PIC32 DEVELOPMENT

-- SD CARD LIBRARY

A Design Project Report

Presented to the School of Electrical and Computer Engineering of

Cornell University in Partial Fulfillment of the Requirements for

the Degree of Master of Engineering, Electrical and Computer Engineering

Submitted by

Chang Liu (cl2428)

Pei Xu (px29)

MEng Field Advisor: Bruce R Land (brl4)

Degree Date: May 2017

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: PIC32 development -- SD card Library

Author: Chang Liu, Pei Xu

Abstract: This project aims to design and develop a secure digital (SD) card library based on PIC32 microcontroller. The main function of this system is to read and store files from the SD card. In addition, this system gives PIC32 developers access to large memory to store image and files. It also serves for later projects need SD card implementation. Thus, by using the library, the later PIC32 developers can get the information and write data to the SD card easily. The basic functions in the SD card library are write and read functions. The user can access the file stored in the SD card with calling a read function in the library.

Individual Contribution

Chang Liu

He researched the way how SD card communicate with the microcontroller. He created the SPI communication from PIC32 to SD card. Then he coded sd_routines.c, sd_routines.h and created a user test interface using UART.

Pei Xu

She set up the hardware connection of SD card and PIC32. Together with Chang, Pei researched the way of implementing FAT file system on the SD card. Pei and Chang worked together finishing the code of fat32.c and fat32.h.

Executive Summary

The current situation in ECE4760 PIC32 developers is that there is a lack of library for them to directly access the file stored in an SD card. To enhance the feasibility and capability of the use of PIC32, a SD card library is needed to be created. Therefore, the developers are able to read, write or update information in the system directly.

According to our research, including the secondary research on the internet, we find that it is feasible and potential to enrich this peripheral for PIC32 developers. This improvement will contribute to the convenience for PIC32 developers in their work. Thus, this project aims to design and develop a secure digital (SD) card library based on PIC32 microcontroller.

The SD card library offers a place to store data, images, sound and other information which needs of large memory space. The main function of the library is to read and store files from the SD card. In addition, this library provides the functionality to get the file list from the root directory.

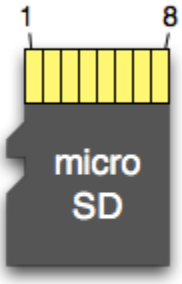
A user test interface is built based on the communication from computer and PIC32 via UART. Read or Write function selection and other basic functions can be selected from the user interface. By typing the command on the test console, users can choose the mode, select the files to open, read or write data to the file. As the SD card library is implemented separately with the TFT screen, an independent SD card slot is used to design the hardware.

Various tests are designed to verify the functionality of the SD card library system. By checking the data and information from both computer and PIC32, the tests guaranteed the correctness of each function our group designed.

Table of Contents

1. Introduction

SD card is a common daily life erasable storage device, because of its large storage capacity and low price, it is widely used in digital cameras, mobile phones and other digital products. SD card supports two bus modes: SD mode and SPI mode. SD mode using 6-wire buses, the use of CLK, CMD, DAT0, DAT1, DAT2, DAT3 for data communication, which has the data transform rate at 4bits at a time. SPI mode using 4-wire buses, the use of CS, CLK, DataIn, DataOut, these four ports for exchanging data only has 1 bit at a time which is slower than the SD mode, but the communication protocol is simple and there is no need to check the CRC, which is desirable for this project to read and write operations on the SD card。

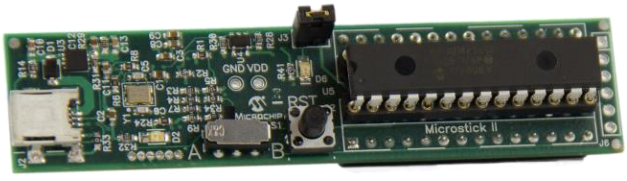


Pin	SD	SPI
1	DAT2	X
2	CD/DAT3	CS
3	CMD	DI
4	VDD	VDD
5	CLK	SCLK
6	VSS	VSS
7	DAT0	DO
8	DAT1	X

Figure 1. Pinout description of SD card

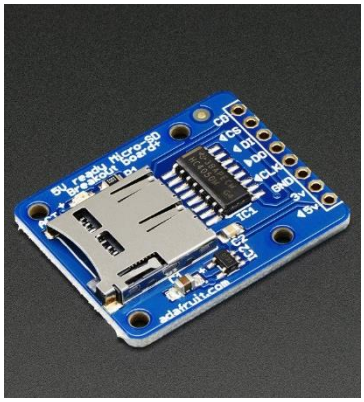
2. Design Alternatives

2.1 Components



MicrostickII

8GB SDHC card



SD card socket

UART Cable

Figure 2. Components of the design

2.2 Project budget

To interface the SD card with PIC32 microcontroller, the following parts and devices are needed to build the circuit. And the total cost of the project has a budget of 50$.

- UART cable (5.00$)
- Bread board (10.00$)
- MicrostickII Pic32 kit (10.00$)

- ❖ Jumper wires 20 pieces (1.00$)
- ❖ SanDisk 8GB SDHC card (7.99$)
- ❖ Standard Adfruit SD card socket (8.99$)

Total cost is 42.98$.

3. System Design

The SD card contains two basic semiconductor sections, a ‘memory core’ and a ‘SD card controller’. The ‘memory core’ is the flash memory region where the actual data of the file is saved. When we format the SD card a file system will be written into this region. Hence this is the region where the file system exists. The ‘SD card controller’ helps to communicate the ‘memory core’ with the external devices like microcontrollers. It can respond to certain set of standard SD commands and read or write data from the memory core in for the external device. Thus, the ‘SD card controller’ is the device our PIC32 would communicate with.

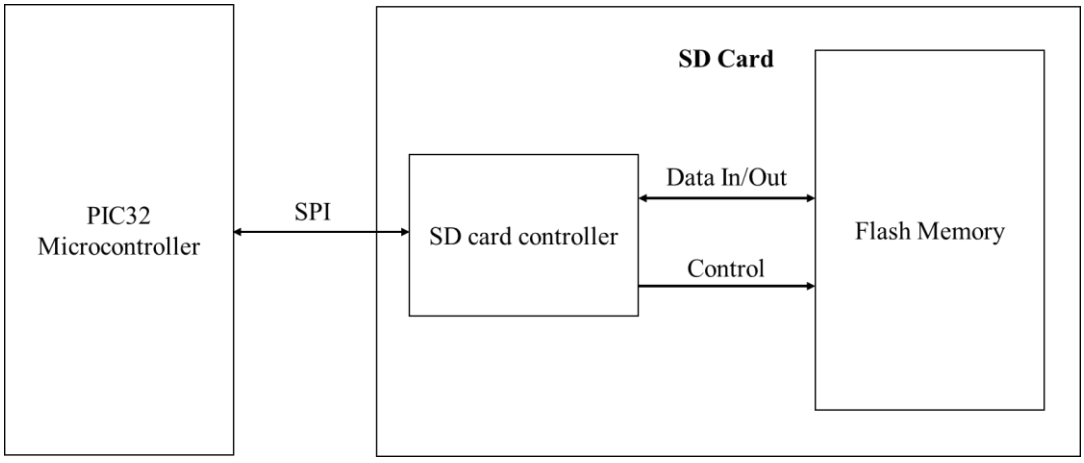


Figure 3. Block diagram of the system

As the figure listed above, to read or write data into the SD card. Our team divide the project into three sections. They are SPI section, SD command section and FAT32 file system section.

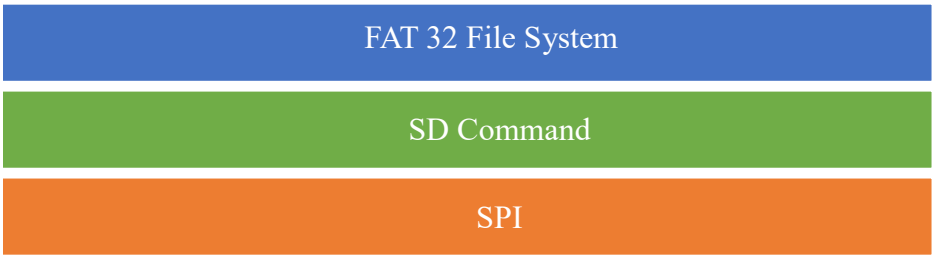


Figure 3. Three sections of the system

I. The PIC32 microcontroller needs to communicate with SD card controller using SPI buses. The data transmitted and received via SPI can be written and read through *SPI1BUF* from PIC32.

II. The internal SD card controller can decode the commands transmitted using SPI. Those commands are called standard SD command which can read the registers of the SD card, and also read/write the 'Memory Core'.

III. A FAT32 file system is mapped into the flash memory. This enables the user to directly access or modify the files. With FAT32 file system, it will be very useful that the files can be read directly not only from PIC32 microcontroller but in windows and other operating systems.

3.1 SPI section

The pin out for SD card and PIC32 for the SPI interfacing mode is shown in the following figures.

Physical pin Number on PIC32	Name	Description
6 (RB2)	CS	Chip select (active low)
24 (RB13)	MOSI(SDO1)	Master out slave in

27 (GND)	GND	Ground
25 (RB14)	SCK1	Clock
22 (RB11)	MISO(SDI1)	Master in slave out
3 (RA1)	U2RX	UART receive
21 (RB10)	U2TX	UART transmit

Table 1. Pinout for PIC32

Pin Number on SD card	Name	Description
1	CS	Chip select (active low)
2	MOSI(DataIn)	Master out slave in
3	V_{SS1}	Ground
4	V_{DD}	Voltage supply
5	CLK	Clock
6	V_{SS2}	Ground
7	MISO(DataOut)	Master in slave out
8	Reserved	Reserved for SPI mode
9	Reserved	Reserved for SPI mode

Table 2. Pinout for SD card

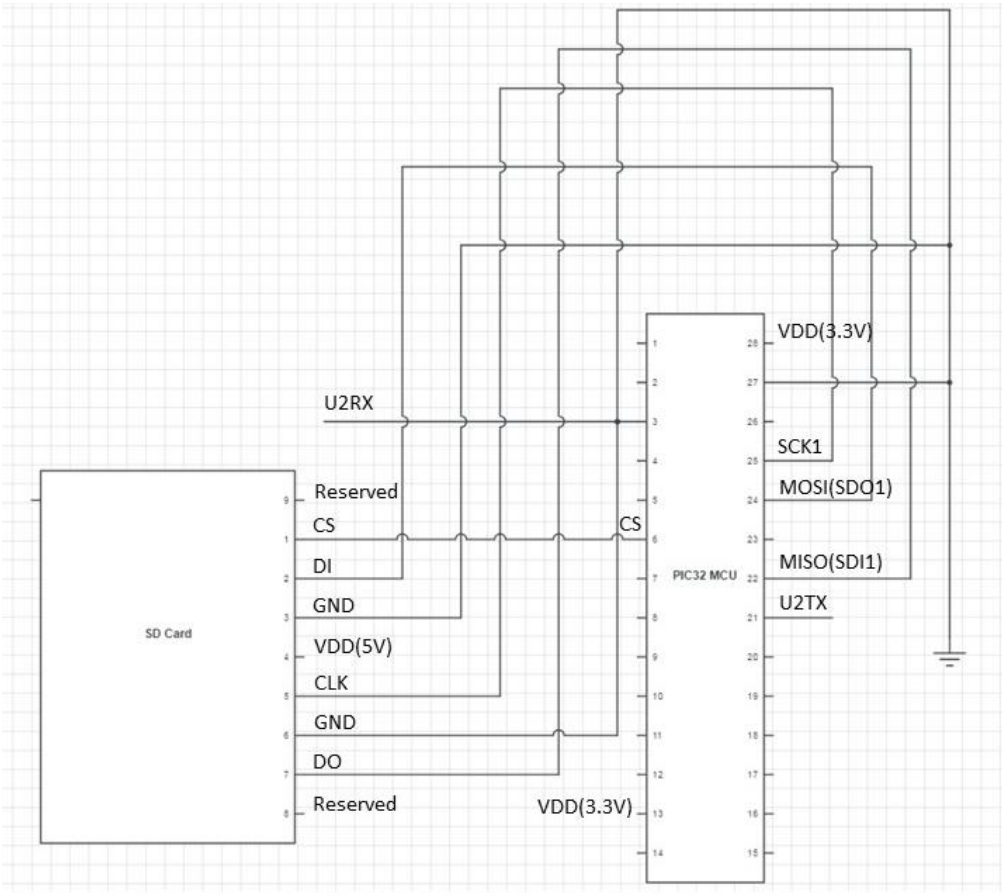


Figure 4. Connection for PIC32 and SD card

After setting up the circuit, the SPI communication needs to be initialized. The following code is used to set up SPI. It is mandatory to set up the SPI with a lower clock rate, since the initialization of SD card only allow low clock frequency. After initialization, the SPI clock frequency is speeded up as 20MHz to

```
volatile SpiChannel spiChn = SPI_CHANNEL1; // the SPI channel to use
    volatile int spiClkDiv = 160; // 250k Hz speed for SD initialization
    SpiChnOpen(spiChn, SPI_OPEN_ON | SPI_OPEN_MODE8 | SPI_OPEN_MSTEN ,
spiClkDiv | SPI_OPEN_SMP_END);
```

After initialization, our group is able to transmit and receive data by reading/writing to the register called *SPI1BUF* in PIC32. Therefore, two functions are generated with the purpose

which are called *SPI_transmit* and *SPI_receive*.

Basic functions in SPI section

Fuction name	Description
SPI_transmit(unsigned char data)	Transmit the 8 bits data to the spi buffer
SPI_receive(unsigned char data)	Get the 8 bits data from the spi buffer

Table 2. Basic functions for SPI communication

3.2 SD command section

All the SD commands supported in the SPI mode are 6 bytes long. The MSB is transmitted first and the actual command occupies the first byte. The command byte is followed by its 4 bytes long arguments. The last byte is the CRC byte respective of the command and the argument bytes.

When the host sends a command to the SD card, the SD card will first send a corresponding respond to the host, if the command is not wrong SD card will be followed by the implementation of the host command.

The structure of a command block in the SPI interface mode of a SD card is shown in the following figure.

Byte 1			Bytes 2—5	Byte 6	
7	6	5 0	31 0	7	0
0	1	Command	Command Argument	CRC	1

Figure 5. Structure of a command block

Below is a list of the basic commands our team uses in the project.

```
#define GO_IDLE_STATE 0
#define SEND_OP_COND 1
#define SEND_IF_COND 8
#define SEND_CSD 9
#define STOP_TRANSMISSION 12
#define SEND_STATUS 13
#define SET_BLOCK_LEN 16
#define READ_SINGLE_BLOCK 17
#define READ_MULTIPLE_BLOCKS 18
```

```
#define WRITE_MULTIPLE_BLOCKS 25
#define ERASE_BLOCK_START_ADDR 32
#define ERASE_BLOCK_END_ADDR 33
#define ERASE_SELECTED_BLOCKS 38
#define SD_SEND_OP_COND 41
#define APP_CMD 55
#define READ_OCR 58
#define CRC_ON_OFF 59
#define WRITE_SINGLE_BLOCK 24
```

SD card default read and write mode is SD mode. To use the SPI mode, our team need to write CMD0 and CMD1 command to SD controller. After the two commands are written successfully, we can use SPI mode, which can be easily used for microcontroller to read and write operations. Our team follows listed below steps for SD initialization.

3.2.1 SD send command

I. We first send 0xff synchronous clock cycles. (any number above 74 in decimal is preferred)

II. Send the CMD0 command to the SD card (since the highest order of the command number is always 0 and the second bit is 1, the command sent to the SD card is the result of 0 or 0x40 operation). The first, third, fifth, and fifth bytes of the command word are 0x00. The sixth byte of the command word is the CRC check byte, fixed to 0x95.

III. Checking the response of CMD58, then we can verify whether the SD card is standard of SDHC card.

IV. If (0x00 && cmd == 58) is true, we send 8 extra clock cycles, and then desert the chip select.

3.2.2 SD initialization

I. First send the instruction number CMD1 (0x01 | 0x40 = 0x41), and then send four 0x00 bytes, and finally send the CRC check code, here 0xFF.

II. Since SD card has been working in SPI mode, SD card does not default to CRC, so we write a 0xFF byte to fill the entire command word.

III. When the CMD1 instruction is sent to the SD card, we send 8 clock cycles until the SD card gives a response byte 0x00.

IV. After receiving the response byte of the SD card, the CS line is pulled high and then send 8 extra clock cycles.

3.2.3 SD Read single block

I. Send SD read command CMD17 (0x11 | 0x40 = 0x51).

II. Write four address parameters, 4 bytes into a 32-bit address value, the first byte is 32-bit address value of the highest 8-bit data, the first four bytes is the lowest 32-bit value 8-bit data.

III. Write CRC check bit 0xFF.

IV. Write a number of 0xFF empty operations.

V. Check SD card 0x00 response.

VI. Write a number of 0xFF empty operations.

VII. SD card sends 0x FE data header.

VIII. The SD card sends a 512-byte data block with the specified address.

IX. Since the SPI mode does not require the default CRC check, so the two bytes of data can be discarded.

X. Pull CS high, send 8 empty clock cycles.

3.2.4 SD write single block

I. Send SD write command CMD24 (0x18 | 0x40 = 0x58).

II. Write four address parameters, 4 bytes into a 32-bit address value, the first byte is the lowest 8-bit address 8-bit data, the fourth byte is the highest 32-bit

address value 8-bit data.

III. Write CRC check bit 0xFF.

IV. Write a number of 0xFF empty operation.

V. Check SD 0x00 response.

VI. Write a number of 0xFF empty operations.

VII. Write 512 bytes of data blocks.

VIII. Write two bytes of 0xFF as the CRC bytes.

IX. SD card sends x00101B response.

X. The CS line pulled low if the SD card writes 512 bytes of data to the specified address get interrupted.

XI. Pull CS high, send 8 empty clock cycles.

Basic functions in SD command section

Fuction name	Description
char SD_init(void);	SD card initialization
SD_sendCommand(unsigned char cmd, unsigned long arg);	Send SD command to SD controller
SD_readSingleBlock(unsigned long startBlock);	Read data from a specific block
SD_writeSingleBlock(unsigned long startBlock);	Write data to a specific block

Table 3. Basic functions for SD command

3.3 FAT32 file system section

FAT file system is widely used in the windows operating system. FAT32 file system is employed to store the files in this project. With FAT32 file system, it will be very useful

that the files can be read directly not only from PIC32 microcontroller but in windows and other operating systems.

From our research and reading FAT file system manual, our group define the consecutive 8 bit memory locations into ‘Sectors’ and The consecutive Sectors are grouped to form ‘Clusters’ by regulation. Our team implement FAT32 file system inside the Memory Core in a particular defined format. There are certain defined Sectors at the beginning of the Memory Core which are then followed by Clusters. The format of a FAT32 file system is as shown below:

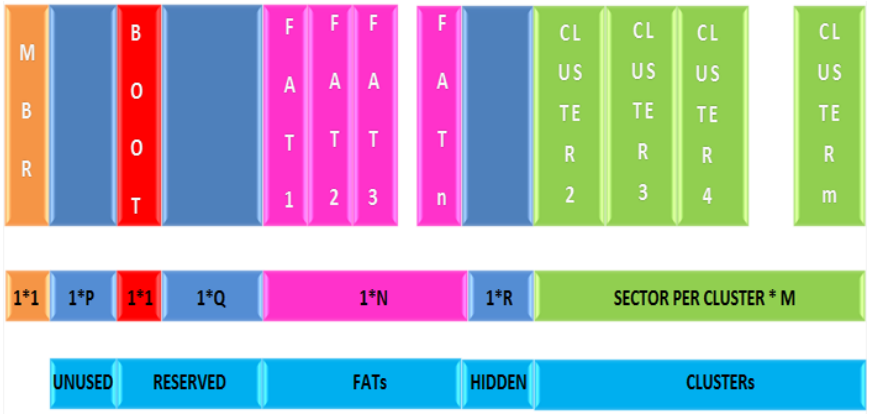


Figure 6. FAT32 format

The very first Sector is the MBR (Master Boot Record) which follows significant number of Unused Sectors. The Unused Sectors are followed by Reserved Sectors among which the first Sector is the BOOT Sector. The Reserved Sectors are followed by the FAT Sectors. The number of FAT Sectors depends upon the size of the file system. The FAT sectors are followed by few Hidden Sectors. The Hidden Sectors are followed by the Clusters. A File with a specific name can be read from the FAT32 formatted file system using the logic shown below; Take a closer look and it can be found that every process finally ends with a Sector read. This Sector read from the Memory Core of the SD card can be achieved by using the SD_readSingleBlock command from the SD Command section.

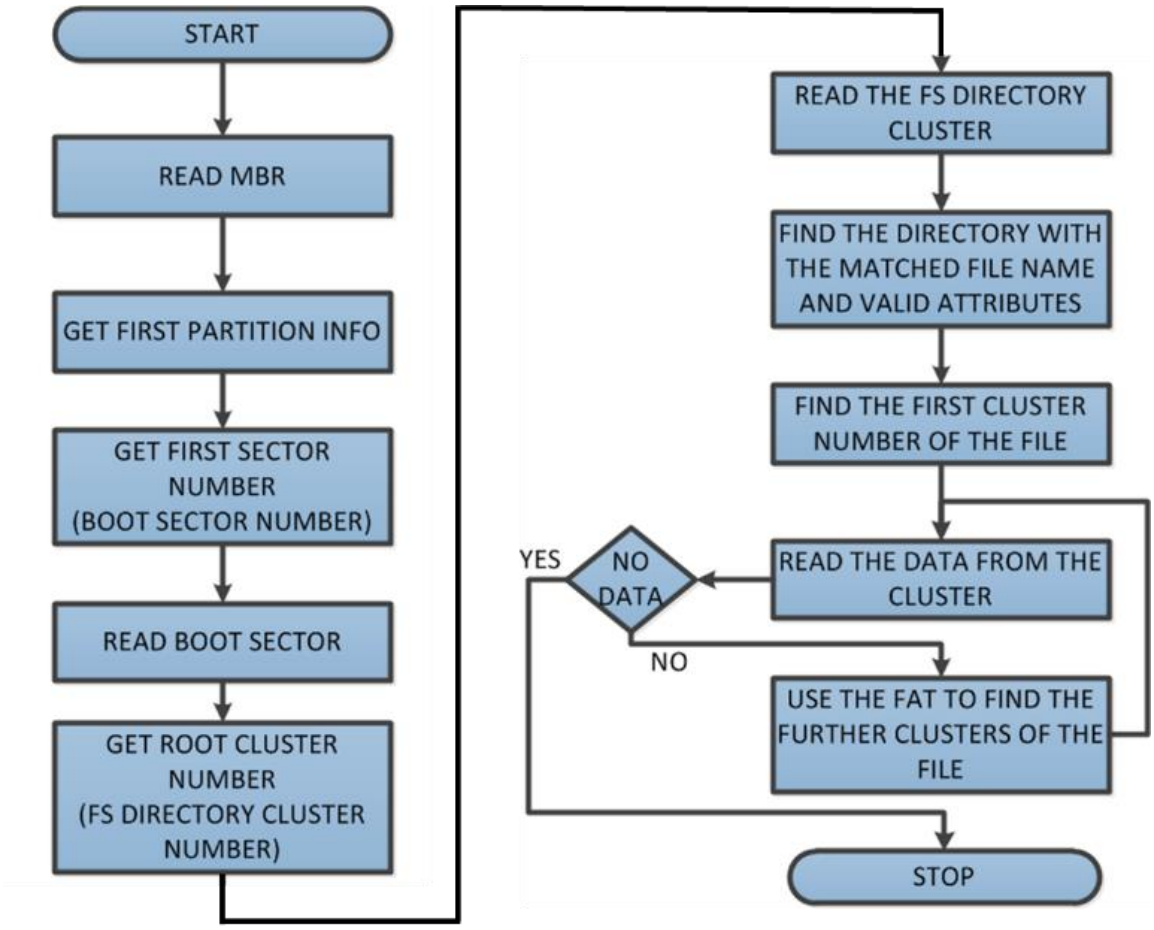


Figure 7. Logic workflow from FAT32

Basic functions in FAT32 file system section

Fuction name	Description
appendFile (void);	Write data to an exsisting file
memoryStatistics (void);	Get the memory usage of the SD card
writeFile (unsigned char *fileName);	Create a file in FAT32 format in the root directory if given file name does not exist; if the file already exists then append the data

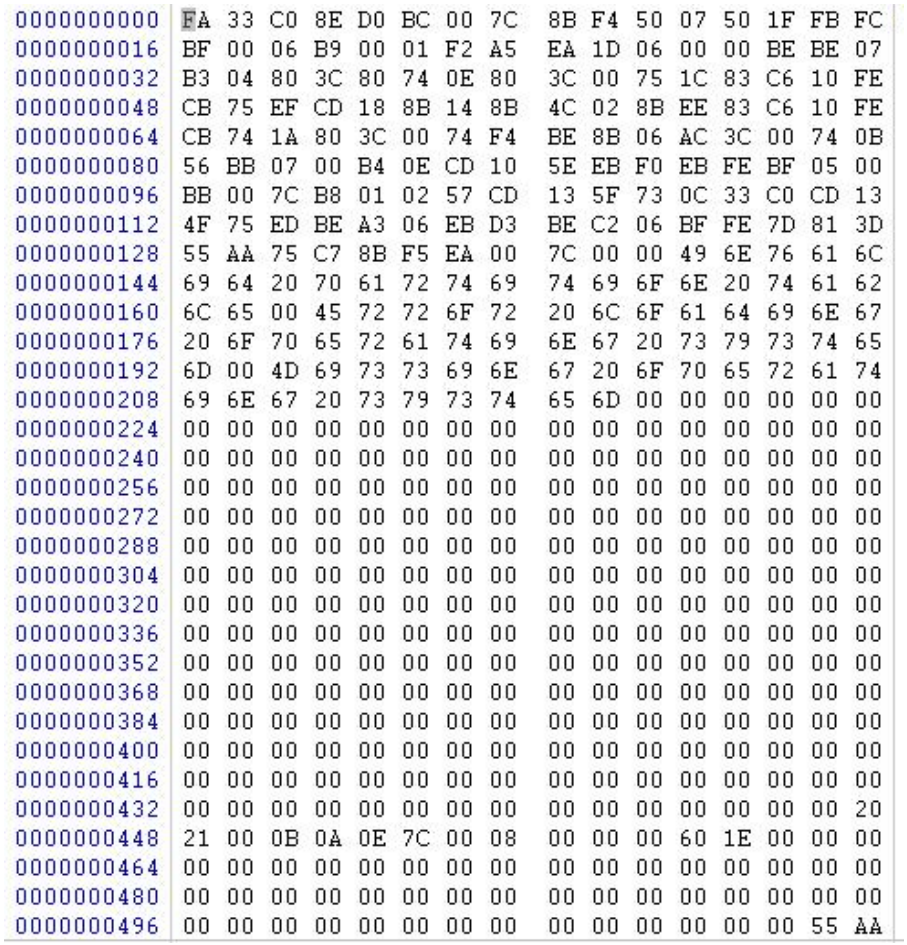
deleteFile (unsigned char *fileName);	Delete the file
findFiles (unsigned char flag, unsigned char *fileName);	Print file/dir list of the root directory, if flag = GET_LIST Delete the file, if flag = DELETE
readFile (unsigned char flag, unsigned char *fileName);	Read file from SD card if flag=READ; Verify whether a specified file is already existing if flag=VERIFY

Table 4. Basic functions for FAT32 file system

4. Testing and results

To test the accuracy and reliability of the SD card library two major tests are performed in the debugging stage.

I. Winhex is employed to read the information from the SD card on the personal computer. For instance, if we write data to a specific block on the SD card from UART of PIC32, the information can be checked using winhex. To evaluate the accuracy and debug during the design, our team verify the information of SD card library read and write functions by checking the block data using winhex. Following is figure when we read the detailed information from a specific block on the SD card using winhex.



0000000000	FA	33	C0	8E	D0	BC	00	7C	8B	F4	50	07	50	1F	FB	FC
0000000016	BF	00	06	B9	00	01	F2	A5	EA	1D	06	00	00	BE	BE	07
0000000032	B3	04	80	3C	80	74	0E	80	3C	00	75	1C	83	C6	10	FE
0000000048	CB	75	EF	CD	18	8B	14	8B	4C	02	8B	EE	83	C6	10	FE
0000000064	CB	74	1A	80	3C	00	74	F4	BE	8B	06	AC	3C	00	74	0B
0000000080	56	BB	07	00	B4	0E	CD	10	5E	EB	F0	EB	FE	BF	05	00
0000000096	BB	00	7C	B8	01	02	57	CD	13	5F	73	0C	33	C0	CD	13
0000000112	4F	75	ED	BE	A3	06	EB	D3	BE	C2	06	BF	FE	7D	81	3D
0000000128	55	AA	75	C7	8B	F5	EA	00	7C	00	00	49	6E	76	61	6C
0000000144	69	64	20	70	61	72	74	69	74	69	6F	6E	20	74	61	62
0000000160	6C	65	00	45	72	72	6F	72	20	6C	6F	61	64	69	6E	67
0000000176	20	6F	70	65	72	61	74	69	6E	67	20	73	79	73	74	65
0000000192	6D	00	4D	69	73	73	69	6E	67	20	6F	70	65	72	61	74
0000000208	69	6E	67	20	73	79	73	74	65	6D	00	00	00	00	00	00
0000000224	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000000240	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000000256	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000000272	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000000288	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000000304	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000000320	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000000336	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000000352	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000000368	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000000384	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000000400	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000000416	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000000432	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	20
0000000448	21	00	0B	0A	0E	7C	00	08	00	00	00	60	1E	00	00	00
0000000464	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000000480	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000000496	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA

Figure 8. SD data information read from Winhex

II. To evaluate the reliability of the SD card library, a testing user interface is designed to test important function in the library. The test interface can communicate from the computer to PIC32 using UART. In our test baud rate of 9600 is chosen from putty to transmit and receive data from PIC32. The user interface is shown as follows including read/ write function to a single block and read/ write data from a file in the file system.

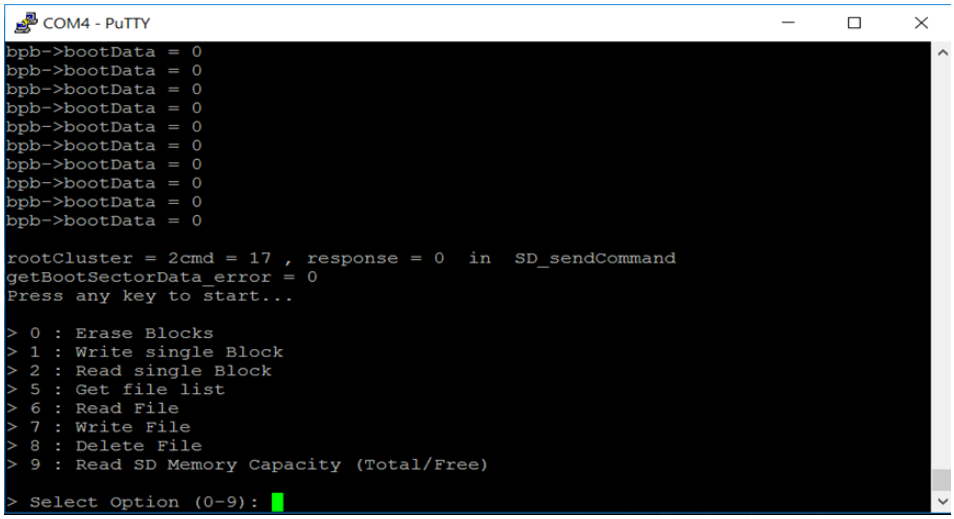


Figure 9. Test interface from Putty

As a result, all the functions are running properly without exceptions and all the information read from Winhex is identical as the information our team write using the library.

The speed of using SD_readSingleBlock and SD_writeSingleBlock functions are measured by running each function 1000 times. A timer is opened to measure the total running time. The function used to read or write from a single block has the data size of 512 Bytes. The result is shown as follows.

Function name	Total running time	Single block Write/Read time	Read/ Write rate(Bytes/s)
SD_readSingleBlock(unsigned long startBlock);	1225ms	1.225ms	417.96KB/s
SD_writeSingleBlock(unsigned long startBlock);	1343ms	1.343ms	381.23KB/s

Table 5. SD card speed performance

5. Conclusion

Overall, SD card library is reliable and is accurate enough to store large files using PIC32. During the design process, our team faced with various issues and bugs. One of the biggest issue was that when define the struct using MPLAB IDE, the compiler didn't allocate each variable in the struct within a consecutive memory location. To fix the problem, #pragma pack(1) is needed to be set, so that the compiler would compile the struct in a correct allocation. The measured speed of write or read data from PIC32 to SD card is roughly 400KB per second. This result is decent, since the 4 bit high speed SD protocol was not employed. Given 400KB/s read/ write speed from SPI mode, a speed of 2MB/s from SD mode could be estimated. This is identical as the rate provided from the datasheet driven by SPI clock frequency of 20MHz by PIC32. To summarize the project design, the functions our team designed could be easily implemented. With the reliability and decent transmit speed, the SD card library for PIC32 will enrich design alternatives for PIC users who need large space to store files.

6. Code Appendix

6.1 sd_routines.c

C:/Users/changliu/Desktop/lab3.X/sd_routines.c

```
#define clearPutty()    printf( "\x1b[2J")
#define homePutty()     printf( "\x1b[H")
#define PB_CLK          sys_clock
#include "config.h"

#include <stdlib.h>
#include <math.h>
#include "plib.h"
#include <xc.h>
#include "sd_routines.h"
#include "fat32.h"



void setupUART(void) {
    PPSInput (2, U2RX, RPA1); //Assign U2RX to pin RPA1 -- Physical pin 3 on 28 PDIPS
    PPSOutput(4, RPB10, U2TX); //Assign U2TX to pin RPB10 -- Physical pin 21 on 28 PDIP

    // white on adafruit 954 is RX in - so connect to pin 21
    // green on adafruit 954 is TX out - so connect to pin 3

    UARTConfigure(UART2, UART_ENABLE_PINS_TX_RX_ONLY);
    UARTSetLineControl(UART2, UART_DATA_SIZE_8_BITS | UART_PARITY_NONE | UART_STOP_BITS_1);
    UARTSetDataRate(UART2, PB_CLK, BAUDRATE);
    UARTEnable(UART2, UART_ENABLE_FLAGS(UART_PERIPHERAL | UART_RX | UART_TX));
}

//*****************************************************
//Function to receive a single byte
//*****************************************************
unsigned char receiveByte( void )
{
unsigned char data;

while(!DataRdyUART2()); // Wait for incomming data

data = ReadUART2();

return(data);
}


//*****************************************************
//Function to transmit a single byte
//*****************************************************
void transmitByte( unsigned char data )
{
do{while(!U2STAbits.TRMT); WriteUART2(data);}while(0);
}

//*****************************************************
//Function to transmit a string in RAM
//*****************************************************
void transmitString(unsigned char* string)
{
  while (*string)
```

```
   transmitByte(*string++);
}

//***************************************************
void transmitHex( unsigned char dataType, unsigned long data )
{
unsigned char count, i, temp;
unsigned char dataString[] = "0x        ";

if (dataType == 0) count = 2;
if (dataType == 1) count = 4;
if (dataType == 2) count = 8;

for(i=count; i>0; i--)
{
  temp = data % 16;
  if((temp>=0) && (temp<10)) dataString [i+1] = temp + 0x30;
  else dataString [i+1] = (temp - 10) + 0x41;

  data = data/16;
}

transmitString (dataString);
}

//***************************************************
//Function to transmit a string in Flash
//***************************************************
/*  SDO:    RB13    (TRIS defined in HardwareProfile.h and PPS defined in SD-SPI.c)
 *          physical pin 24
 *  SDI:    RB11    (TRIS defined in HardwareProfile.h and PPS defined in SD-SPI.c)
 *          physical pin 22
 *  SCK:    RB14    (TRIS defined in HardwareProfile.h)
 *          physical pin 25
 *  CS:     RB2     (defined in HardwareProfile.h)*
 *
 *  #define TRIS_SCK TRISBbits.TRISB14
 *  #define TRIS_SDI TRISBbits.TRISB11
 *  #define TRIS_SDO TRISBbits.TRISB13
 *  #define TRIS_CS  TRISBbits.TRISB2   */

void setupSPI() {
    volatile SpiChannel spiChn = SPI_CHANNEL1; // the SPI channel to use
    volatile int spiClkDiv = 160; // 250k Hz speed for SD initialization
    PPSOutput(3, RPB13, SDO1);    // RB13 as SDO1
    PPSInput(2, SDI1, RPB11);     // RB11 as SDI1

    mPORTBSetPinsDigitalOut(BIT_2); //RB2 as CS
    mPORTBSetBits(BIT_2);           //High for deselect
    SpiChnOpen(spiChn, SPI_OPEN_ON | SPI_OPEN_MODE8 | SPI_OPEN_MSTEN , spiClkDiv | SPI_OPEN_SMP_END);
}

unsigned char SPI_transmit(unsigned char data)
{
// Start transmission
```

```
  SPI1BUF = data;

// Wait for transmission complete
  while(!SPI1STATbits.SPITBE);
  data = SPI1BUF ;
  return data;
}

unsigned char SPI_receive(void)
{
  unsigned char data;
// Wait for reception complete
  SPI1BUF = 0xff;


  while(!SPI1STATbits.SPIRBF);
  data = SPI1BUF;
  return data;
}

//***
//**********************************************************************
unsigned char SD_init(void)
{
unsigned char i, response, SD_version;
unsigned int retry=0 ;

 for(i=0;i<10;i++)
      SPI_transmit(0xff);   //80 clock pulses spent before sending the first command

SD_CS_ASSERT;
//printf("sd_cs assert\n\r");
do
{

   response = SD_sendCommand(GO_IDLE_STATE, 0); //send 'reset & go idle' command
   retry++;
   if(retry>0x20){
      printf("time out, card not detected\n\r");
     return 1;   //time out, card not detected
   }

} while(response != 0x01);

SD_CS_DEASSERT;
//transmitString("sd deassert\n");

SPI_transmit (0xff);
SPI_transmit (0xff);

retry = 0;

SD_version = 2; //default set to SD compliance with ver2.x;
//this may change after checking the next command
do
```

```
{
response = SD_sendCommand(SEND_IF_COND,0x000001AA); //Check power supply status, mendatory for SDHC card
retry++;
if(retry>0xfe)
   {
  //TX_NEWLINE;
  SD_version = 1;
  cardType = 1;
  break;
   } //time out

}while(response != 0x01);

retry = 0;

do
{
response = SD_sendCommand(APP_CMD,0); //CMD55, must be sent before sending any ACMD command
response = SD_sendCommand(SD_SEND_OP_COND,0x40000000); //ACMD41

retry++;
if(retry>0xfe)
   {

  return 2;  //time out, card initialization failed
   }

}while(response != 0x00);


retry = 0;
SDHC_flag = 0;

if (SD_version == 2)
{
   do
   {
 response = SD_sendCommand(READ_OCR,0);
 retry++;
 if(retry>0xfe)
     {
       //TX_NEWLINE;
   cardType = 0;
   break;
     } //time out

   }while(response != 0x00);

   if(SDHC_flag == 1) cardType = 2;
   else cardType = 3;
}

printf("sd version = %d, cardType = %d\n\r",SD_version,cardType);
printf("\n");
//; //disable CRC; deafault - CRC disabled in SPI mode
```

```
//SD_sendCommand(SET_BLOCK_LEN, 512); //set block size to 512; default size is 512


return 0; //successful return
}


//*****************************************************************
unsigned char SD_sendCommand(unsigned char cmd, unsigned long arg)
{
unsigned char response, retry=0, status;

//SD card accepts byte address while SDHC accepts block address in multiples of 512
//so, if it's SD card we need to convert block address into corresponding byte address by
//multipying it with 512. which is equivalent to shifting it left 9 times
//following 'if' loop does that

if(SDHC_flag == 0)
{if(cmd == READ_SINGLE_BLOCK     ||
   cmd == READ_MULTIPLE_BLOCKS  ||
   cmd == WRITE_SINGLE_BLOCK    ||
   cmd == WRITE_MULTIPLE_BLOCKS ||
   cmd == ERASE_BLOCK_START_ADDR||
   cmd == ERASE_BLOCK_END_ADDR )
   {
     arg = arg << 9;
   }
}
SD_CS_ASSERT;

SPI_transmit(cmd | 0x40); //send command, first two bits always '01'
SPI_transmit(arg>>24);
SPI_transmit(arg>>16);
SPI_transmit(arg>>8);
SPI_transmit(arg);

if(cmd == SEND_IF_COND) //it is compulsory to send correct CRC for CMD8 (CRC=0x87) & CMD0 (CRC=0x95)
  SPI_transmit(0x87);    //for remaining commands, CRC is ignored in SPI mode
else
  SPI_transmit(0x95);

while((response = SPI_receive())== 0xff){ //wait response

    if(retry++ > 0xfe) break; //time out error
}
//printf("cmd = %d , response = %d  in  SD_sendCommand\n\r",cmd,response);

if(response == 0x00 && cmd == 58)  //checking response of CMD58
{
  status = SPI_receive() & 0x40;     //first byte of the OCR register (bit 31:24)
  if(status == 0x40) SDHC_flag = 1;  //we need it to verify SDHC card
  else SDHC_flag = 0;

  SPI_receive(); //remaining 3 bytes of the OCR register are ignored here
  SPI_receive(); //one can use these bytes to check power supply limits of SD
```

```
  SPI_receive();
}


SPI_receive(); //extra 8 CLK
SD_CS_DEASSERT;

return response; //return state
}


unsigned char SD_erase (unsigned long startBlock, unsigned long totalBlocks)
{
unsigned char response;

response = SD_sendCommand(ERASE_BLOCK_START_ADDR, startBlock); //send starting block address
if(response != 0x00) //check for SD status: 0x00 - OK (No flags set)
  return response;

response = SD_sendCommand(ERASE_BLOCK_END_ADDR,(startBlock + totalBlocks - 1)); //send end block address
if(response != 0x00)
  return response;

response = SD_sendCommand(ERASE_SELECTED_BLOCKS, 0); //erase all selected blocks
if(response != 0x00)
  return response;

return 0; //normal return
}


unsigned char SD_readSingleBlock(unsigned long startBlock)
{
unsigned char response;
unsigned int i, retry=0;

 response = SD_sendCommand(READ_SINGLE_BLOCK, startBlock); //read a Block command

 if(response != 0x00)
 {
     printf("no response in SD_readSingleBlock\n\r");
     return response; //check for SD status: 0x00 - OK (No flags set)
 }

SD_CS_ASSERT;

retry = 0;
while(SPI_receive() != 0xfe) //wait for start block token 0xfe (0x11111110)
  if(retry++ > 0xfffe){SD_CS_DEASSERT; return 1;} //return if time-out

for(i=0; i<512; i++) //read 512 bytes
  buffer[i] = SPI_receive();

SPI_receive(); //receive incoming CRC (16-bit), CRC is ignored here
```

```
SPI_receive();

SPI_receive(); //extra 8 clock pulses
SD_CS_DEASSERT;

return 0;
}


//*****************************************************************
unsigned char SD_writeSingleBlock(unsigned long startBlock)
{
unsigned char response;
unsigned int i, retry=0;

 response = SD_sendCommand(WRITE_SINGLE_BLOCK, startBlock); //write a Block command

 if(response != 0x00)
 {
  printf("no response in SD_writeSingleBlock\n\r");
  return response; //check for SD status: 0x00 - OK (No flags set)
 }
SD_CS_ASSERT;  // CS set to low

//SPI_transmit(0xff);
//SPI_transmit(0xff);
SPI_transmit(0xfe);     //Send start block token 0xfe (0x11111110)

for(i=0; i<512; i++)    //send 512 bytes data
  SPI_transmit(buffer[i]);

SPI_transmit(0xff);     //transmit dummy CRC (16-bit), CRC is ignored here
SPI_transmit(0xff);

response = SPI_receive();
response = SPI_receive();

if( (response & 0x1f) != 0x05) //response= 0xXXX0AAA1 ; AAA='010' - data accepted
{                              //AAA='101'-data rejected due to CRC error
  SD_CS_DEASSERT;              //AAA='110'-data rejected due to write error
 // printf("a xi ba");
  return response;
}

while(!SPI_receive()) //wait for SD card to complete writing and get idle
if(retry++ > 0xfffe){SD_CS_DEASSERT; return 1;}

SD_CS_DEASSERT;
SPI_transmit(0xff);   //just spend 8 clock cycle delay before reasserting the CS line
SD_CS_ASSERT;         //re-asserting the CS line to verify if card is still busy

while(!SPI_receive()) //wait for SD card to complete writing and get idle
   if(retry++ > 0xfffe){SD_CS_DEASSERT; return 1;}
SD_CS_DEASSERT;
```

```
return 0;
}
```

6.2 sd_routines.h

C:/Users/changliu/Desktop/lab3.X/sd_routines.h

```
#include "plib.h"
#include <xc.h>
//************************************************
// ***** HEADER FILE : SD_routines.h ******
//************************************************
#ifndef _SD_ROUTINES_H_
#define _SD_ROUTINES_H_

//Use following macro if you don't want to activate the multiple block access functions
//those functions are not required for FAT32

#define FAT_TESTING_ONLY

//use following macros if PB1 pin is used for Chip Select of SD
#define SD_CS_ASSERT     mPORTBClearBits(BIT_2)
#define SD_CS_DEASSERT   mPORTBSetBits(BIT_2)

//SD commands, many of these are not used here
#define GO_IDLE_STATE            0
#define SEND_OP_COND             1
#define SEND_IF_COND 8
#define SEND_CSD                 9
#define STOP_TRANSMISSION        12
#define SEND_STATUS              13
#define SET_BLOCK_LEN            16
#define READ_SINGLE_BLOCK        17
#define READ_MULTIPLE_BLOCKS     18
#define WRITE_SINGLE_BLOCK       24
#define WRITE_MULTIPLE_BLOCKS    25
#define ERASE_BLOCK_START_ADDR   32
#define ERASE_BLOCK_END_ADDR     33
#define ERASE_SELECTED_BLOCKS    38
#define SD_SEND_OP_COND 41
#define APP_CMD 55
#define READ_OCR 58
#define CRC_ON_OFF               59

#define ON     1
#define OFF    0

volatile unsigned long startBlock, totalBlocks;

volatile unsigned char SDHC_flag, cardType, buffer[512];

unsigned char SD_init(void);
unsigned char SD_sendCommand(unsigned char cmd, unsigned long arg);
unsigned char SD_readSingleBlock(unsigned long startBlock);
unsigned char SD_writeSingleBlock(unsigned long startBlock);
unsigned char SD_readMultipleBlock (unsigned long startBlock, unsigned long totalBlocks);
unsigned char SD_writeMultipleBlock(unsigned long startBlock, unsigned long totalBlocks);
unsigned char SD_erase (unsigned long startBlock, unsigned long totalBlocks);

#endif
```

1.1 of 1 2017.05.22 11:38:48

6.3 fat32.c

C:/Users/changliu/Desktop/lab3.X/fat32.c

```
#include "sd_routines.h"
#include "fat32.h"


//************************************************************************

//************************************************************************
unsigned char getBootSectorData (void)
{
struct BS_Structure *bpb; //mapping the buffer onto the structure
struct MBRinfo_Structure *mbr;
struct partitionInfo_Structure *partition;
unsigned long dataSectors;
int k;
unusedSectors = 0;

SD_readSingleBlock(0);

bpb = (struct BS_Structure *)buffer;

if(bpb->jumpBoot[0]!=0xE9 && bpb->jumpBoot[0]!=0xEB)   //check if it is boot sector
{

  mbr = (struct MBRinfo_Structure *) buffer;       //if it is not boot sector, it must be MBR   mbr =512

  if(mbr->signature != 0xaa55) return 1;       //if it is not even MBR then it's not FAT32

    unsigned char mama[16];
    for (k=0;k<16;k++) {mama[k] = mbr->partitionData[k];}

  partition = (struct partitionInfo_Structure *) mama;//first partition

 // partition = (struct partitionInfo_Structure *)(mbr->partitionData);

  printf("\n");
//  printf("firstSector = %d",partition->firstSector);printf("\n");  //first sectot = 8192
  unusedSectors = partition->firstSector; //the unused sectors, hidden to the FAT

  SD_readSingleBlock(partition->firstSector);//read the bpb sector
   bpb = (struct BS_Structure *) buffer;


  if(bpb->jumpBoot[0]!=0xE9 && bpb->jumpBoot[0]!=0xEB) return 1;
}

bytesPerSector = bpb->bytesPerSector;
//bytesPerSector = 512;

//transmitHex(INT, bytesPerSector); transmitByte(' ');
sectorPerCluster = bpb->sectorPerCluster;
//transmitHex(INT, sectorPerCluster); transmitByte(' ');
reservedSectorCount = bpb->reservedSectorCount;
rootCluster = bpb->rootCluster;// + (sector / sectorPerCluster) +1;

printf("rootCluster = %d",rootCluster);
firstDataSector = bpb->hiddenSectors + reservedSectorCount + (bpb->numberofFATs * bpb->FATsize_F32);
```

1.1 of 28 2017.05.22 10:31:32

```
dataSectors = bpb->totalSectors_F32
              - bpb->reservedSectorCount
              - ( bpb->numberofFATs * bpb->FATsize_F32);
totalClusters = dataSectors / sectorPerCluster;
//transmitHex(LONG, totalClusters); transmitByte(' ');

if((getSetFreeCluster (TOTAL_FREE, GET, 0)) > totalClusters)  //check if FSinfo free clusters count is v
     freeClusterCountUpdated = 0;
else
 freeClusterCountUpdated = 1;
return 0;
}


//*****************************************************************************
unsigned long getFirstSector(unsigned long clusterNumber)
{
  return (((clusterNumber - 2) * sectorPerCluster) + firstDataSector);
}


//*****************************************************************************
unsigned long getSetNextCluster (unsigned long clusterNumber,
                                 unsigned char get_set,
                                 unsigned long clusterEntry)
{
unsigned short FATEntryOffset;
unsigned long *FATEntryValue;
unsigned long FATEntrySector;
unsigned char retry = 0;

//get sector number of the cluster entry in the FAT
FATEntrySector = unusedSectors + reservedSectorCount + ((clusterNumber * 4) / bytesPerSector) ;

//get the offset address in that sector number
FATEntryOffset = (unsigned short) ((clusterNumber * 4) % bytesPerSector);

//read the sector into a buffer
while(retry <10)
{ if(!SD_readSingleBlock(FATEntrySector)) break; retry++;}

//get the cluster address from the buffer
FATEntryValue = (unsigned long *) &buffer[FATEntryOffset];

if(get_set == GET)
  return ((*FATEntryValue) & 0x0fffffff);


*FATEntryValue = clusterEntry;   //for setting new value in cluster entry in FAT

SD_writeSingleBlock(FATEntrySector);

return (0);
}
```

C:/Users/changliu/Desktop/lab3.X/fat32.c

```
//*****************************************************************************************
unsigned long getSetFreeCluster(unsigned char totOrNext, unsigned char get_set, unsigned long FSEntry)
{
struct FSInfo_Structure *FS = (struct FSInfo_Structure *) &buffer;
unsigned char error;

SD_readSingleBlock(unusedSectors + 1);

if((FS->leadSignature != 0x41615252) || (FS->structureSignature != 0x61417272) || (FS->trailSignature !=
  return 0xffffffff;


 if(get_set == GET)
 {
   if(totOrNext == TOTAL_FREE)
      return(FS->freeClusterCount);
   else // when totOrNext = NEXT_FREE
      return(FS->nextFreeCluster);
 }
 else
 {
   if(totOrNext == TOTAL_FREE)
      FS->freeClusterCount = FSEntry;
   else // when totOrNext = NEXT_FREE
  FS->nextFreeCluster = FSEntry;

   error = SD_writeSingleBlock(unusedSectors + 1);//update FSinfo
 }
 return 0xffffffff;
}



//*************************************************************************
struct dir_Structure* findFiles (unsigned char flag, unsigned char *fileName)
{
unsigned long cluster, sector, firstSector, firstCluster, nextCluster;
struct dir_Structure *dir;
unsigned short i;
unsigned char j;

cluster = rootCluster; //root cluster
            int k;
while(1)
{
//            printf("\n");
//
//            printf("cluster = %d", cluster);
//            printf("\n");
   firstSector = getFirstSector (cluster);

   for(sector = 0; sector < sectorPerCluster; sector++)
   {
```

```
           printf("\n");


    SD_readSingleBlock (firstSector + sector);



    for(i=0; i<bytesPerSector; i+=32)
    {
   dir = (struct dir_Structure *) &buffer[i];


       if(dir->name[0] == EMPTY) //indicates end of the file list of the directory
{
  if((flag == GET_FILE) || (flag == DELETE))
     printf( "File does not exist!");
  return 0;
}
if((dir->name[0] != DELETED) && (dir->attrib != ATTR_LONG_NAME))
        {
          if((flag == GET_FILE) || (flag == DELETE))
          {
            for(j=0; j<11; j++)
            if(dir->name[j] != fileName[j]) break;
            if(j == 11)
{
  if(flag == GET_FILE)
             {
    appendFileSector = firstSector + sector;
appendFileLocation = i;
appendStartCluster = (((unsigned long) dir->firstClusterHI) << 16) | dir->firstClusterLO;
fileSize = dir->fileSize;
    return (dir);
  }
  else    //when flag = DELETE
  {

 printf( "Deleting..");


 firstCluster = (((unsigned long) dir->firstClusterHI) << 16) | dir->firstClusterLO;

 //mark file as 'deleted' in FAT table
 dir->name[0] = DELETED;
 SD_writeSingleBlock (firstSector+sector);

 freeMemoryUpdate (ADD, dir->fileSize);

 //update next free cluster entry in FSinfo sector
 cluster = getSetFreeCluster (NEXT_FREE, GET, 0);
 if(firstCluster < cluster)
     getSetFreeCluster (NEXT_FREE, SET, firstCluster);

 //mark all the clusters allocated to the file as 'free'
    while(1)
```

```
    {
       nextCluster = getSetNextCluster (firstCluster, GET, 0);
getSetNextCluster (firstCluster, SET, 0);
if(nextCluster > 0x0ffffff6)
   {printf( "File deleted!");return 0;}
firstCluster = nextCluster;
   }
  }
           }
         }
         else  //when flag = GET_LIST
  {

 for(j=0; j<11; j++)
     {
   if(j == 8) transmitByte(' ');
   transmitByte (dir->name[j]);
 }
     printf ( "   ");
     if((dir->attrib != 0x10) && (dir->attrib != 0x08))
 {
     printf ( "FILE" );
         printf ( "   ");
     displayMemory (LOW, dir->fileSize);
                printf("\n");
 }
 else
   printf ((dir->attrib == 0x10)?  "DIR" :  "ROOT");
  }
      }
     }
   }

   cluster = (getSetNextCluster (cluster, GET, 0));

   if(cluster > 0x0ffffff6)
    return 0;
   if(cluster == 0)
   {printf( "Error in getting cluster");  return 0;}
 }
return 0;
}



//***************************************************************************
unsigned char readFile (unsigned char flag, unsigned char *fileName)
{
struct dir_Structure *dir;
unsigned long cluster, byteCounter = 0, fileSize, firstSector;
unsigned short k;
unsigned char j, error;

error = convertFileName (fileName); //convert fileName into FAT format
if(error) return 2;
```

```
dir = findFiles (GET_FILE, fileName); //get the file location
if(dir == 0)
  return (0);

if(flag == VERIFY) return (1);//specified file name is already existing

cluster = (((unsigned long) dir->firstClusterHI) << 16) | dir->firstClusterLO;

fileSize = dir->fileSize;




while(1)
{
  firstSector = getFirstSector (cluster);

  for(j=0; j<sectorPerCluster; j++)
  {
    SD_readSingleBlock(firstSector + j);

for(k=0; k<512; k++)
    {
      transmitByte(buffer[k]);
      if ((byteCounter++) >= fileSize ) return 0;
    }
  }
  cluster = getSetNextCluster (cluster, GET, 0);
  if(cluster == 0) {printf( "Error in getting cluster"); return 0;}
}
return 0;
}


//**************************************************************************
unsigned char convertFileName (unsigned char *fileName)
{
unsigned char fileNameFAT[11];
unsigned char j, k;

for(j=0; j<12; j++)
if(fileName[j] == '.') break;

if(j>8) {printf( "Invalid fileName.."); return 1;}

for(k=0; k<j; k++) //setting file name
  fileNameFAT[k] = fileName[k];

for(k=j; k<=7; k++) //filling file name trail with blanks
  fileNameFAT[k] = ' ';

j++;
for(k=8; k<11; k++) //setting file extention
```

```
{
  if(fileName[j] != 0)
    fileNameFAT[k] = fileName[j++];
  else //filling extension trail with blanks
    while(k<11)
      fileNameFAT[k++] = ' ';
}

for(j=0; j<11; j++) //converting small letters to caps
  if((fileNameFAT[j] >= 0x61) && (fileNameFAT[j] <= 0x7a))
    fileNameFAT[j] -= 0x20;

for(j=0; j<11; j++)
  fileName[j] = fileNameFAT[j];

return 0;
}


//******************************************************************************
void writeFile (unsigned char *fileName)
{
unsigned char j, data, error, fileCreatedFlag = 0, start = 0, appendFile = 0, sectorEndFlag = 0, sector;
unsigned short i, firstClusterHigh, firstClusterLow;
struct dir_Structure *dir;
unsigned long cluster, nextCluster, prevCluster, firstSector, clusterCount, extraMemory;

j = readFile (VERIFY, fileName);

if(j == 1)
{
  printf( "  File already existing, appending data..");
  appendFile = 1;
  cluster = appendStartCluster;
  clusterCount=0;
  while(1)
  {
    nextCluster = getSetNextCluster (cluster, GET, 0);
    if(nextCluster == EOF) break;
cluster = nextCluster;
clusterCount++;
  }

  sector = (fileSize - (clusterCount * sectorPerCluster * bytesPerSector)) / bytesPerSector; //last sect
  start = 1;
//  appendFile();
//  return;
}
else if(j == 2)
   return; //invalid file name
else
{

  printf( " Creating File..");
```

C:/Users/changliu/Desktop/lab3.X/fat32.c

```
  cluster = getSetFreeCluster (NEXT_FREE, GET, 0);
  if(cluster > totalClusters)
     cluster = rootCluster;

  cluster = searchNextFreeCluster(cluster);
   if(cluster == 0)
   {

      printf( " No free cluster!");
  return;
   }
  getSetNextCluster(cluster, SET, EOF);   //last cluster of the file, marked EOF

  firstClusterHigh = (unsigned short) ((cluster & 0xffff0000) >> 16 );
  firstClusterLow = (unsigned short) ( cluster & 0x0000ffff);
  fileSize = 0;
}

while(1)
{
   if(start)
   {
      start = 0;
  startBlock = getFirstSector (cluster) + sector;
  SD_readSingleBlock (startBlock);
  i = fileSize % bytesPerSector;
  j = sector;

   }
   else
   {
      startBlock = getFirstSector (cluster);
  i=0;
  j=0;
   }



   printf( " Enter text (end with ~):");

   do
   {
     if(sectorEndFlag == 1) //special case when the last character in previous sector was '\r'
 {
 transmitByte ('\n');
       buffer[i++] = '\n'; //appending 'Line Feed (LF)' character
fileSize++;
 }

sectorEndFlag = 0;

 data = receiveByte();
 if(data == 0x08)//'Back Space' key pressed
 {
   if(i != 0)
   {
```

```
    transmitByte(data);
transmitByte(' ');
    transmitByte(data);
    i--;
fileSize--;
  }
  continue;
}
transmitByte(data);
    buffer[i++] = data;
fileSize++;
    if(data == '\r')  //'Carriege Return (CR)' character
    {
       if(i == 512)
  sectorEndFlag = 1;  //flag to indicate that the appended '\n' char should be put in the next sector
   else
{
  transmitByte ('\n');
         buffer[i++] = '\n'; //appending 'Line Feed (LF)' character
  fileSize++;
   }
    }

    if(i >= 512)   //though 'i' will never become greater than 512, it's kept here to avoid
{//infinite loop in case it happens to be greater than 512 due to some data corruption
  i=0;
  error = SD_writeSingleBlock (startBlock);
      j++;
  if(j == sectorPerCluster) {j = 0; break;}
  startBlock++;
    }
}while (data != '~');

  if(data == '~')
  {
     fileSize--;//to remove the last entered '~' character
 i--;
 for(;i<512;i++)  //fill the rest of the buffer with 0x00
       buffer[i]= 0x00;
    error = SD_writeSingleBlock (startBlock);

     break;
  }

  prevCluster = cluster;

  cluster = searchNextFreeCluster(prevCluster); //look for a free cluster starting from the current clu

  if(cluster == 0)
  {

     printf( " No free cluster!");
 return;
  }
```

C:/Users/changliu/Desktop/lab3.X/fat32.c

```
   getSetNextCluster(prevCluster, SET, cluster);
   getSetNextCluster(cluster, SET, EOF);   //last cluster of the file, marked EOF
}

getSetFreeCluster (NEXT_FREE, SET, cluster); //update FSinfo next free cluster entry

if(appendFile)  //executes this loop if file is to be appended
{
  SD_readSingleBlock (appendFileSector);
  dir = (struct dir_Structure *) &buffer[appendFileLocation];
  extraMemory = fileSize - dir->fileSize;
  dir->fileSize = fileSize;
  SD_writeSingleBlock (appendFileSector);
  freeMemoryUpdate (REMOVE, extraMemory); //updating free memory count in FSinfo sector;



  printf( " File appended!");

  return;
}

//executes following portion when new file is created

prevCluster = rootCluster; //root cluster

while(1)
{
   firstSector = getFirstSector (prevCluster);

   for(sector = 0; sector < sectorPerCluster; sector++)
   {
     SD_readSingleBlock (firstSector + sector);


     for(i=0; i<bytesPerSector; i+=32)
     {
    dir = (struct dir_Structure *) &buffer[i];

if(fileCreatedFlag)   //to mark last directory entry with 0x00 (empty) mark
 {   //indicating end of the directory file list
   dir->name[0] = 0x00;
           return;
         }

        if((dir->name[0] == EMPTY) || (dir->name[0] == DELETED))  //looking for an empty slot to enter f
{
  for(j=0; j<11; j++)
  dir->name[j] = fileName[j];
  dir->attrib = ATTR_ARCHIVE;//settting file attribute as 'archive'
  dir->NTreserved = 0;//always set to 0
  dir->timeTenth = 0;//always set to 0
  dir->createTime = 0x9684;//fixed time of creation
  dir->createDate = 0x3a37;//fixed date of creation
```

C:/Users/changliu/Desktop/lab3.X/fat32.c

```
  dir->lastAccessDate = 0x3a37;//fixed date of last access
  dir->writeTime = 0x9684;//fixed time of last write
  dir->writeDate = 0x3a37;//fixed date of last write
  dir->firstClusterHI = firstClusterHigh;
  dir->firstClusterLO = firstClusterLow;
  dir->fileSize = fileSize;

  SD_writeSingleBlock (firstSector + sector);
  fileCreatedFlag = 1;



  printf( " File Created!");

  freeMemoryUpdate (REMOVE, fileSize); //updating free memory count in FSinfo sector

        }
     }
   }

   cluster = getSetNextCluster (prevCluster, GET, 0);

   if(cluster > 0x0ffffff6)
   {
      if(cluster == EOF)   //this situation will come when total files in root is multiple of (32*sector
  {
cluster = searchNextFreeCluster(prevCluster); //find next cluster for root directory entries
getSetNextCluster(prevCluster, SET, cluster); //link the new cluster of root to the previous cluster
getSetNextCluster(cluster, SET, EOF);  //set the new cluster as end of the root directory
      }

      else
      {
    printf( "End of Cluster Chain");
    return;
      }
   }
   if(cluster == 0) {printf( "Error in getting cluster"); return;}

   prevCluster = cluster;
 }

 return;
}

//*****************************************************************
unsigned long searchNextFreeCluster (unsigned long startCluster)
{
  unsigned long cluster, *value, sector;
  unsigned char i;

startCluster -=  (startCluster % 128);   //to start with the first file in a FAT sector
    for(cluster =startCluster; cluster <totalClusters; cluster+=128)
    {
      sector = unusedSectors + reservedSectorCount + ((cluster * 4) / bytesPerSector);
```

```
      SD_readSingleBlock(sector);
      for(i=0; i<128; i++)
      {
        value = (unsigned long *) &buffer[i*4];
         if(((*value) & 0x0fffffff) == 0)
            return(cluster+i);
      }
    }

 return 0;
}



//***************************************************************************
void memoryStatistics (void)
{
unsigned long freeClusters, totalClusterCount, cluster;
unsigned long totalMemory, freeMemory;
unsigned long sector, *value;
unsigned short i;


totalMemory = totalClusters * sectorPerCluster / 1024;
totalMemory *= bytesPerSector;



printf( "Total Memory: ");

displayMemory (HIGH, totalMemory);

freeClusters = getSetFreeCluster (TOTAL_FREE, GET, 0);
//freeClusters = 0xffffffff;

if(freeClusters > totalClusters)
{
   freeClusterCountUpdated = 0;
   freeClusters = 0;
   totalClusterCount = 0;
   cluster = rootCluster;
    while(1)
    {
      sector = unusedSectors + reservedSectorCount + ((cluster * 4) / bytesPerSector) ;
      SD_readSingleBlock(sector);
      for(i=0; i<128; i++)
      {
           value = (unsigned long *) &buffer[i*4];
         if(((*value)& 0x0fffffff) == 0)
            freeClusters++;;

         totalClusterCount++;
         if(totalClusterCount == (totalClusters+2)) break;
      }
      if(i < 128) break;
```

```
      cluster+=128;
    }
}

if(!freeClusterCountUpdated)
  getSetFreeCluster (TOTAL_FREE, SET, freeClusters); //update FSinfo next free cluster entry
freeClusterCountUpdated = 1;  //set flag
freeMemory = freeClusters * sectorPerCluster / 1024;
freeMemory *= bytesPerSector ;

printf( " Free Memory: ");
displayMemory (HIGH, freeMemory);

}



//*************************************************************
void displayMemory (unsigned char flag, unsigned long memory)
{
  unsigned char memoryString[] = "              Bytes"; //19 character long string for memory display
  unsigned char i;
  for(i=12; i>0; i--) //converting freeMemory into ASCII string
  {
    if(i==5 || i==9)
{
   memoryString[i-1] = ',';
   i--;
}
    memoryString[i-1] = (memory % 10) | 0x30;
    memory /= 10;
if(memory == 0) break;
  }
  if(flag == HIGH)  memoryString[13] = 'K';
  transmitString(memoryString);
}


//*********************************************************************
void deleteFile (unsigned char *fileName)
{
  unsigned char error;

  error = convertFileName (fileName);
  if(error) return;

  findFiles (DELETE, fileName);
}


//*********************************************************************
void freeMemoryUpdate (unsigned char flag, unsigned long size)
{
  unsigned long freeClusters;
  //convert file size into number of clusters occupied
```

C:/Users/changliu/Desktop/lab3.X/fat32.c

```
  if((size % 512) == 0) size = size / 512;
  else size = (size / 512) +1;
  if((size % 8) == 0) size = size / 8;
  else size = (size / 8) +1;

  if(freeClusterCountUpdated)
  {
freeClusters = getSetFreeCluster (TOTAL_FREE, GET, 0);
if(flag == ADD)
     freeClusters = freeClusters + size;
else  //when flag = REMOVE
   freeClusters = freeClusters - size;
getSetFreeCluster (TOTAL_FREE, SET, freeClusters);
  }
}
```

14.1 of 28 2017.05.22 10:31:32

6.4 fat32.h

C:/Users/changliu/Desktop/lab3.X/fat32.h

```
//**************************************************
// ***** HEADER FILE : FAT32.h ******
//**************************************************
#ifndef _FAT32_H_
#define _FAT32_H_


#include <stdlib.h>
#include <math.h>

#include "plib.h"
#include <xc.h>
////////////////////////////////////
#include "sd_routines.h"

#pragma pack(1)
//Structure to access Master Boot Record for getting info about partioions
struct MBRinfo_Structure{
unsigned charnothing[446];//ignore, placed here to fill the gap in the structure
unsigned charpartitionData[64];//partition records (16x4)
unsigned shortsignature;//0xaa55
};

//Structure to access info of the first partioion of the disk   16 bytes
struct partitionInfo_Structure{
unsigned charstatus;//0x80 - active partition
unsigned char headStart;//starting head
unsigned shortcylSectStart;//starting cylinder and sector
unsigned chartype;//partition type
unsigned charheadEnd;//ending head of the partition
unsigned shortcylSectEnd;//ending cylinder and sector
unsigned longfirstSector;//total sectors between MBR & the first sector of the partition
unsigned longsectorsTotal;//size of this partition in sectors
//unsigned char   notused_data[48];
};

//Structure to access boot sector data 512 bytes
struct BS_Structure{
unsigned char jumpBoot[3]; //default: 0x009000EB

unsigned char OEMName[8];
unsigned short bytesPerSector; //deafault: 512
unsigned char sectorPerCluster;
unsigned short reservedSectorCount;
unsigned char numberofFATs;
unsigned short rootEntryCount;
unsigned short totalSectors_F16; //must be 0 for FAT32
unsigned char mediaType;
unsigned short FATsize_F16; //must be 0 for FAT32
unsigned short sectorsPerTrack;
unsigned short numberofHeads;
unsigned long hiddenSectors;
unsigned long totalSectors_F32;
unsigned long FATsize_F32; //count of sectors occupied by one FAT
```

C:/Users/changliu/Desktop/lab3.X/fat32.h

```
unsigned short extFlags;
unsigned short FSversion; //0x0000 (defines version 0.0)
unsigned long rootCluster; //first cluster of root directory (=2)
unsigned short FSinfo; //sector number of FSinfo structure (=1)
unsigned short BackupBootSector;
unsigned char reserved[12];
unsigned char driveNumber;
unsigned char reserved1;
unsigned char bootSignature;
unsigned long volumeID;
unsigned char volumeLabel[11]; //"NO NAME "
unsigned char fileSystemType[8]; //"FAT32"
unsigned char bootData[420];
unsigned short bootEndSignature; //0xaa55
}bpb;


//Structure to access FSinfo sector data
struct FSInfo_Structure
{
unsigned long leadSignature; //0x41615252
unsigned char reserved1[480];
unsigned long structureSignature; //0x61417272
unsigned long freeClusterCount; //initial: 0xffffffff
unsigned long nextFreeCluster; //initial: 0xffffffff
unsigned char reserved2[12];
unsigned long trailSignature; //0xaa550000
};

//Structure to access Directory Entry in the FAT
struct dir_Structure{
unsigned char name[11];
unsigned char attrib; //file attributes
unsigned char NTreserved; //always 0
unsigned char timeTenth; //tenths of seconds, set to 0 here
unsigned short createTime; //time file was created
unsigned short createDate; //date file was created
unsigned short lastAccessDate;
unsigned short firstClusterHI; //higher word of the first cluster number
unsigned short writeTime; //time of last write
unsigned short writeDate; //date of last write
unsigned short firstClusterLO; //lower word of the first cluster number
unsigned long fileSize; //size of file in bytes
};

//Attribute definitions for file/directory
#define ATTR_READ_ONLY     0x01
#define ATTR_HIDDEN        0x02
#define ATTR_SYSTEM        0x04
#define ATTR_VOLUME_ID     0x08
#define ATTR_DIRECTORY     0x10
#define ATTR_ARCHIVE       0x20
#define ATTR_LONG_NAME     0x0f
```

C:/Users/changliu/Desktop/lab3.X/fat32.h

```
#define DIR_ENTRY_SIZE     0x32
#define EMPTY              0x00
#define DELETED            0xe5
#define GET     0
#define SET     1
#define READ0
#define VERIFY  1
#define ADD0
#define REMOVE1
#define LOW0
#define HIGH1
#define TOTAL_FREE   1
#define NEXT_FREE    2
#define GET_LIST     0
#define GET_FILE     1
#define DELETE 2
#define EOF0x0fffffff


//************* external variables *************
volatile unsigned long firstDataSector, rootCluster, totalClusters;
volatile unsigned short  bytesPerSector, reservedSectorCount;
volatile unsigned char sectorPerCluster;
unsigned long unusedSectors, appendFileSector, appendFileLocation, fileSize, appendStartCluster;

//global flag to keep track of free cluster count updating in FSinfo sector
unsigned char freeClusterCountUpdated;



//************* functions *************
unsigned char getBootSectorData (void);
unsigned long getFirstSector(unsigned long clusterNumber);
unsigned long getSetFreeCluster(unsigned char totOrNext, unsigned char get_set, unsigned long FSEntry);
struct dir_Structure* findFiles (unsigned char flag, unsigned char *fileName);
unsigned long getSetNextCluster (unsigned long clusterNumber,unsigned char get_set,unsigned long cluster
unsigned char readFile (unsigned char flag, unsigned char *fileName);
unsigned char convertFileName (unsigned char *fileName);
void writeFile (unsigned char *fileName);
void appendFile (void);
unsigned long searchNextFreeCluster (unsigned long startCluster);
void memoryStatistics (void);
void displayMemory (unsigned char flag, unsigned long memory);
void deleteFile (unsigned char *fileName);
void freeMemoryUpdate (unsigned char flag, unsigned long size);

#endif
```

6.5 meng_test.c

C:/Users/changliu/Desktop/lab3.X/meng_test.c

```
void main(void){

    INTEnableSystemMultiVectoredInt();
    ANSELA = 0; ANSELB = 0;                 // Disable analog inputs
    CM1CON = 0; CM2CON = 0; CM3CON = 0;     // Disable analog comparators

    setupUART();
    homePutty();
    clearPutty();
    setupSPI();


//************************** MAIN *****************************//

unsigned char option, error, data, FAT32_active = 1;
unsigned int i;
unsigned char fileName[13];

printf ("*********************************************\n");
printf ("*********************************************\n");
printf ("     Meng project microSD Card Testing..     \n");
printf ("*********************************************\n");
printf ("*********************************************\n");
printf("\n");

cardType = 0;

for (i=0; i<10; i++)
{
  error = SD_init();
  if(!error) break;
}

if(error)
{
  if(error == 1) printf("SD card not detected..");
  if(error == 2) printf("Card Initialization failed..");

  while(1);  //wait here forever if error in SD init
}



    SpiChnClose(1);
   // SpiChnOpen(1, SPI_OPEN_ON | SPI_OPEN_MODE8 | SPI_OPEN_MSTEN , 4 | SPI_OPEN_SMP_END);// change the


    SpiChnOpen(1, SPI_OPEN_ON | SPI_OPEN_MODE8 | SPI_OPEN_MSTEN , 4 | SPI_OPEN_SMP_END);// change the SP

switch (cardType)
{
  case 1:printf("Standard Capacity Card (Ver 1.x) Detected!");
   break;
  case 2:printf("High Capacity Card Detected!");
   break;
  case 3:printf("Standard Capacity Card (Ver 2.x) Detected!");
```

1.1 of 12 2017.05.22 09:09:41

C:/Users/changliu/Desktop/lab3.X/meng_test.c

```
   break;
  default:printf("Unknown SD Card Detected!");
   break;
}

printf("\n");

error = getBootSectorData(); //read boot sector and keep necessary data in global variables
printf("getBootSectorData_error = %d",error);
if(error)
{
  printf("\n");
  printf("FAT32 not found!");  //FAT32 incompatible drive
  FAT32_active = 0;
}

while(1)
{
printf("\n");
printf("Press any key to start...");
printf("\n");
option = receiveByte();
printf("\n");
printf("> 0 : Erase Blocks");
printf("\n");
printf("> 1 : Write single Block");
printf("\n");
printf("> 2 : Read single Block");




#ifndef FAT_TESTING_ONLY
printf("\n");
printf("> 3 : Write multiple Blocks");
printf("\n");
printf("> 4 : Read multiple Blocks");
#endif


printf("\n");
printf("> 5 : Get file list");
printf("\n");
printf("> 6 : Read File");
printf("\n");
printf("> 7 : Write File");
printf("\n");
printf("> 8 : Delete File");
printf("\n");
printf("> 9 : Read SD Memory Capacity (Total/Free)");

printf("\n");
printf("\n");
printf("> Select Option (0-9): ");
```

```
/*WARNING: If option 0, 1 or 3 is selected, the card may not be detected by PC/Laptop again,
as it disturbs the FAT format, and you may have to format it again with FAT32.
This options are given for learnig the raw data transfer to & from the SD Card*/

option = receiveByte();
transmitByte(option);

if(option >=0x35 && option <=0x39)  //options 5 to 9 disabled if FAT32 not found
{
  if(!FAT32_active)
  {
    printf("\n");

    printf("\n");
printf("FAT32 options disabled!");
continue;
  }
}


if((option >= 0x30) && (option <=0x34)) //get starting block address for options 0 to 4
{
printf("\n");
printf("\n");
printf("Enter the Block number (0000-9999):");
data = receiveByte(); transmitByte(data);
startBlock = (data & 0x0f) * 1000;
data = receiveByte(); transmitByte(data);
startBlock += (data & 0x0f) * 100;
data = receiveByte(); transmitByte(data);
startBlock += (data & 0x0f) * 10;
data = receiveByte(); transmitByte(data);
startBlock += (data & 0x0f);
printf("\n");
printf("start_block_addr = %d" , startBlock);
printf("\n");

}

totalBlocks = 1;

#ifndef FAT_TESTING_ONLY

if((option == 0x30) || (option == 0x33) || (option == 0x34)) //get total number of blocks for options 0,
{
printf("\n");
printf("\n");
printf("How many blocks? (000-999):");
data = receiveByte(); transmitByte(data);
totalBlocks = (data & 0x0f) * 100;
data = receiveByte(); transmitByte(data);
totalBlocks += (data & 0x0f) * 10;
data = receiveByte(); transmitByte(data);
totalBlocks += (data & 0x0f);
```

```
printf("\n");
}
#endif

switch (option)
{
case '0': //error = SD_erase (block, totalBlocks);


          error = SD_erase (startBlock, totalBlocks);
         printf("\n");
          if(error)
              printf("Erase failed..");
          else
              printf("Erased!");
          break;

case '1':
          printf("\n");;
          printf(" Enter text (End with ~):");
          i=0;
            do
            {
                data = receiveByte();
                transmitByte(data);
                buffer[i++] = data;
                if(data == 0x0d)
                {
                    transmitByte(0x0a);
                    buffer[i++] = 0x0a;
                }
                if(i == 512) break;
            }while (data != '~');

            error = SD_writeSingleBlock (startBlock);
            printf("\n");
            printf("\n");
            if(error)
                printf("Write failed..");
            else
                printf("Write successful!");

            break;

case '2': error = SD_readSingleBlock (startBlock);   //SD_readSingleBlock (74100588);
          printf("\n");
          if(error)
            printf("Read failed..");
          else
          {
            for(i=0;i<512;i++)
            {
                if(buffer[i] == '~') break;
                transmitByte(buffer[i]);
            }
            printf("\n");
```

```
            printf("\n");
            printf("Read successful!");
          }

          break;
//next two options will work only if following macro is cleared from SD_routines.h
#ifndef FAT_TESTING_ONLY

case '3':
          error = SD_writeMultipleBlock (startBlock, totalBlocks);
          printf("\n");
          if(error)
            printf("Write failed..");
          else
            printf("Write successful!");
          break;

case '4': error = SD_readMultipleBlock (startBlock, totalBlocks);
          printf("\n");
          if(error)
            printf("Read failed..");
          else
            printf("Read successful!");
          break;
#endif

case '5': printf("\n");
          printf("\n");

    findFiles(GET_LIST,0);



          break;


case '6':
case '7':
case '8': printf("\n");
  printf("\n");
          printf("Enter file name: ");
          for(i=0; i<13; i++)
fileName[i] = 0x00;   //clearing any previously stored file name
          i=0;
          while(1)
          {
            data = receiveByte();
            if(data == 0x0d) break;  //'ENTER' key pressed
if(data == 0x08)//'Back Space' key pressed
 {
   if(i != 0)
   {
     transmitByte(data);
transmitByte(' ');
     transmitByte(data);
```

```
     i--;
   }
   continue;
 }
if(data <0x20 || data > 0x7e) continue;  //check for valid English text character
transmitByte(data);
            fileName[i++] = data;
            if(i==13){printf(" file name too long.."); break;}
          }
          if(i>12) break;

      printf("\n");
  if(option == '6')
     readFile( READ, fileName);
  if(option == '7')
   writeFile(fileName);
   if(option == '8')
     deleteFile(fileName);
          break;

case '9': memoryStatistics();
data = receiveByte();
          break;

default: printf("\n");
         printf("\n");
         printf(" Invalid option!");
         printf("\n");
}

printf("\n");
}
return 0;
}
```