
```

As you may notice, there is a difference between the two functions all though they may appear similar. To generate the clock pulses using the function proceed(). Sent the 0xFF byte to the buffer and it does not send to the SD card. To write the 0xFF byte turn off the slave selection.

While in the case of dummy_clocks( ), our purpose is to send clock pulses to the SD card soon after power-up or prior to sending a command, to prepare the SD card for the appropriate communication frequency. At this point, writing a 0xFF does not actually mean anything to the SD card.

For a detailed explanation of the timing and flow diagram of SD card initialization, read and write process, please refer the tutorial: Interfacing Microcontrollers with SD Card

SD Card Initialization

After power up, wait for one second or two seconds. In the initialization process, the first step is to make the card to expose to the frequency at which the data is being sent. And then reset the SPI mode from its default operating mode. For this, send around 75 dummy bits approximately (dummy byte * 10 times = 80 bits) and the card will be ready to communicate in the required frequency.
The first command to send is CMD0. Which resets the SD card and set the card at an idle state.

```
do{
        dummy_clocks(8);    // Initial Clock Pulse
        Command(0X40, 0X00000000, 0X95);
        proceed();    // Generate Bus Clock
        do{
            buff = response();
```

```
    }while(buff!=0X01);
```

The loop **do-while** continues to runs until the card is inserted into SD card slot. And it receives an idle response (0x01). At this point, the SD card is in an idle state and will only accept the commands, CMD0, CMD1, ACMD41, CMD58 and CMD59.

Sent the command CMD8 is after the checking the version of SD card. If the SD card is Version 2, then continuously sent the CMD1 command after receiving the response of the CMD0 command. Keep sending the CMD1 command till the SD card gets out of its idle mode and returns a 0x00 byte as a response.

```
do{
    Command(0X41, 0X00000000, 0X00);
    proceed();     //Generate Bus Clock
    do{
        buff = response();
        count++;
    }while((buff!=0X00) && (count<10));
    count = 0;
    }while(buff!=0x00);
```

To read or write the raw data first accept the command CMD1. There is also the pat.bility that the SDC requires a CMD41 command to initialize the card instead of the CMD1 command. Send the command CMD41 to write a general code. Sent the command CMD1 if the command CMD41 is to reject. Sent the command CMD1 if any rejection of the command CMD41 has occurred.

The CMD<n> command is CMD55 command followed by CMD<n>. So follow the same procedure to send both the commands i.e. CMD55 and wait for a 0x01 response and then CMD41 and wait for a 0x00 response. Send command CMD1 otherwise after a definite amount of time the command will not accept.

Block Length

After the initialization is complete, set the block length. The command to set the block length is CMD16 and it takes the argument as the size of the block. Wait for the command to accept.

The initial command is the same to send for raw data read/write. first, we will look at how to write a single block.
First, send the command CMD18 with the argument. The argument contains the address of the sector. The command is to accept with a usual response of 0x00 byte.

Then the SD card is waiting for the microcontroller to give a **start token** of 0xFE. This indicates the beginning of the block write and the card would be ready to accept a block of data. After block write completes then send two 0xFF bytes as CRC. It is mandatory to send the CMD13 command after the completion of every data block write.

```
dummy_clocks(8);
    Command(0X58, 0X00000A00, 0X00);
    proceed();
    do{
        buff = response();
      }while(buff!=0x00);

     /*CMD 24 Accepted*/

      Delay_s(1);

      dummy_clocks(8);
      WriteSPI_(0XFE); //START TOKEN

      for(i=0;i<512;i++) //DATA BLOCK
      {
          WriteSPI_(/*WRITE YOUR DATA BYTE HERE*/);
          i++;
      }

      WriteSPI_(0XFF); //CRC
      WriteSPI_(0XFF); //CRC

      Delay_s(1);

      dummy_clocks(8);
      Command(0X4D,0X00000000,0XFF);

    proceed();
      do{
        buff = response();
        }while(buff!=0x00);

     /*DATA WRITE OVER*/
```

Next, we will see how to read a block of data from an SD Card. First, send the command CMD24 and wait for the command to accept. Then wait for its response. The microcontroller immediately receives the command 0xFE from the SD card. It is the indication of the start of data transmission. Set the chip select pin as low and remains the same throughout the process. The command CMD12 indicate the end of the reading process and it sent a block of data at the end.

```
do{
       buff = response();
     }while(buff!=0xFE);

         length = 0;
         LATB4 = 0; // CS is driven low during the read process

           while(length<512)
           {
               str[length]=response();  //read buffer (R1) should be 0x01 = idle mode
               length++;
           }
         length = 0;

         WriteUART_(0XFF); //CRC
         WriteUART_(0XFF); //CRC

           Delay_s(1);
           dummy_clocks(8);

         Command(0X4C,0X00000000,0X00); //COMMAND12. Stop To Read
         proceed();
           do{
               buff = response();
             }while(buff!=0xFF);
```

Multi-Block write

To write **multi-block** use the command CMD25. The 0xFC byte indicating **start token** indicating in the case of a multi-block write.

```
while(i!=/*number of blocks*/)
      {
      WriteSPI_(0XFC); //START TOKEN
      i++;
      for(j=0;j<512;j++) //DATA BLOCK
      {
          WriteSPI_(/*your data byte*/);
```

```
WriteSPI_(0XFF);
Delay_s(1);
dummy_clocks(8);
}
```

Finally, once the data write is over, issue a **stop token** 0x4D to indicate the end of the transmission. In the case of a multi-read, wait till the SD Card issues a 0xFE byte(**start read)** after accepting the CMD18 command.

Multi-Block read

Now, continuously keep reading the data bytes until the end of the specified number of blocks. The readings are as one block at a time because of space limitation. And collects every block of data as different array in a static declaration. And using a 0x4C as **stop token** at the end of the reading process.

```
while(count){
        while ( length < 512 )
        {
            str[length] = response(); //Repeat with a different array for each count
            rd++;
            length++;
        }
        count--;
        }
```

Finally, create a switch condition for the user to choose between the different functions of the system.

Spread the love, share this

- SPI, I2C interfacing with code
- 150+ premium articles, code library, online courses

EXPLORE NOW