<u>8051 Memory Card Reader</u>

Aniket Kumar Lata

Final Project Report

Embedded System Design

December 13th 2014

CONTENTS

1. Abstract

Memory crunch has been a major problem with embedded systems and computers over the years. Having enough memory enables the programmers to include more applications into the system. 8051 has the capability of processing and running multiple applications. One can even design and run an operating system over it. This can be made possible if all these applications have enough memory to work with.

With my final project, I aimed to provide 8051 embedded systems with sufficient amount of memory to run multiple programs over the system. My objective was to interface a sophisticated form of memory, the MultiMediaCard (MMC), later known as the SecureDigitalCard (SDC), to the 8051. The idea was to enable the 8051 to read from and write to specific locations on the SD Card, read data stored in the form of files from the memory card, copy data between sectors in the memory card and provide the application programs with a form of data memory for use.

The SD Card provides a huge amount of memory to the system which ranges from 128MB to 32GB. These SD cards are formatted with a file system which is used to store files and locate them from specific locations. The project successfully accomplishes the objectives of analysing the file system, retrieving the file information and location, reading the files which have been stored on the device and displaying these files on a graphic LCD.

Features:

- Memory Card information
- Memory read – Read from specific blocks selected by the user
- Memory write – write to a specific block selected by the user
- Multiple sector read - Hex Memory dump of locations across sectors
- File information display
- File Read – Read a specific file from the displayed file list
- Graphic LCD display – Display the file contents on a graphic LCD (The application can be used like a text file reader or eBook reader)

1.1 System Overview:

A good system overview can be provided with the block diagram in Figure 1.

The design incorporates several new components on the 8051 board which span hardware, firmware and software.

The hardware components and their interfaces can be shown clearly using the block diagram.

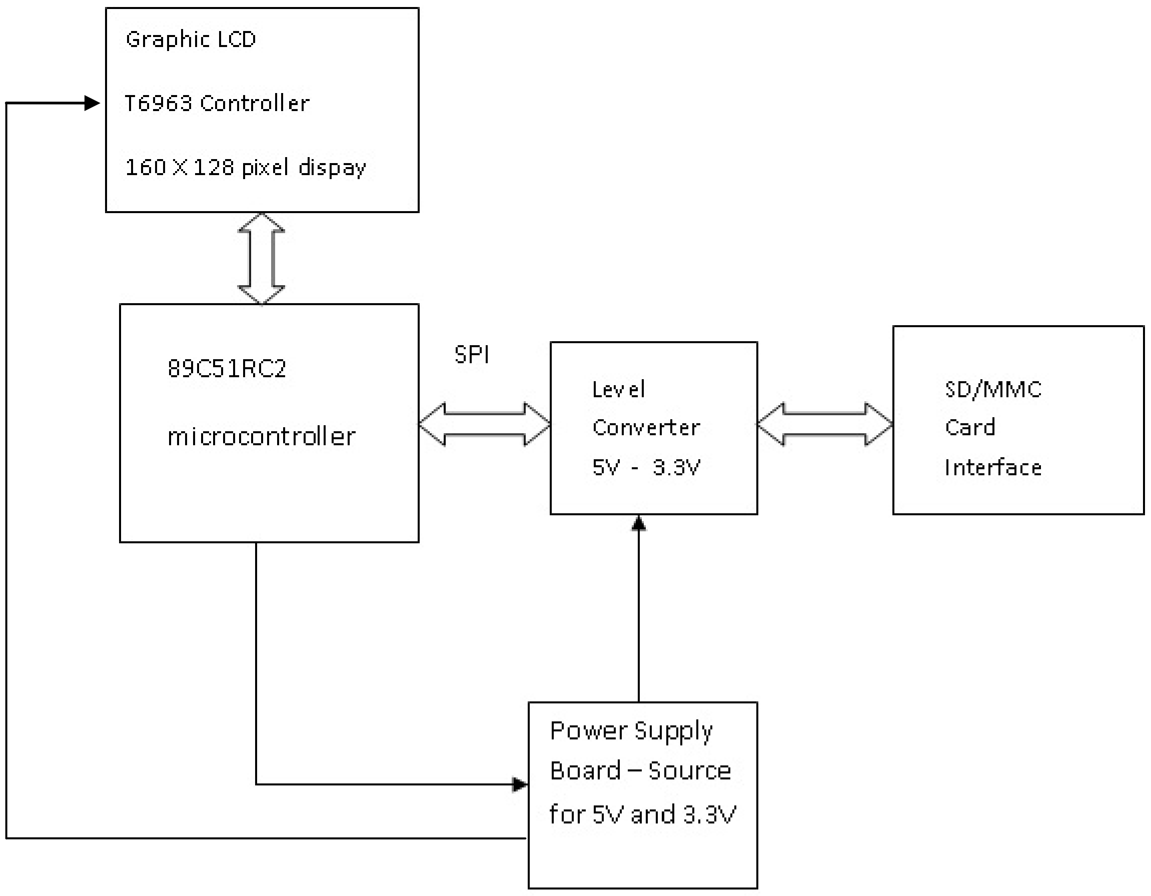


Figure 1.1 Block diagram – Memory Card Reader 8051

1.1.1 **89C51RC2 microcontroller**:

The 89C51RC2 microcontroller was used to implement the memory card reader functionality. The SPI interface has been used to communicate with the memory card through the level converter. A parallel interface to the graphic LCD is another connection from the microcontroller. The graphic LCD control signals have been provided from the port 2 of 89C51.

1.1.2 **Level converter**:

Level coveter has been used to convert logic levels from 5V to 3.3V and 3.3V to 5V for the bidirectional SPI communication between the microcontroller and the SD card module.

1.1.3 **SD/MMC Card interface**:

The SD/MMC Card interface is a module that holds the SD card and provides an electrical interface to communicate with the card. The SPI pins MISO, MOSI, SCK and SS are available on the card and have to be connected to the 8051 through the logic level converter.

1.1.4 **Graphic LCD**:

The graphic LCD provides an interface to the user to view contents of a memory card file. A parallel interface from the microcontroller connects the LCD to the 8051 circuitry.

1.1.5 **Power Supply**:

3.3V generation circuitry has been added to the board. The SD Card works on 3.3V.

LM317 voltage regulator has been made to output 3.3V using a voltage divider network. LM7805 has been used to generate 5V for LED backlight for the graphic LCD.

The system can divided into three major components:

1. The hardware interfaces
2. The firmware drivers
3. Application program and user interface

Along with these components, the system functionality can be understood by going over the appendices and references for each of the relevant topics mentioned in the report.

2. Hardware Interfaces

This project has been implemented on the 8051 development board that was being used for the labs #1 to #4. The board, at the point of the start of this project, had the below components mounted:

89C51R2 microcontroller, serial port, MAX232, NVSRAM 32KB, SPLD, power circuitry, reset circuitry, crystal oscillator and some more components used in the labs but not relevant from the project point of view. Below are the schematic diagrams of the existing circuitry on the board.

2.1 Existing Hardware

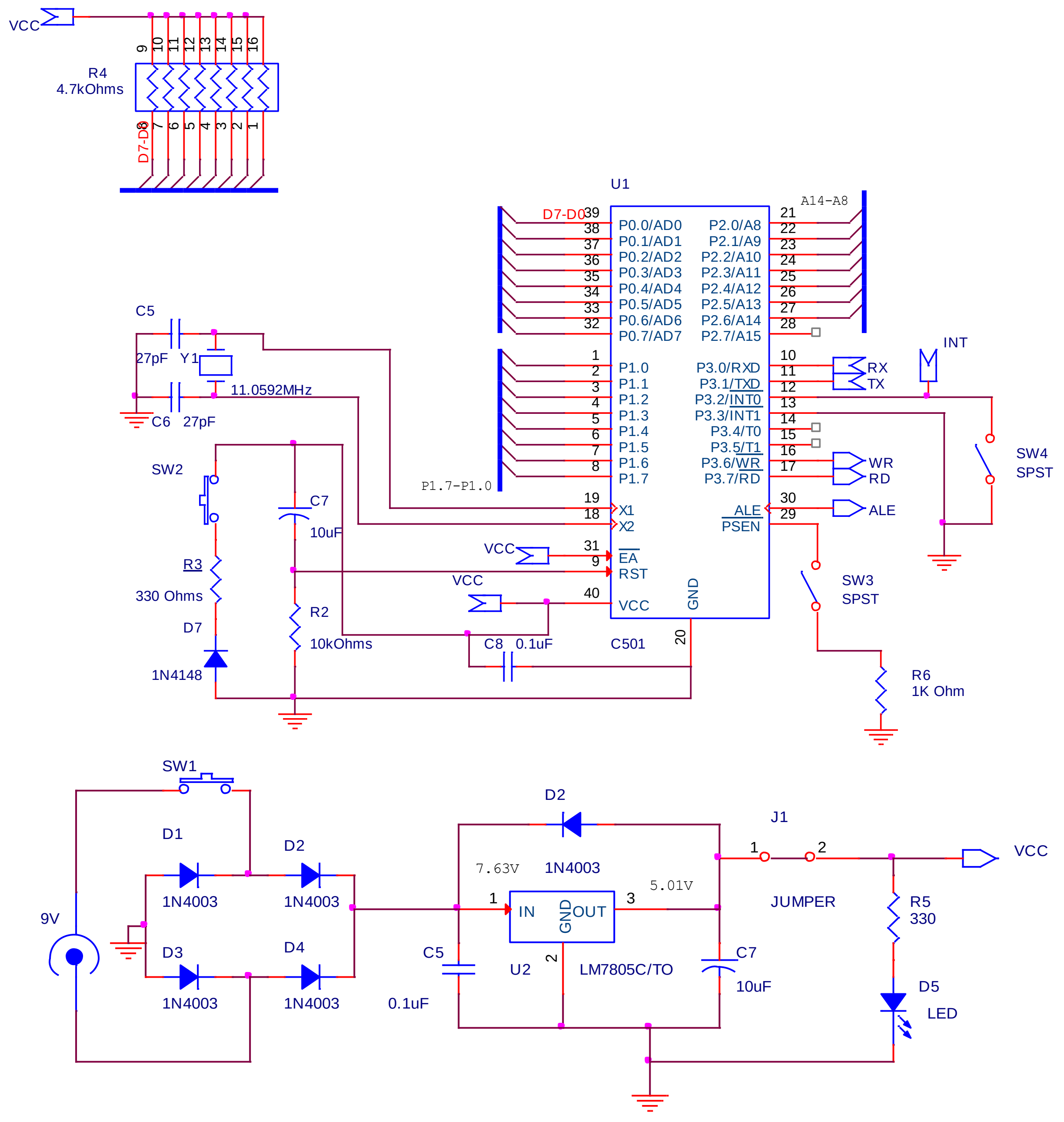


Figure 2.1 Microcontroller, power, reset circuitry

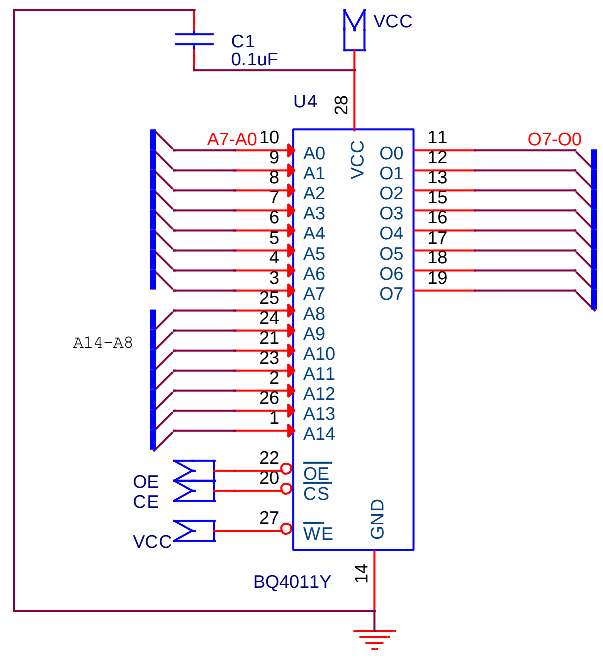


Figure 2.2 NVSRAM

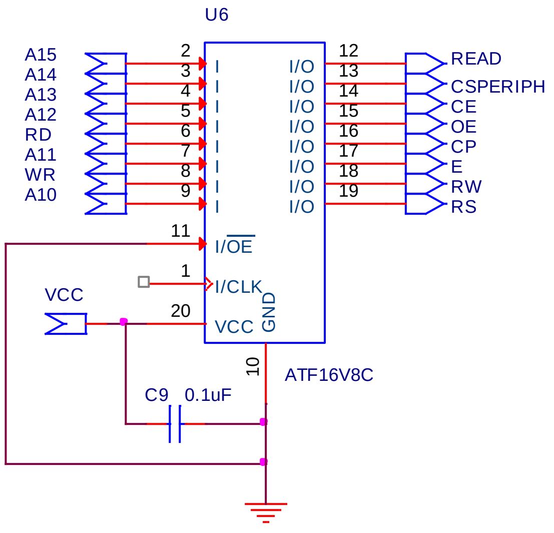


Figure 2.3 SPLD

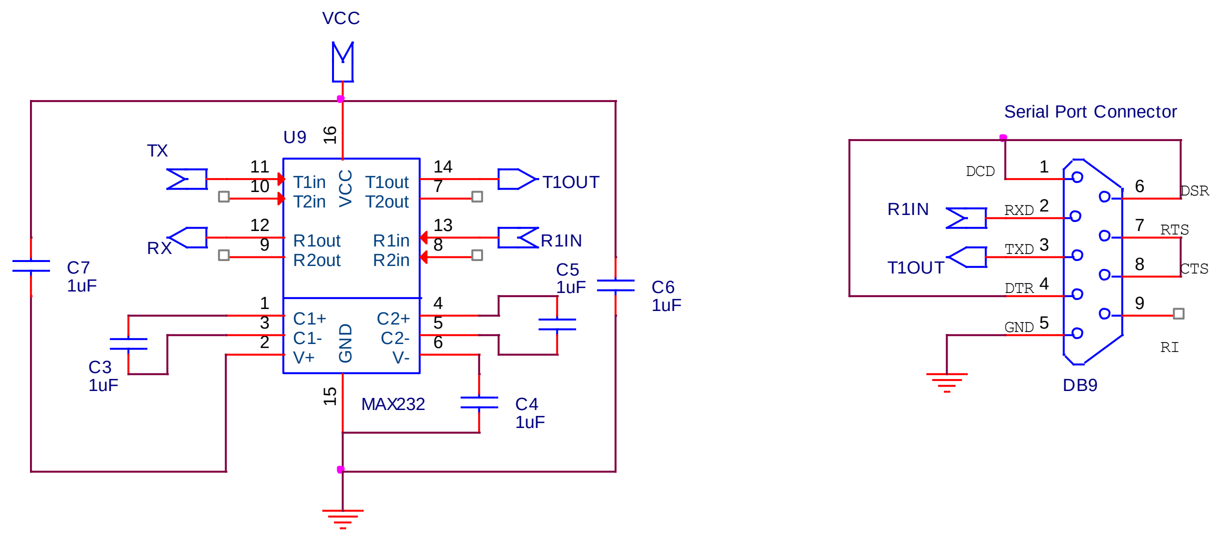


Figure 2.4 MAX232 and Serial port (RS232)

2.2 Hardware added for project:

As SD/MMC Card module for SD card interface was added to the board. The SD Card works in the voltage range 2.7-3.6V.

A level converter for converting logic levels from 5V to 3.3V was added to the board. The SPI lines from port 1 go to the logic level converter and its output lines go to the SD card module.

3.3V generation circuitry was added to the board. LM317 has been used to generate 3.3V. A voltage divider at the Adj pin of the LM317 has been designed to provide a 3.3V output.

The graphic LCD has been interfaced to the board using the parallel interface i.e. D7-D0 lines. The port 2 pins have been used as control signals for the LCD. For the 160 X 128 LCD, backlight supply has been provided using a separate 7805. This prevents the LCD from drawing more power from the on board 7805.

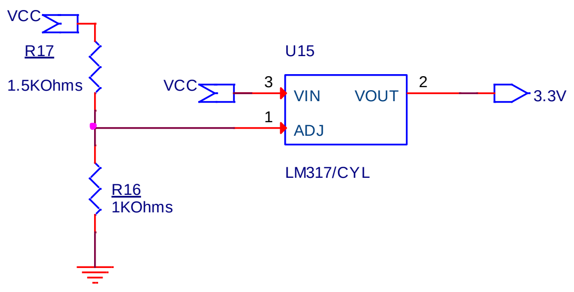


Figure 2.5 3.3V generation circuitry

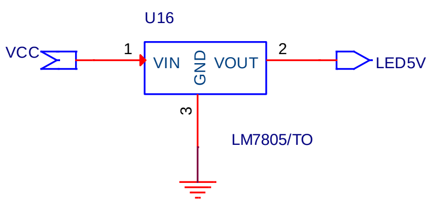


Figure 2.6 LED backlight power circuit

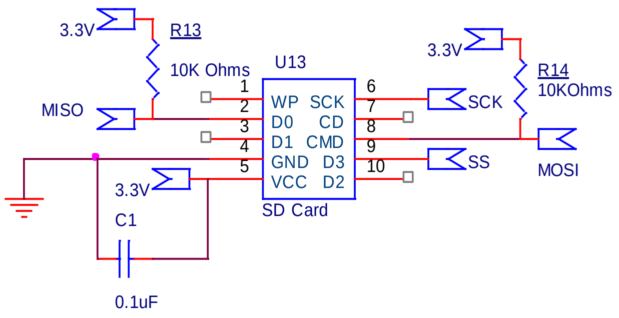


Figure 2.7 SD Card module

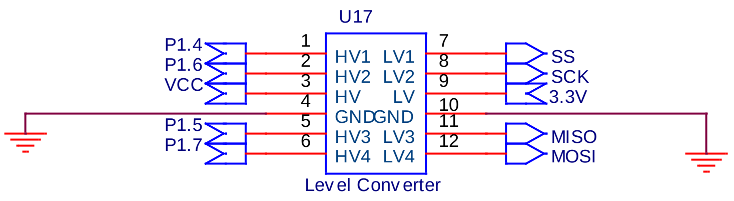
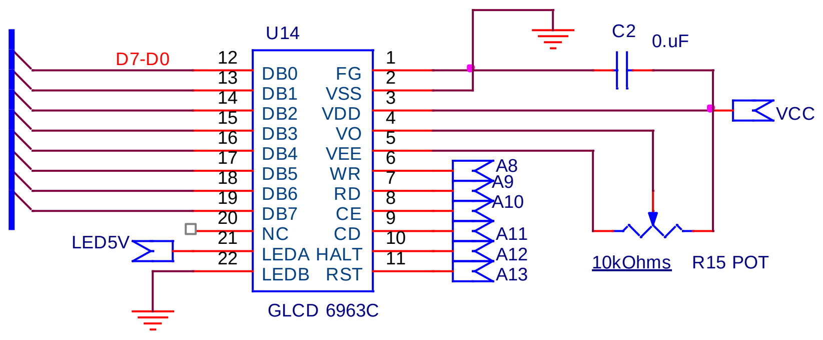


Figure 2.8 Level converter



U14
D7-D0
DB0 DB1 DB2 DB3 DB4 DB5 DB6 DB7 NC LEDA LEDB
FG VSS VDD VO VEE WR RD CE CD HALT RST
LED5V
A8 A9 A10 A11 A12 A13
C2
0.uF
VCC
10kOhms
R15 POT
GLCD 6963C

Figure 2.9 Graphic LCD interface circuit

2.2.1 Power circuit (3.3V generation):

The output voltage of the LM317 regulator can be controlled by using proper values of resisters R16 and R17.

Vo = 1.25(1+(R1/R2))

In this case, R1 = R17 and R2 = R16

Taking R2 as **1kOhms**,

3.3V = 1.25(1+(R1/1k))

Hence, R1 = 1640 Ohms

Approximating R1 = R17 = **1.5kOhms**

2.2.2 SD Card Module:

SPI protocol used to communicate with SD Card. MISO, MOSI, SCK and SS connected to SD Card module through the level converter. Pull up resistors added on lines MISO and MOSI of SD Card. Decoupling capacitor placed to protect the SD Card from high frequency transients from the supply. Below are the connections between the host MCU and the device:

MOSI – P1.7

MISO – P1.5

SCK – P1.6

SS – P1.4

2.2.3 Level Converter:

HV(5V) and LV(3.3V) supply provided to the level converter from power circuit.

2.2.4 Graphic LCD:

Graphic LCD interfaced using parallel data lines D7-D0 of the microcontroller. The control signals WR, RD, CE, RS, RESET and HALT have been provided from Port 2.

3. Code Structure:

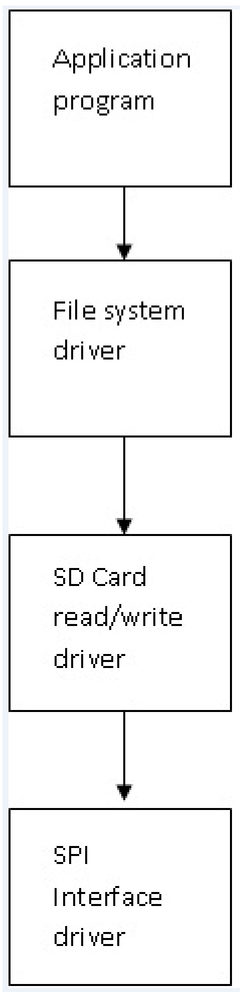


Figure 3.1 Code structure

The figure above denotes the code structure adopted to implement the memory card reader application.

The code base consists of the following components:

1) Application program	-	main.c
2) SD Card driver	-	sdcdriver.c
		sdcdriver.h
3) Graphic LCD driver	-	glcddriver.c
		glcddriver.h
4) Serial Port driver	-	serialinit.c
		serialinit.h
5) Data type header	-	integer.h

3.1 **Code overview**:

1. SD Card driver: The SD Card driver implements the low level SD Card initialization, read/ write functions, file read, sd card copy and other basic functions.
   These functions can be called from the application program to communicate with the SD card.

2. Graphic LCD driver: The Graphic LCD driver implements basic LCD initialization, read, write and cursor position functions. The code needed to communicate with the LC is written in this driver.

3. Serial driver: The serial driver implements low level code to communicate with the on board serial port and send/receive data.

4. Application program: The application program has been coded to execute functions written for each of the drivers. The application program provides features of memory card read, write, copy, multiple sector read, hex dump and file information.

Software: The application program constitutes the software written for the project.

Firmware: The serial, sd card and spi drivers constitute the firmware implemented for the project.

3.2 Firmware

A major part of the project has been the firmware written for the SD Card.

3.2.1 SPI driver

SPI allows full duplex, synchronous, serial communication between the MCU and serial devices. The SD Card can be accessed in SD Card (parallel) mode or in SPI mode. For this project, I have chosen to implement the SPI mode of operation.

The SPI protocol works on a command – response functionality. The device, i.e., the memory card in this case, has a set of commands wired into its internal controller. To perform operations on the memory card, it is necessary to send a set of commands to it. The card controller identifies these commands and returns valid responses to the host controller. As we have seen in the hardware configuration, four signals pins of the MCU have been connected to the memory card module:

- **Master In Slave Out (MISO):** Data transfer from Slave to a Master
- **Master Out Slave In (MOSI):** Data transfer from Master to a Slave
- **SPI Serial Clock (SCK):** Synchronizes data movement between host and device
- **Slave Select (SS):** Used to select a particular device. Active low signal

The microcontroller acts as a master in the SPI mode. It controls the data movement across MOSI and MISO lines.

Getting to the SPI driver, it is required to initialize the below SPI internal registers:

- SPCON
- SPDAT
- SPSTA

SPI Initialization:

The spi_init() function initializes the SPI interface by setting the clock frequency, master mode and enabling the SPI interrupt. The SPI functions like a UART. Whenever some data is written to the SPDAT register, the data is transmitted via MOSI line which connects to the CMD line of the device. This command may be a single byte or multiple bytes. The device responds at the end of command with a single or multiple byte response.

Refer to the initialization code below:

```
void spi_init()

{

  SPCON &= 0x00;            /* Fclk Periph/2 */

  SPCON |= 0x10;            /* Master mode */

  IEN1 |= 0x04;             /* enable spi interrupt */
}
SPCON |= 0x40;            // Enables SPI interface
```

To get the response, we need to make sure that the device is receiving the serial peripheral clock. For this purpose, to read data the microcontroller needs to keep sending dummy bytes to the device. This means simply writing to the SPDAT register. To avoid any confusion with valid commands, a dummy byte is signified by "'0xFF" in my entire driver code. As sending this byte provides clock signal to the device, it outputs the response on the MISO line. The microcontroller reads this data and puts it in the SPDAT register.

It is necessary that the transmission or reception of a byte is signalled. This signal is provided by the SPI interrupt generated by the microcontroller. Hence, when we are expecting a response, the program simply reads the SPDAT register within the interrupt service routine.

```
// SPI interrupt service routine
void int_SPI(void) __interrupt (9)          /* interrupt address is 0x004B */
{
        switch   (SPSTA)                    /* read and clear spi status register */
        {
                case 0x80:
    serial_data=SPDAT;                      /* read receive data */
    transmit_completed=1;                   /* set software flag */
    delay(100);
     break;
        }
}
```

SPSTA is the status register. The Most significant bit of this register being set denotes successful transmission or reception of data.

The Slave Select (SS) pin has been kept low from the start of the program since we have only one slave. Else, this line could have been used to select a particular device.

Thus, the SPI driver consists of sending and receiving bytes over the serial lines MISO and MOSI controlled by the SCK and SS signals from the microcontroller end.

Timing analysis for SPI:

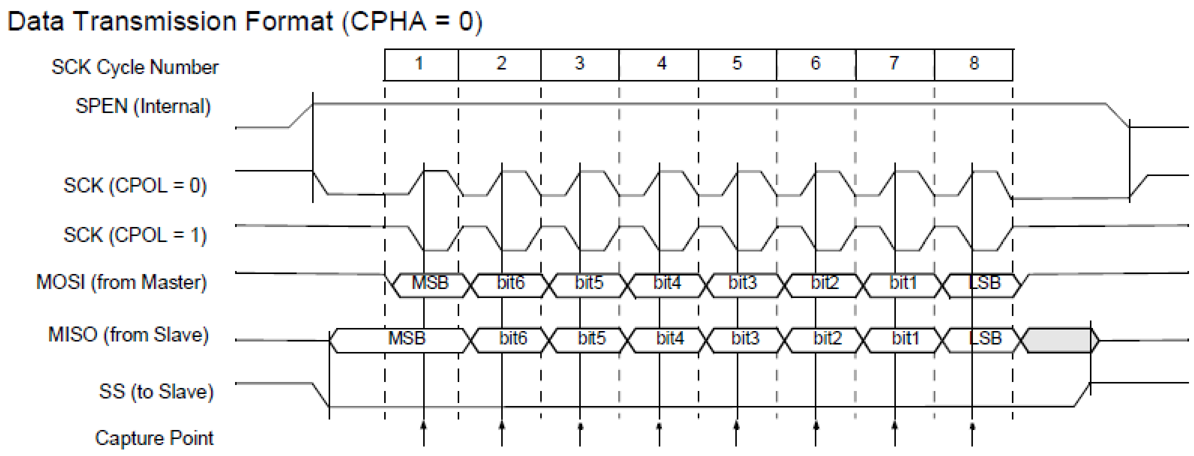


Figure 3.2 SPI data transmission format CPHA =0

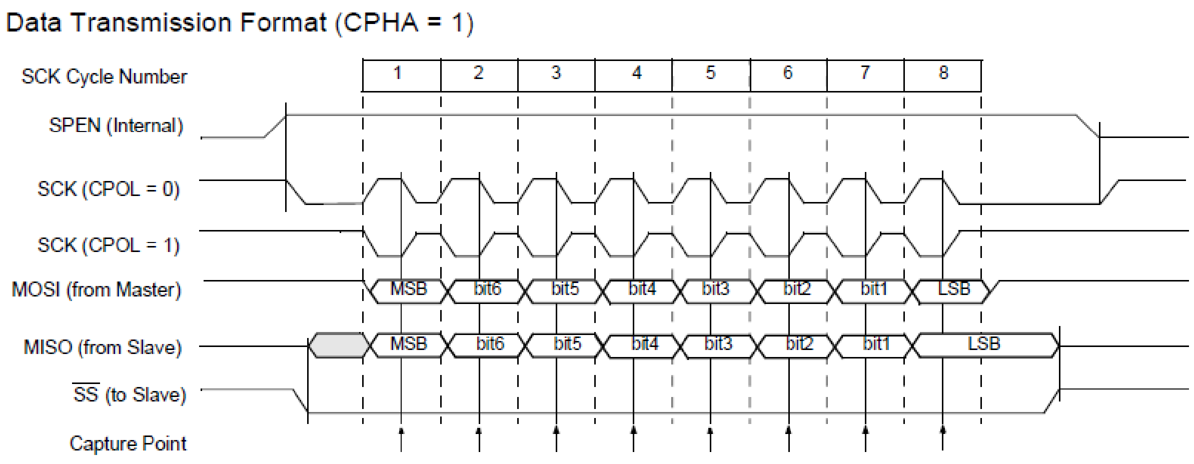


Figure 3.3 SPI data transmission format CPHA =1

The CPHA bit defines the edges on which the input data is sampled and the edges on which the output data is shifted. CPOL bit defines the default SCK line level in idle state.

In this project, CPHA and CPOL have been kept low, i.e. input is sampled/ output is shifted on falling edge & the SCK line is low in idle state.

SPEN bit is made ‘1’ at the start of the main SD card driver program and immediately SS pin is made ‘0’.

This takes care of the timing requirements for SPI protocol.

Before getting into the firmware implementation, below is an overview of SD Card commands/responses under SPI mode.

3.2.2 SD Card

A 1GB SanDisk SD Card formatted with FAT16 has been used in the project.

Below is a block diagram of the SD Card.

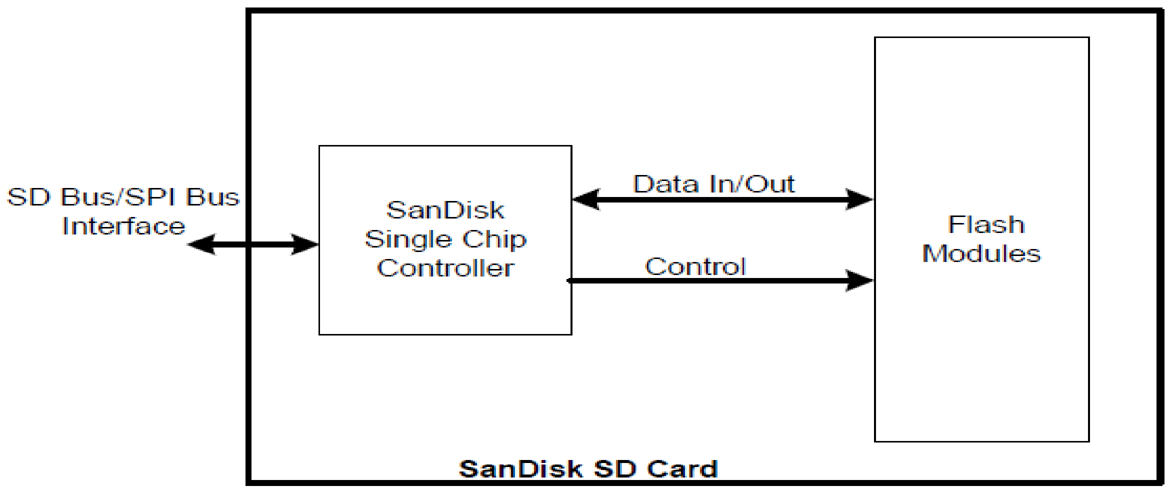


Figure 3.4 SD Card internal block diagram

The SD Card is interfaced to the 8051 microcontroller using SPI protocol. The SD Card supports either an SD bus interface or an SPI interface. The Single chip controller present inside the SD Card identifies commands sent to it and reads or writes data to the card. The voltage range for the SD card is 2.7V to 3.6V.

The SD Card pin out for SPI Mode is as below:

SPI Mode			
1	CS	I	Chip Select (active low)
2	DataIn	I	Host-to-card Commands and Data
3	V_{SS1}	S	Supply voltage ground
4	V_{DD}	S	Supply voltage
5	CLK	I	Clock
6	V_{SS2}	S	Supply voltage ground
7	DataOut	O	Card-to-host Data and Status
8	RSV[4]	---	Reserved
9	RSV[5]	---	Reserved

SPI Bus Protocol:

The SPI channel provides a byte oriented protocol. The SPI messages are built from command, response and data-block tokens. The host (master) controls all communication between host and cards. The host starts every bus transaction by asserting the CS signal, low.

The response behaviour in SPI Bus mode is as below:

1. The selected card always responds to the command.
2. An 8- or 16-bit response structure is used.
3. When the card encounters a data retrieval problem, it will respond with an error response

Mode Selection:

The SD Card wakes up in the SD Bus mode. It will enter SPI mode if the CS signal is asserted (negative) during the reception of the reset command (CMD0). If the card recognizes that the SD Bus mode is required it will not respond to the command and remain in the SD Bus mode. If SPI mode is required, the card will switch to SPI mode and respond with the SPI mode R1 response.

The default command structure/protocol for SPI mode is that CRC checking is disabled. Since the card powers up in SD Bus mode, CMD0 must be followed by a valid CRC byte (even though the command is sent using the SPI structure). Once in SPI mode, CRCs are disabled by default.

The entire CMD0 sequence appears as 40 00 00 00 00 95 (hexadecimal).

Data Read:

SPI mode supports single block and multiple-block read operations (SD Card CMD17 or CMD18). Upon reception of a valid read command the card will respond with a response token followed by a data token in the length defined in a previous SET_BLOCK_LENGTH (CMD16) command.

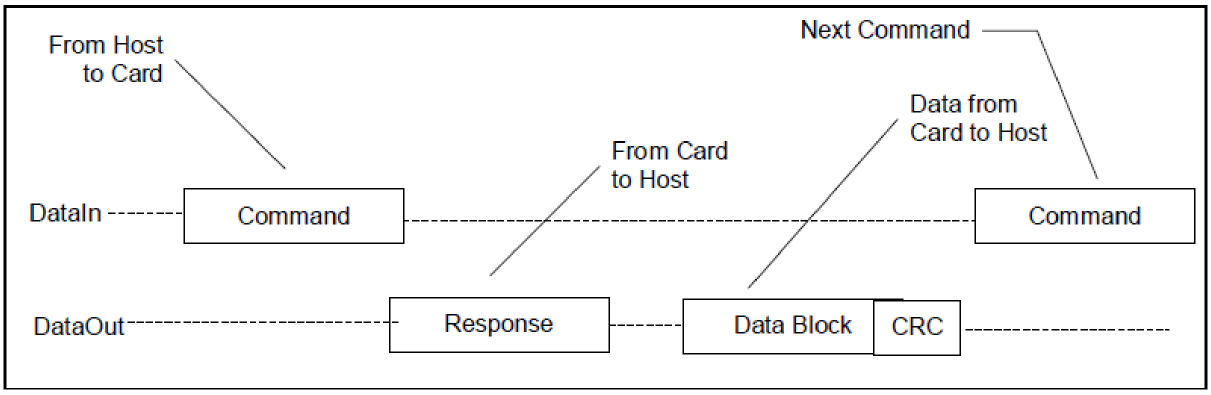


Figure 3.5 SD Read Operation

In the case of a Multiple Block Read operation, every transferred block has a 16-bit CRC suffix. The Stop Transmission command (CMD12) will actually stop the data transfer operation (the same as in SD Bus mode).

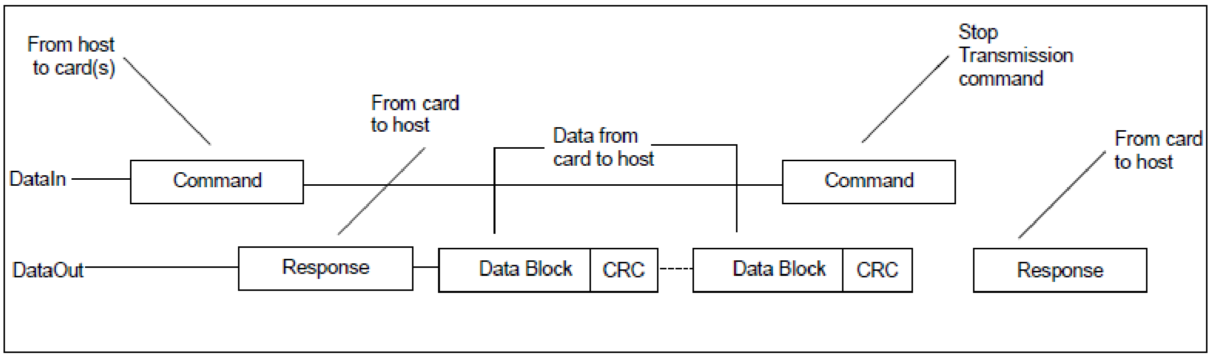


Figure 3.6 Multiple Data Read operation

Data Write:

In SPI mode, the SD Card supports single block or multiple-block write operations. Upon reception of a valid write command (SD Card CMD24 or CMD25), the card will respond with a response token and will wait for a data block to be sent from the host. CRC suffix and start address restrictions are identical to the read operation. The only valid block length, however, is 512 bytes. Setting a smaller block length will cause a write error on the next write command.

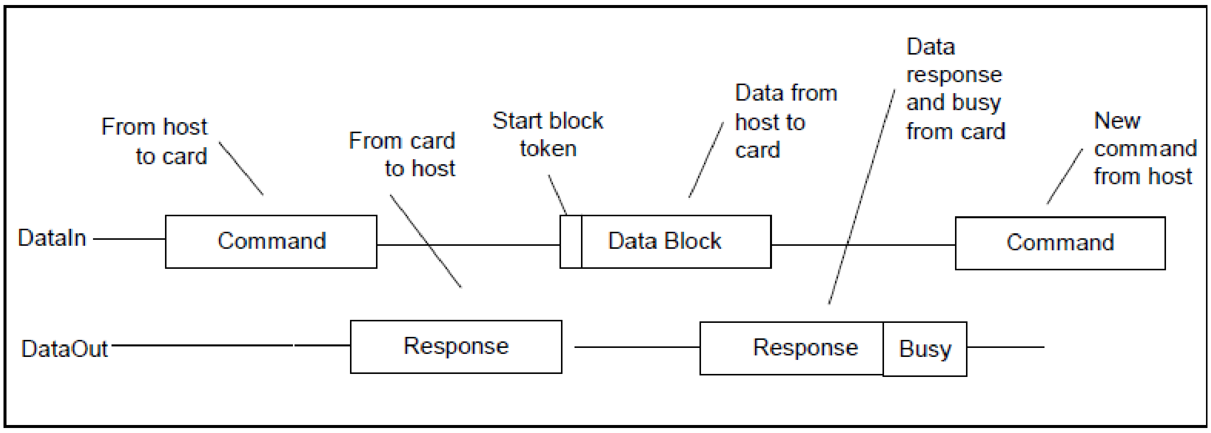


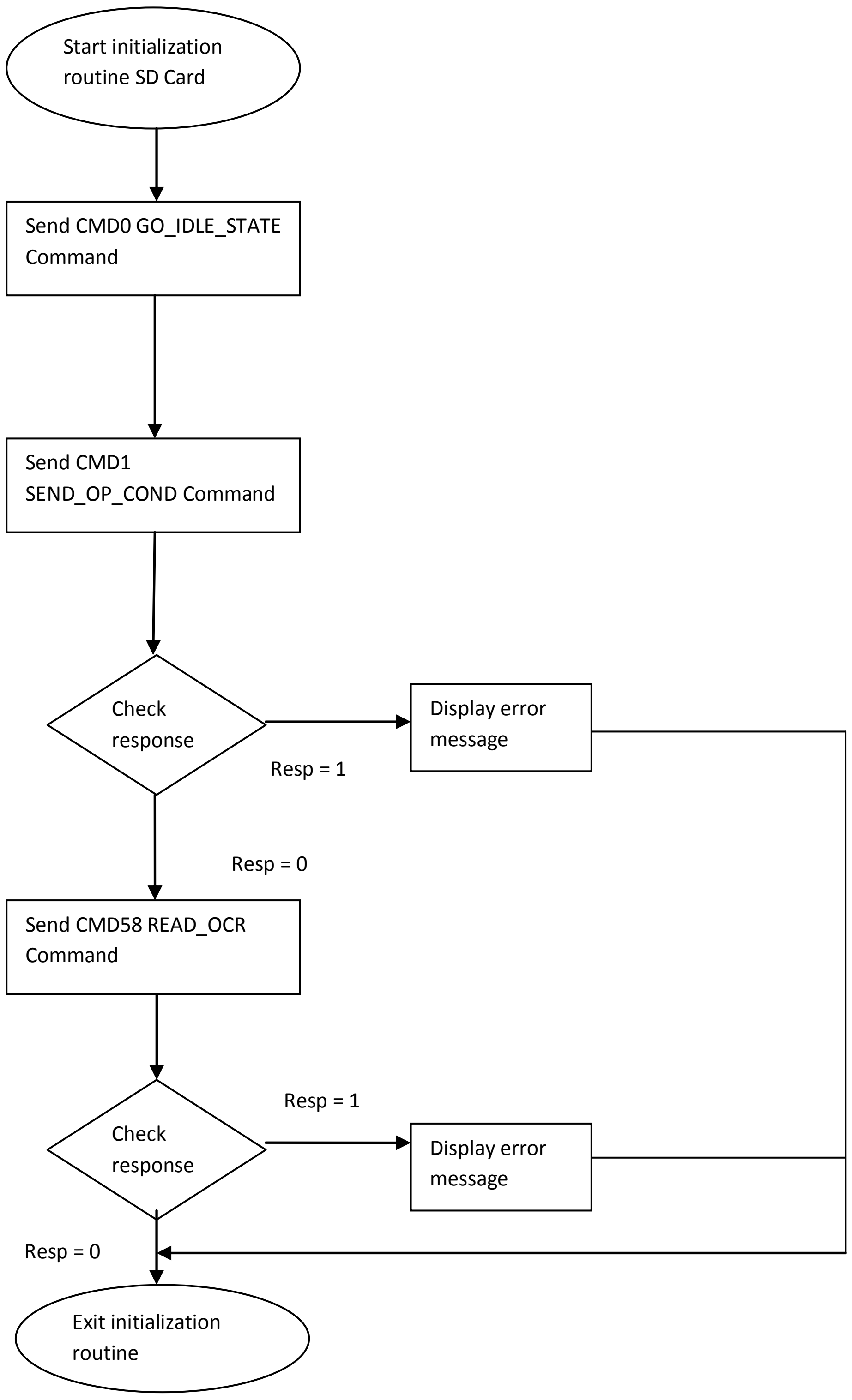
Figure 3.7 SD Data Write operation

3.2.3 SD Card driver

The SD Card responds to pre defined commands. It must be put in a mode so that it can accept and respond to commands from the host. This action is performed by the initialization code. It basically consists of a set of initial commands which reset the SD Card and put it into idle state. The responses of some of these commands provide information about the SD Card itself so that our driver program is designed to utilize the SD Card efficiently.

3.2.3.1 Initialization:

Below is a flowchart which explains the steps in SD Card initialization:


Start initialization routine SD Card
Send CMD0 GO_IDLE_STATE Command
Send CMD1 SEND_OP_COND Command
Check response
Resp = 1
Display error message
Resp = 0
Send CMD58 READ_OCR Command
Check response
Resp = 1
Display error message
Resp = 0
Exit initialization routine

The sd card initialization code has been written in **sd_init()** function.

The CMD0 command resets the SD card. The CMD1 command starts the SD Card's initialization process.

The Read Operating Conditions register (OCR) command reads the OCR of the SD card . The response of the Read OCR gives information about the operating levels of the SD Card.

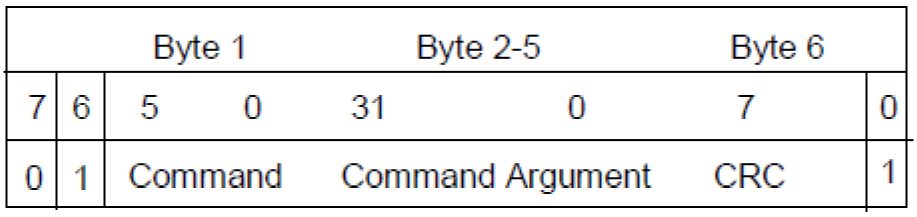
To know more about the SD Card, there is a Read CSD command (CMD8). This command gives information about the SD Card size, block length, transmission speed etc.

Once the card has been initialized, it is ready to accept read and write commands.

3.2.3.2 Single Sector Read:

The sector size of an SD Card can be set using the set block length command (CMD16). However, since we are using a 1GB card, its default block length is set to 512 bytes. A single sector read, reads 512 bytes from the address specified in the command argument. To understand the protocol successfully, we need to know the command structure of the SD Card:

All SD Card commands are six bytes long:



		Byte 1	Byte 2-5	Byte 6	
7	6	5 0	31 0	7	0
0	1	Command	Command Argument	CRC	1

Figure 3.8 SD Card Command Format

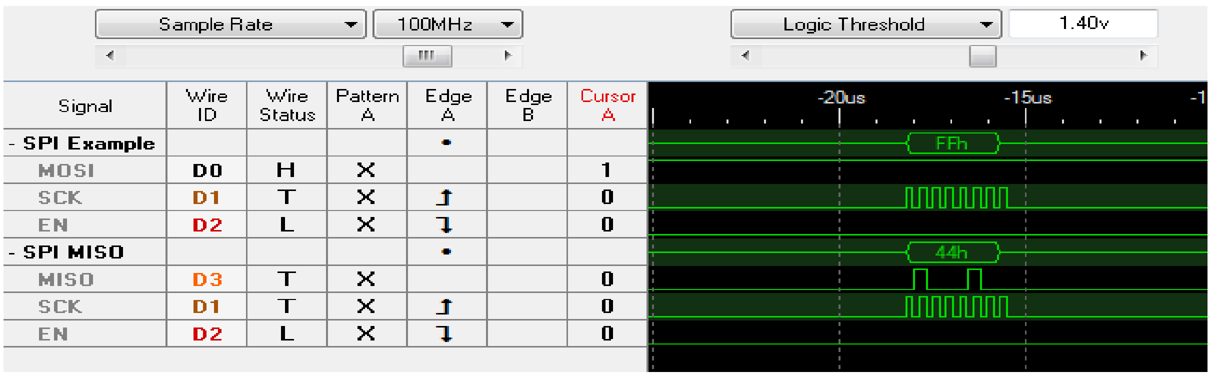
The first two bits are 0 & 1 indicating the start of a command.

The bits 5-0 of byte 1 represent the 6-bit command of the sd card. The next four bytes are the command arguments sent by the host MCU. Some commands require arguments while some don't require them. When an argument is not required, we send some dummy bytes in place of these arguments. The next byte is the CRC which is optional. We can send a dummy byte in place of CRC. The last bit needs to be 1 indicating the end of command.

When we send a read command i.e. CMD17, the device responds with a 0x00 if the command is received properly. The device then sends 512 continuous bytes when the clock pulses are supplied on the SCK line by the host MCU.

The single sector read functionality has been implemented in **sd_single_read()** function.

The timing diagram of a single byte being read has been captured using a logic analyzer. The snapshot is shown below:

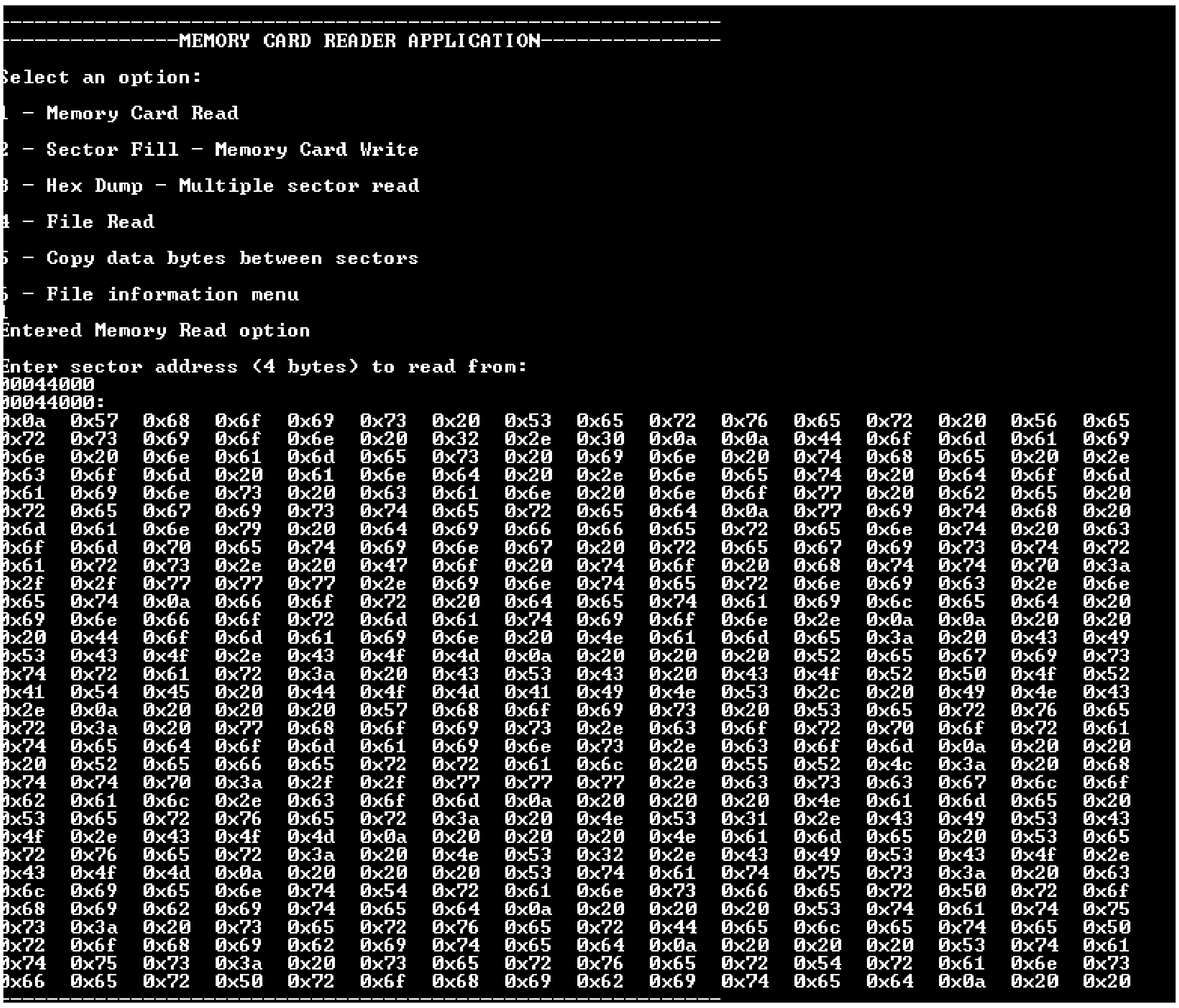


Sample Rate: 100MHz
Logic Threshold: 1.40v

Signal	Wire ID	Wire Status	Pattern A	Edge A	Edge B	Cursor A
- SPI Example				•		
MOSI	D0	H	X			1
SCK	D1	T	X	↥		0
EN	D2	L	X	↧		0
- SPI MISO				•		
MISO	D3	T	X			0
SCK	D1	T	X	↥		0
EN	D2	L	X	↧		0

Figure 3.9 SD Read operation

A 512 byte hex dump has been shown in the snapshot below after a read operation. The data sector starts from 0x00040000. This hex dump is from location 0x00044000.



```
-----------------------------------------------------------
--------------MEMORY CARD READER APPLICATION---------------

Select an option:

1 - Memory Card Read

2 - Sector Fill - Memory Card Write

3 - Hex Dump - Multiple sector read

4 - File Read

5 - Copy data bytes between sectors

6 - File information menu
1
Entered Memory Read option

Enter sector address (4 bytes) to read from:
00044000
00044000:
0x0a  0x57  0x68  0x6f  0x69  0x73  0x20  0x53  0x65  0x72  0x76  0x65  0x72  0x20  0x56  0x65
0x72  0x73  0x69  0x6f  0x6e  0x20  0x32  0x2e  0x30  0x0a  0x0a  0x44  0x6f  0x6d  0x61  0x69
0x6e  0x20  0x6e  0x61  0x6d  0x65  0x73  0x20  0x69  0x6e  0x20  0x74  0x68  0x65  0x20  0x2e
0x63  0x6f  0x6d  0x20  0x61  0x6e  0x64  0x20  0x2e  0x6e  0x65  0x74  0x20  0x64  0x6f  0x6d
0x61  0x69  0x6e  0x73  0x20  0x63  0x61  0x6e  0x20  0x6e  0x6f  0x77  0x20  0x62  0x65  0x20
0x72  0x65  0x67  0x69  0x73  0x74  0x65  0x72  0x65  0x64  0x0a  0x77  0x69  0x74  0x68  0x20
0x6d  0x61  0x6e  0x79  0x20  0x64  0x69  0x66  0x66  0x65  0x72  0x65  0x6e  0x74  0x20  0x63
0x6f  0x6d  0x70  0x65  0x74  0x69  0x6e  0x67  0x20  0x72  0x65  0x67  0x69  0x73  0x74  0x72
0x61  0x72  0x73  0x2e  0x20  0x47  0x6f  0x20  0x74  0x6f  0x20  0x68  0x74  0x74  0x70  0x3a
0x2f  0x2f  0x77  0x77  0x77  0x2e  0x69  0x6e  0x74  0x65  0x72  0x6e  0x69  0x63  0x2e  0x6e
0x65  0x74  0x0a  0x66  0x6f  0x72  0x20  0x64  0x65  0x74  0x61  0x69  0x6c  0x65  0x64  0x20
0x69  0x6e  0x66  0x6f  0x72  0x6d  0x61  0x74  0x69  0x6f  0x6e  0x2e  0x0a  0x0a  0x20  0x20
0x20  0x44  0x6f  0x6d  0x61  0x69  0x6e  0x20  0x4e  0x61  0x6d  0x65  0x3a  0x20  0x43  0x49
0x53  0x43  0x4f  0x2e  0x43  0x4f  0x4d  0x0a  0x20  0x20  0x20  0x52  0x65  0x67  0x69  0x73
0x74  0x72  0x61  0x72  0x3a  0x20  0x43  0x53  0x43  0x20  0x43  0x4f  0x52  0x50  0x4f  0x52
0x41  0x54  0x45  0x20  0x44  0x4f  0x4d  0x41  0x49  0x4e  0x53  0x2c  0x20  0x49  0x4e  0x43
0x2e  0x0a  0x20  0x20  0x20  0x57  0x68  0x6f  0x69  0x73  0x20  0x53  0x65  0x72  0x76  0x65
0x72  0x3a  0x20  0x77  0x68  0x6f  0x69  0x73  0x2e  0x63  0x6f  0x72  0x70  0x6f  0x72  0x61
0x74  0x65  0x64  0x6f  0x6d  0x61  0x69  0x6e  0x73  0x2e  0x63  0x6f  0x6d  0x0a  0x20  0x20
0x20  0x52  0x65  0x66  0x65  0x72  0x72  0x61  0x6c  0x20  0x55  0x52  0x4c  0x3a  0x20  0x68
0x74  0x74  0x70  0x3a  0x2f  0x2f  0x77  0x77  0x77  0x2e  0x63  0x73  0x63  0x67  0x6c  0x6f
0x62  0x61  0x6c  0x2e  0x63  0x6f  0x6d  0x0a  0x20  0x20  0x20  0x4e  0x61  0x6d  0x65  0x20
0x53  0x65  0x72  0x76  0x65  0x72  0x3a  0x20  0x4e  0x53  0x31  0x2e  0x43  0x49  0x53  0x43
0x4f  0x2e  0x43  0x4f  0x4d  0x0a  0x20  0x20  0x20  0x4e  0x61  0x6d  0x65  0x20  0x53  0x65
0x72  0x76  0x65  0x72  0x3a  0x20  0x4e  0x53  0x32  0x2e  0x43  0x49  0x53  0x43  0x4f  0x2e
0x43  0x4f  0x4d  0x0a  0x20  0x20  0x20  0x53  0x74  0x61  0x74  0x75  0x73  0x3a  0x20  0x63
0x6c  0x69  0x65  0x6e  0x74  0x54  0x72  0x61  0x6e  0x73  0x66  0x65  0x72  0x50  0x72  0x6f
0x68  0x69  0x62  0x69  0x74  0x65  0x64  0x0a  0x20  0x20  0x20  0x53  0x74  0x61  0x74  0x75
0x73  0x3a  0x20  0x73  0x65  0x72  0x76  0x65  0x72  0x44  0x65  0x6c  0x65  0x74  0x65  0x50
0x72  0x6f  0x68  0x69  0x62  0x69  0x74  0x65  0x64  0x0a  0x20  0x20  0x20  0x53  0x74  0x61
0x74  0x75  0x73  0x3a  0x20  0x73  0x65  0x72  0x76  0x65  0x72  0x54  0x72  0x61  0x6e  0x73
0x66  0x65  0x72  0x50  0x72  0x6f  0x68  0x69  0x62  0x69  0x74  0x65  0x64  0x0a  0x20  0x20
-----------------------------------------------------------
```

Figure 3.10 SD card Read hex dump

The timing diagram below explains the sequence of command and response in single sector read:

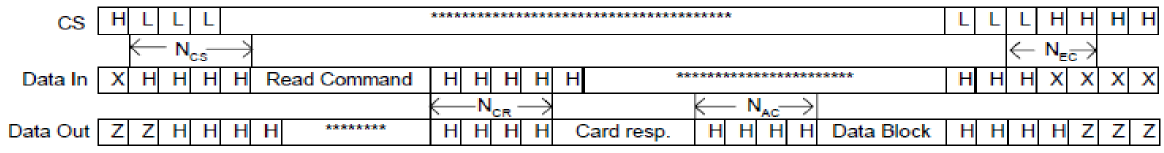


Figure 3.11 Single block read

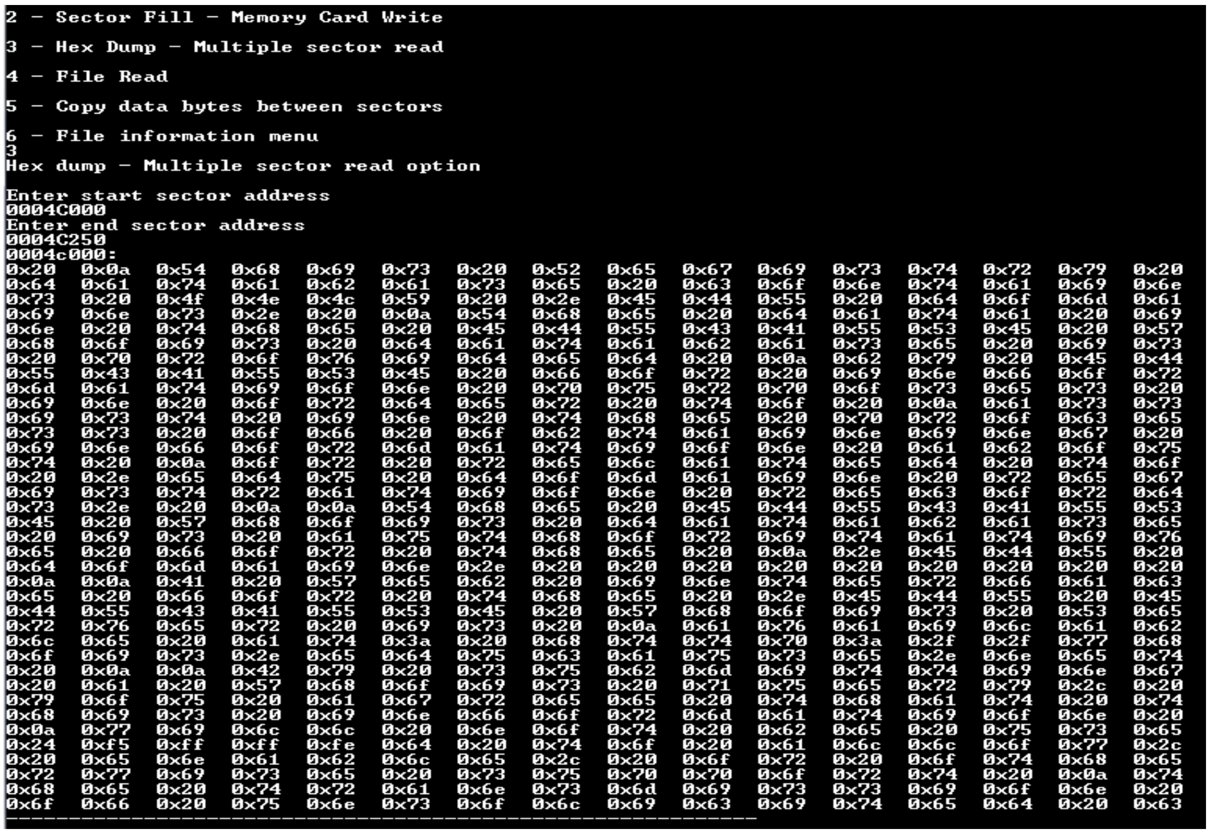
3.2.3.3 Multiple Sector Read:

The multiple sector read command is similar to single sector read command. The only difference is that the data read operation does not stop after a single sector read. The data read responses are sent by the device until the host MCU signals a stop transmission command.

The command argument presented in this case is the 32-bit address from which the read operation should start.

The sd_multiple_read() function implements this functionality. For multiple read, the program takes start and end address from the user itself. The stop transmission command is sent once the end address is reached.

Below is the snapshot for multiple sector read implemented in the project:



```
2 - Sector Fill - Memory Card Write
3 - Hex Dump - Multiple sector read
4 - File Read
5 - Copy data bytes between sectors
6 - File information menu
3
Hex dump - Multiple sector read option

Enter start sector address
0004C000
Enter end sector address
0004C250
0004c000:
0x20  0x0a  0x54  0x68  0x69  0x73  0x20  0x52  0x65  0x67  0x69  0x73  0x74  0x72  0x79  0x20
0x64  0x61  0x74  0x61  0x62  0x61  0x73  0x65  0x20  0x63  0x6f  0x6e  0x74  0x61  0x69  0x6e
0x73  0x20  0x4f  0x4e  0x4c  0x59  0x20  0x2e  0x45  0x44  0x55  0x20  0x64  0x6f  0x6d  0x61
0x69  0x6e  0x73  0x2e  0x20  0x0a  0x54  0x68  0x65  0x20  0x64  0x61  0x74  0x61  0x20  0x69
0x6e  0x20  0x74  0x68  0x65  0x20  0x45  0x44  0x55  0x43  0x41  0x55  0x53  0x45  0x20  0x57
0x68  0x6f  0x69  0x73  0x20  0x64  0x61  0x74  0x61  0x62  0x61  0x73  0x65  0x20  0x69  0x73
0x20  0x70  0x72  0x6f  0x76  0x69  0x64  0x65  0x64  0x20  0x0a  0x62  0x79  0x20  0x45  0x44
0x55  0x43  0x41  0x55  0x53  0x45  0x20  0x66  0x6f  0x72  0x20  0x69  0x6e  0x66  0x6f  0x72
0x6d  0x61  0x74  0x69  0x6f  0x6e  0x20  0x70  0x75  0x72  0x70  0x6f  0x73  0x65  0x73  0x20
0x69  0x6e  0x20  0x6f  0x72  0x64  0x65  0x72  0x20  0x74  0x6f  0x20  0x0a  0x61  0x73  0x73
0x69  0x73  0x74  0x20  0x69  0x6e  0x20  0x74  0x68  0x65  0x20  0x70  0x72  0x6f  0x63  0x65
0x73  0x73  0x20  0x6f  0x66  0x20  0x6f  0x62  0x74  0x61  0x69  0x6e  0x69  0x6e  0x67  0x20
0x69  0x6e  0x66  0x6f  0x72  0x6d  0x61  0x74  0x69  0x6f  0x6e  0x20  0x61  0x62  0x6f  0x75
0x74  0x20  0x0a  0x6f  0x72  0x20  0x72  0x65  0x6c  0x61  0x74  0x65  0x64  0x20  0x74  0x6f
0x20  0x2e  0x65  0x64  0x75  0x20  0x64  0x6f  0x6d  0x61  0x69  0x6e  0x20  0x72  0x65  0x67
0x69  0x73  0x74  0x72  0x61  0x74  0x69  0x6f  0x6e  0x20  0x72  0x65  0x63  0x6f  0x72  0x64
0x73  0x2e  0x20  0x0a  0x0a  0x54  0x68  0x65  0x20  0x45  0x44  0x55  0x43  0x41  0x55  0x53
0x45  0x20  0x57  0x68  0x6f  0x69  0x73  0x20  0x64  0x61  0x74  0x61  0x62  0x61  0x73  0x65
0x20  0x69  0x73  0x20  0x61  0x75  0x74  0x68  0x6f  0x72  0x69  0x74  0x61  0x74  0x69  0x76
0x65  0x20  0x66  0x6f  0x72  0x20  0x74  0x68  0x65  0x20  0x0a  0x2e  0x45  0x44  0x55  0x20
0x64  0x6f  0x6d  0x61  0x69  0x6e  0x2e  0x20  0x20  0x20  0x20  0x20  0x20  0x20  0x20  0x20
0x0a  0x0a  0x41  0x20  0x57  0x65  0x62  0x20  0x69  0x6e  0x74  0x65  0x72  0x66  0x61  0x63
0x65  0x20  0x66  0x6f  0x72  0x20  0x74  0x68  0x65  0x20  0x2e  0x45  0x44  0x55  0x20  0x45
0x44  0x55  0x43  0x41  0x55  0x53  0x45  0x20  0x57  0x68  0x6f  0x69  0x73  0x20  0x53  0x65
0x72  0x76  0x65  0x72  0x20  0x69  0x73  0x20  0x0a  0x61  0x76  0x61  0x69  0x6c  0x61  0x62
0x6c  0x65  0x20  0x61  0x74  0x3a  0x20  0x68  0x74  0x74  0x70  0x3a  0x2f  0x2f  0x77  0x68
0x6f  0x69  0x73  0x2e  0x65  0x64  0x75  0x63  0x61  0x75  0x73  0x65  0x2e  0x6e  0x65  0x74
0x20  0x0a  0x0a  0x42  0x79  0x20  0x73  0x75  0x62  0x6d  0x69  0x74  0x74  0x69  0x6e  0x67
0x20  0x61  0x20  0x57  0x68  0x6f  0x69  0x73  0x20  0x71  0x75  0x65  0x72  0x79  0x2c  0x20
0x79  0x6f  0x75  0x20  0x61  0x67  0x72  0x65  0x65  0x20  0x74  0x68  0x61  0x74  0x20  0x74
0x68  0x69  0x73  0x20  0x69  0x6e  0x66  0x6f  0x72  0x6d  0x61  0x74  0x69  0x6f  0x6e  0x20
0x0a  0x77  0x69  0x6c  0x6c  0x20  0x6e  0x6f  0x74  0x20  0x62  0x65  0x20  0x75  0x73  0x65
0x24  0xf5  0xff  0xff  0xfe  0x64  0x20  0x74  0x6f  0x20  0x61  0x6c  0x6c  0x6f  0x77  0x2c
0x20  0x65  0x6e  0x61  0x62  0x6c  0x65  0x2c  0x20  0x6f  0x72  0x20  0x6f  0x74  0x68  0x65
0x72  0x77  0x69  0x73  0x65  0x20  0x73  0x75  0x70  0x70  0x6f  0x72  0x74  0x20  0x0a  0x74
0x68  0x65  0x20  0x74  0x72  0x61  0x6e  0x73  0x6d  0x69  0x73  0x73  0x69  0x6f  0x6e  0x20
0x6f  0x66  0x20  0x75  0x6e  0x73  0x6f  0x6c  0x69  0x63  0x69  0x74  0x65  0x64  0x20  0x63
------------------------------------------------------------
```

Figure 3.12 Multiple Sector Read

3.2.3.4 Single Sector Write:

The single sector write command writes the data transferred on the MOSI line to the address specified in the command argument. A 512 byte sector is written with data supplied to the device byte by byte.

Below is the command description of the single sector write:

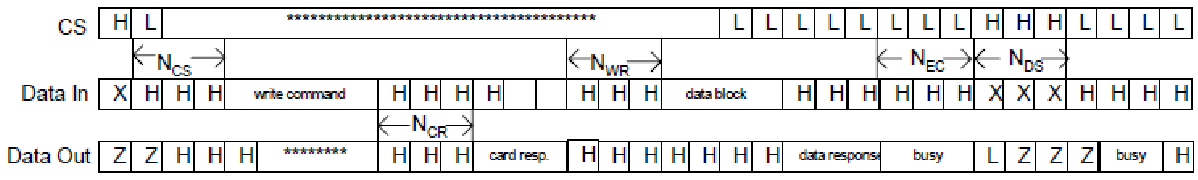
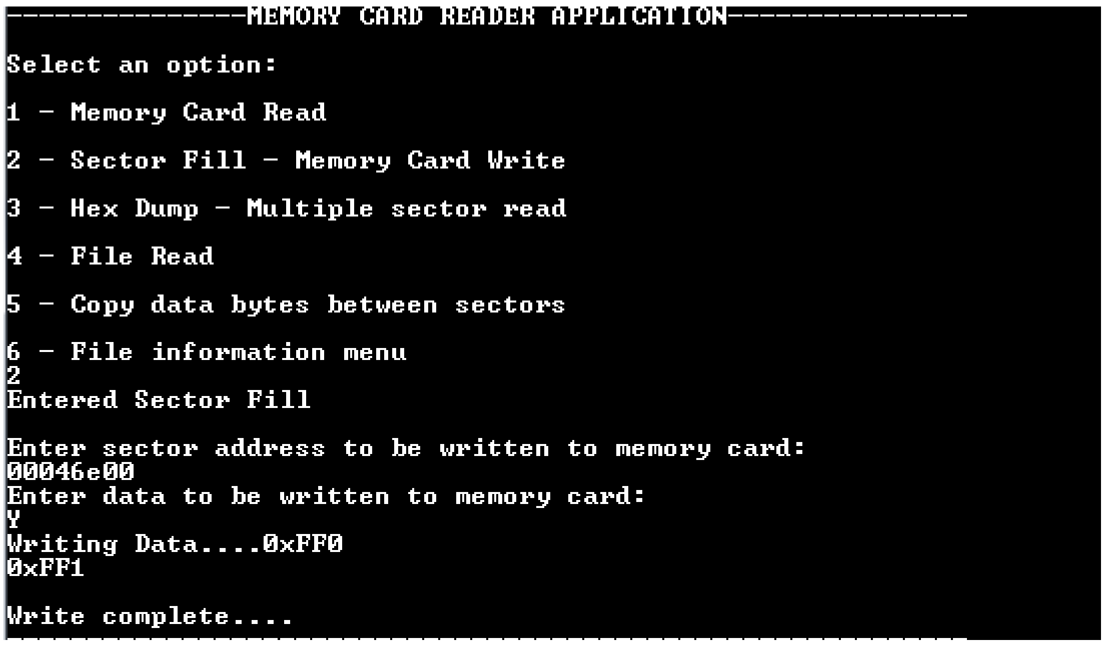


Figure 3.13 Single Sector Write

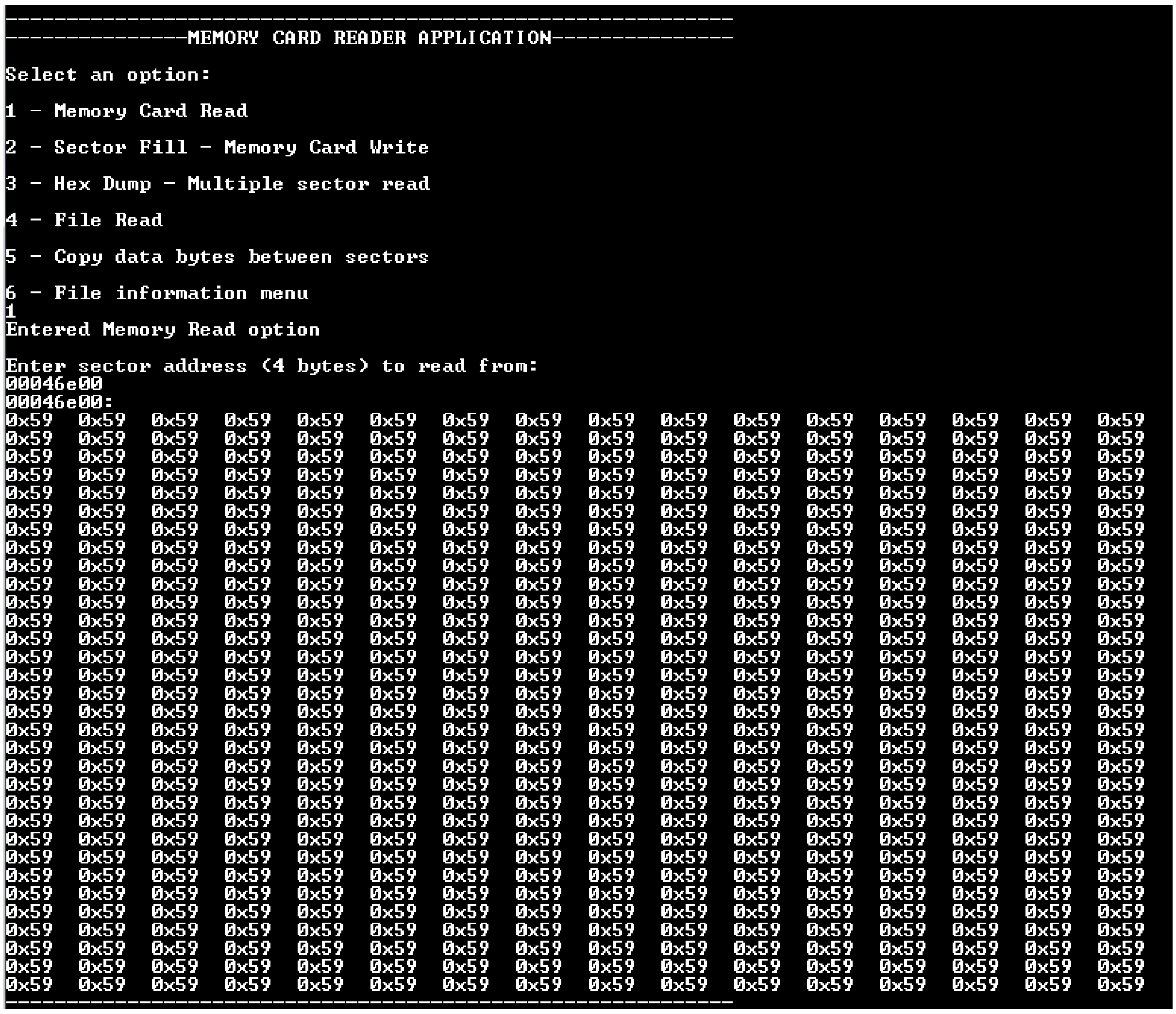
The single sector write has been implemented in the **sd_single_write()** function in the sdcdriver. This command can be used for block fill purposes or copying data from one sector to another.

A snapshot of block write implementation is given below:



```
---------------MEMORY CARD READER APPLICATION---------------

Select an option:

1 - Memory Card Read

2 - Sector Fill - Memory Card Write

3 - Hex Dump - Multiple sector read

4 - File Read

5 - Copy data bytes between sectors

6 - File information menu
2
Entered Sector Fill

Enter sector address to be written to memory card:
00046e00
Enter data to be written to memory card:
Y
Writing Data....0xFF0
0xFF1

Write complete....
------------------------------------------------------------
```

Figure 3.14 Single sector write command accepted



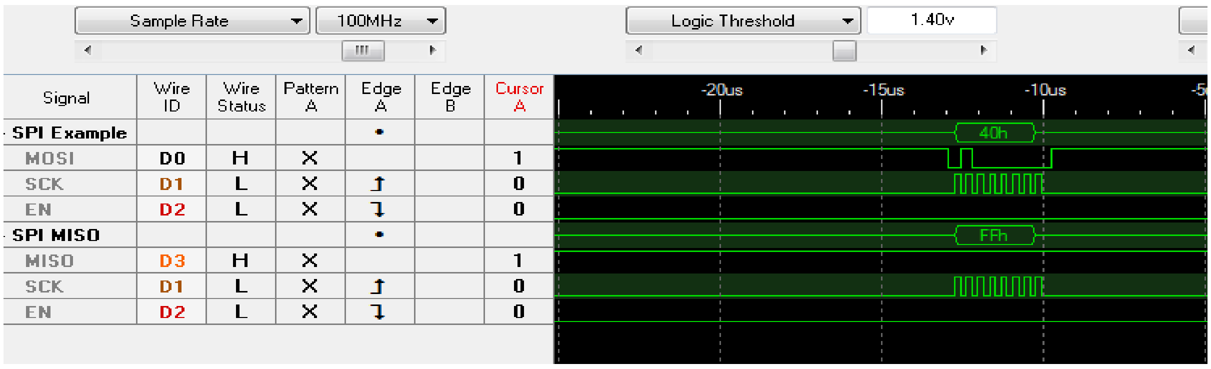
```
------------------------------------------------------------
---------------MEMORY CARD READER APPLICATION---------------
Select an option:
1 - Memory Card Read
2 - Sector Fill - Memory Card Write
3 - Hex Dump - Multiple sector read
4 - File Read
5 - Copy data bytes between sectors
6 - File information menu
1
Entered Memory Read option
Enter sector address (4 bytes) to read from:
00046e00
00046e00:
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59  0x59
------------------------------------------------------------
```

Figure 3.15 Data Read after sector fill operation

The snapshots show that a sector fill was performed using sd_single_write().

The character 'Y' was written to the address 0x00046e00. We can see that the ASCII value of 'Y' appears at the given address after the block write.

A snapshot of the timing diagram capture of a single byte write is given below:



Sample Rate 100MHz   Logic Threshold 1.40v

Signal	Wire ID	Wire Status	Pattern A	Edge A	Edge B	Cursor A
SPI Example				•		
MOSI	D0	H	X			1
SCK	D1	L	X	↥		0
EN	D2	L	X	↧		0
SPI MISO				•		
MISO	D3	H	X			1
SCK	D1	L	X	↥		0
EN	D2	L	X	↧		0

Figure 3.16 Sector Fill timing diagram

3.2.3.5 Memory Copy:

Memory copy functionality has been implemented in the application program using single sector read and single sector write functions.

A single sector is read using sd_single_read(). This data is copied to a temporary buffer. This temporary buffer is provided as an input for single sector write i.e. sd_single_write() function. This operation implements memory copy functionality in the application program.

Below is a code snippet of the memory copy function:

```
// Copy data from one sector to another

void sd_copy(ULONG srcsect,ULONG destsect)

{

   sd_single_read(srcsect);          // read single sector

   memcpy(tempbuf,buff,512);         // Copy data to temporary buffer

   sd_block_write(destsect);         // Write data to destination sector

}
```

3.2.3.6 File Read:

File read functionality identifies a file present in the memory card and reads the file by extracting the file size from the file allocation table.

To understand the file read functionality, an overview of the FAT16 file system has been provided below:

FAT16

The FAT16 file system consists of the following regions:

- Boot Sector
- File Allocation Table
- Root Directory
- Data Region

Boot sector:

The first sector in the reserved region is the boot sector. Though this sector is typically 512 bytes in can be longer depending on the media. The boot sector typical start with a 3 byte jump instruction to where the bootstrap code is stored, followed by an 8 byte long string set by the creating operating system. This is followed by the BIOS Parameter Block, and then by an Extended BIOS Parameter Block. Finally the boot sector contains boot code and a signature.

File Allocation Table (FAT):

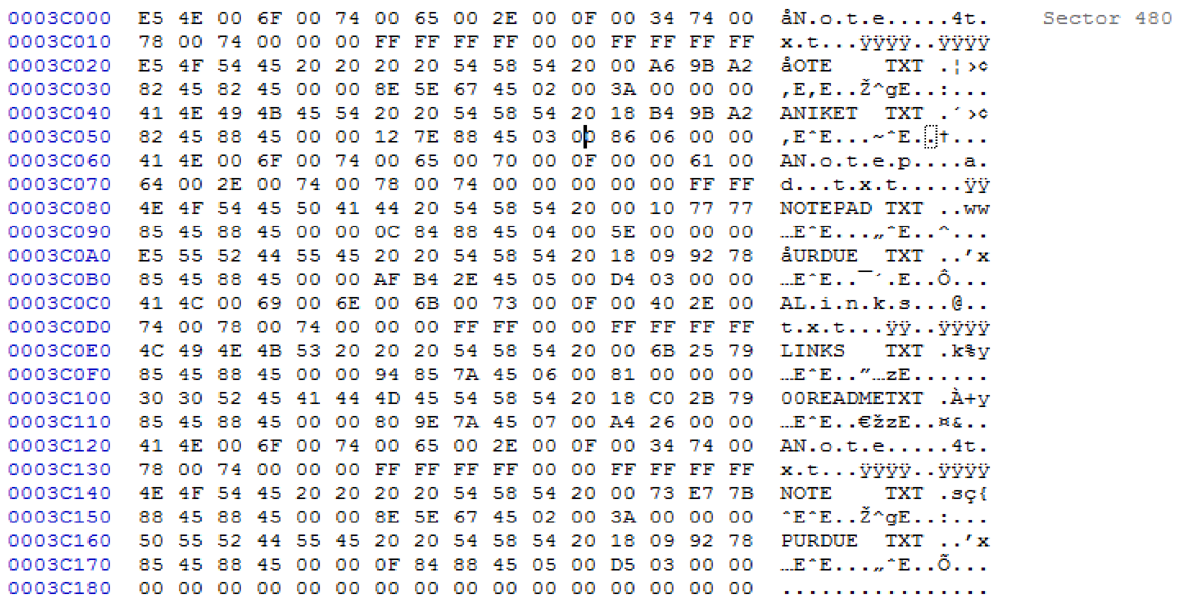
The FAT structure contain linked lists of files in the file system. Any file or directory entry in a (sub)directory list contain a cluster number for the first chunk of the file/directory. This cluster number also has an associated entry in the FAT. At this entry in the FAT a single word value either points to the next cluster/chunk or it contain an End-of-file value.

Directory Entry structure

Each of the entries in a directory list is 32 byte long. The only directory which is in a fixed location is the root directory. This is also the only directory which may contain an entry with the Volume Label attribute set. The size of the root directory is defined in the BIOS Parameter Block.

Sub-directories are created by allocating a cluster which then are cleared so it doesn't contain any directory entries. Two default entries are then created: The '.' entry point to the directory itself, and the '..' entry points to the parent directory. If the contents of a sub-directory grows beyond what can be in the cluster a new cluster is allocated in the same way as for files.

An hex editor HxD was used in the project to view these regions of the file system. A snapshot of the directory entry structure is shown below:



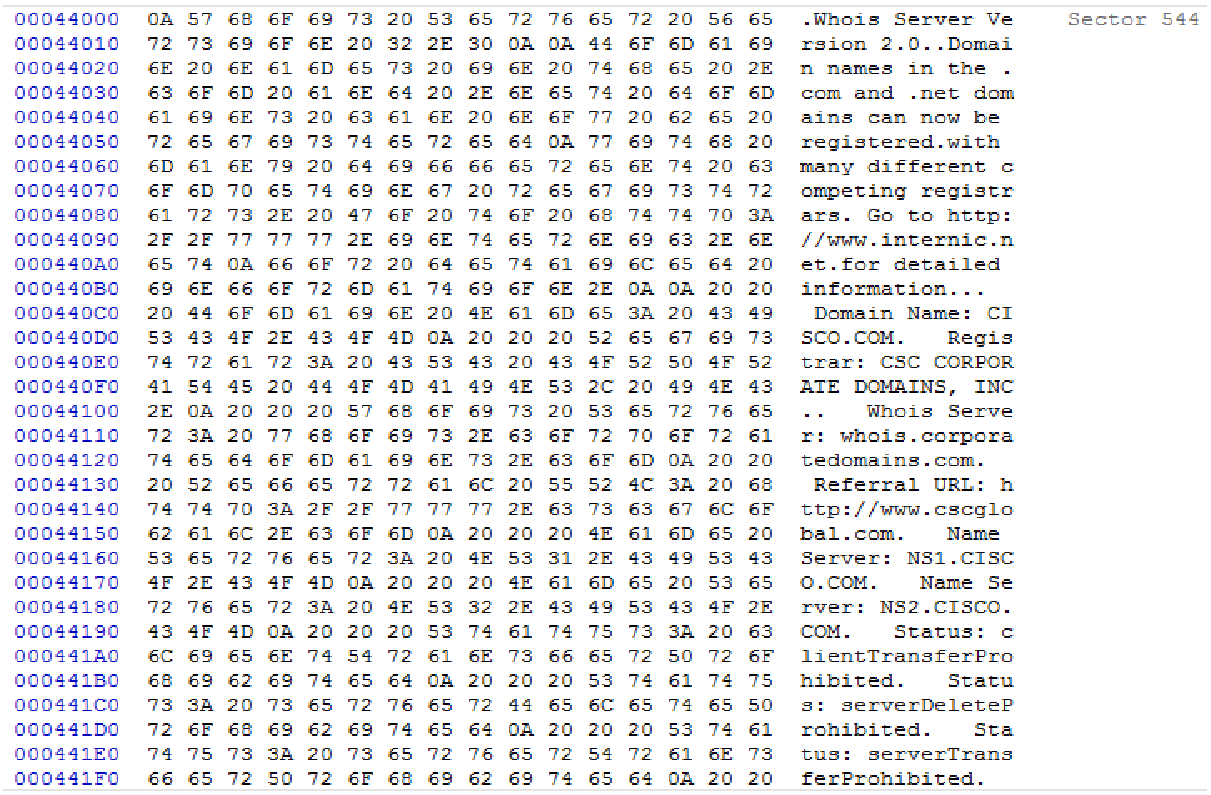
```
0003C000  E5 4E 00 6F 00 74 00 65 00 2E 00 0F 00 34 74 00  åN.o.t.e.....4t.   Sector 480
0003C010  78 00 74 00 00 00 FF FF FF FF 00 00 FF FF FF FF  x.t...ÿÿÿÿ..ÿÿÿÿ
0003C020  E5 4F 54 45 20 20 20 20 54 58 54 20 00 A6 9B A2  åOTE    TXT .¦›¢
0003C030  82 45 82 45 00 00 8E 5E 67 45 02 00 3A 00 00 00  ‚E‚E..Ž^gE..:...
0003C040  41 4E 49 4B 45 54 20 20 54 58 54 20 18 B4 9B A2  ANIKET  TXT .´›¢
0003C050  82 45 88 45 00 00 12 7E 88 45 03 00 86 06 00 00  ‚EˆE...~ˆE..†...
0003C060  41 4E 00 6F 00 74 00 65 00 70 00 0F 00 00 61 00  AN.o.t.e.p....a.
0003C070  64 00 2E 00 74 00 78 00 74 00 00 00 00 00 FF FF  d...t.x.t.....ÿÿ
0003C080  4E 4F 54 45 50 41 44 20 54 58 54 20 00 10 77 77  NOTEPAD TXT ..ww
0003C090  85 45 88 45 00 00 0C 84 88 45 04 00 5E 00 00 00  …EˆE...„ˆE..^...
0003C0A0  E5 55 52 44 55 45 20 20 54 58 54 20 18 09 92 78  åURDUE  TXT ..’x
0003C0B0  85 45 88 45 00 00 AF B4 2E 45 05 00 D4 03 00 00  …EˆE..¯´.E..Ô...
0003C0C0  41 4C 00 69 00 6E 00 6B 00 73 00 0F 00 40 2E 00  AL.i.n.k.s...@..
0003C0D0  74 00 78 00 74 00 00 00 FF FF 00 00 FF FF FF FF  t.x.t...ÿÿ..ÿÿÿÿ
0003C0E0  4C 49 4E 4B 53 20 20 20 54 58 54 20 00 6B 25 79  LINKS   TXT .k%y
0003C0F0  85 45 88 45 00 00 94 85 7A 45 06 00 81 00 00 00  …EˆE..”…zE......
0003C100  30 30 52 45 41 44 4D 45 54 58 54 20 18 C0 2B 79  00READMETXT .À+y
0003C110  85 45 88 45 00 00 80 9E 7A 45 07 00 A4 26 00 00  …EˆE..€žzE..¤&..
0003C120  41 4E 00 6F 00 74 00 65 00 2E 00 0F 00 34 74 00  AN.o.t.e.....4t.
0003C130  78 00 74 00 00 00 FF FF FF FF 00 00 FF FF FF FF  x.t...ÿÿÿÿ..ÿÿÿÿ
0003C140  4E 4F 54 45 20 20 20 20 54 58 54 20 00 73 E7 7B  NOTE    TXT .sç{
0003C150  88 45 88 45 00 00 8E 5E 67 45 02 00 3A 00 00 00  ˆEˆE..Ž^gE..:...
0003C160  50 55 52 44 55 45 20 20 54 58 54 20 18 09 92 78  PURDUE  TXT ..’x
0003C170  85 45 88 45 00 00 0F 84 88 45 05 00 D5 03 00 00  …EˆE...„ˆE..Õ...
0003C180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figure 3.17 Directory entry structure

Data Region:

The data region starts from 0x00040000 for this SD Card.

The contents of the files can be seen on sectors starting from the above address. Below is a snapshot of the file “Aniket.txt” in HxD editor:



```
00044000  0A 57 68 6F 69 73 20 53 65 72 76 65 72 20 56 65  .Whois Server Ve   Sector 544
00044010  72 73 69 6F 6E 20 32 2E 30 0A 0A 44 6F 6D 61 69  rsion 2.0..Domai
00044020  6E 20 6E 61 6D 65 73 20 69 6E 20 74 68 65 20 2E  n names in the .
00044030  63 6F 6D 20 61 6E 64 20 2E 6E 65 74 20 64 6F 6D  com and .net dom
00044040  61 69 6E 73 20 63 61 6E 20 6E 6F 77 20 62 65 20  ains can now be
00044050  72 65 67 69 73 74 65 72 65 64 0A 77 69 74 68 20  registered.with
00044060  6D 61 6E 79 20 64 69 66 66 65 72 65 6E 74 20 63  many different c
00044070  6F 6D 70 65 74 69 6E 67 20 72 65 67 69 73 74 72  ompeting registr
00044080  61 72 73 2E 20 47 6F 20 74 6F 20 68 74 74 70 3A  ars. Go to http:
00044090  2F 2F 77 77 77 2E 69 6E 74 65 72 6E 69 63 2E 6E  //www.internic.n
000440A0  65 74 0A 66 6F 72 20 64 65 74 61 69 6C 65 64 20  et.for detailed
000440B0  69 6E 66 6F 72 6D 61 74 69 6F 6E 2E 0A 0A 20 20  information...
000440C0  20 44 6F 6D 61 69 6E 20 4E 61 6D 65 3A 20 43 49   Domain Name: CI
000440D0  53 43 4F 2E 43 4F 4D 0A 20 20 20 52 65 67 69 73  SCO.COM.   Regis
000440E0  74 72 61 72 3A 20 43 53 43 20 43 4F 52 50 4F 52  trar: CSC CORPOR
000440F0  41 54 45 20 44 4F 4D 41 49 4E 53 2C 20 49 4E 43  ATE DOMAINS, INC
00044100  2E 0A 20 20 20 57 68 6F 69 73 20 53 65 72 76 65  ..   Whois Serve
00044110  72 3A 20 77 68 6F 69 73 2E 63 6F 72 70 6F 72 61  r: whois.corpora
00044120  74 65 64 6F 6D 61 69 6E 73 2E 63 6F 6D 0A 20 20  tedomains.com.
00044130  20 52 65 66 65 72 72 61 6C 20 55 52 4C 3A 20 68   Referral URL: h
00044140  74 74 70 3A 2F 2F 77 77 77 2E 63 73 63 67 6C 6F  ttp://www.cscglo
00044150  62 61 6C 2E 63 6F 6D 0A 20 20 20 4E 61 6D 65 20  bal.com.   Name
00044160  53 65 72 76 65 72 3A 20 4E 53 31 2E 43 49 53 43  Server: NS1.CISC
00044170  4F 2E 43 4F 4D 0A 20 20 20 4E 61 6D 65 20 53 65  O.COM.   Name Se
00044180  72 76 65 72 3A 20 4E 53 32 2E 43 49 53 43 4F 2E  rver: NS2.CISCO.
00044190  43 4F 4D 0A 20 20 20 53 74 61 74 75 73 3A 20 63  COM.   Status: c
000441A0  6C 69 65 6E 74 54 72 61 6E 73 66 65 72 50 72 6F  lientTransferPro
000441B0  68 69 62 69 74 65 64 0A 20 20 20 53 74 61 74 75  hibited.   Statu
000441C0  73 3A 20 73 65 72 76 65 72 44 65 6C 65 74 65 50  s: serverDeleteP
000441D0  72 6F 68 69 62 69 74 65 64 0A 20 20 20 53 74 61  rohibited.   Sta
000441E0  74 75 73 3A 20 73 65 72 76 65 72 54 72 61 6E 73  tus: serverTrans
000441F0  66 65 72 50 72 6F 68 69 62 69 74 65 64 0A 20 20  ferProhibited.
```

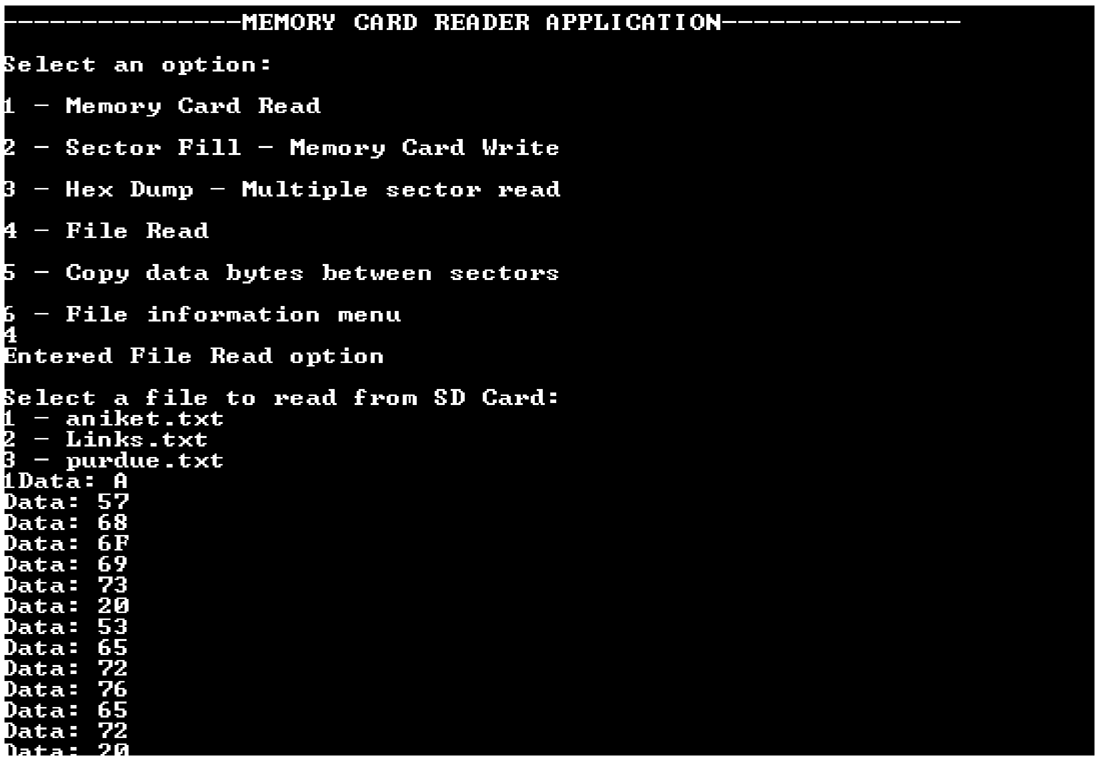
Figure 3.18 File contents – aniket.txt

Implementation:

The file read functionality has been implemented in the sdc driver using application program. The **sd_single_read_glcd()** function implements file read.

The file list is shown on the terminal screen. Depending upon the file selected, the file read function is implemented which starts reading from the sector in which the file is stored and reads upto the length of the file is covered.

The hex contents of the file are shown on the terminal as below:



```
---------------MEMORY CARD READER APPLICATION---------------

Select an option:

1 - Memory Card Read

2 - Sector Fill - Memory Card Write

3 - Hex Dump - Multiple sector read

4 - File Read

5 - Copy data bytes between sectors

6 - File information menu
4
Entered File Read option

Select a file to read from SD Card:
1 - aniket.txt
2 - Links.txt
3 - purdue.txt
1Data: A
Data: 57
Data: 68
Data: 6F
Data: 69
Data: 73
Data: 20
Data: 53
Data: 65
Data: 72
Data: 76
Data: 65
Data: 72
Data: 20
```

Figure 3.19 File Read feature

Display on graphic LCD:

The unique feature of this program is that the file contents are displayed on the graphic LCD. The graphic LCD driver writes file data to the LCD screen. After displaying one page on the LCD screen, the screen freezes until the user signifies to display the next page. This is done by providing an interrupt at INT0.

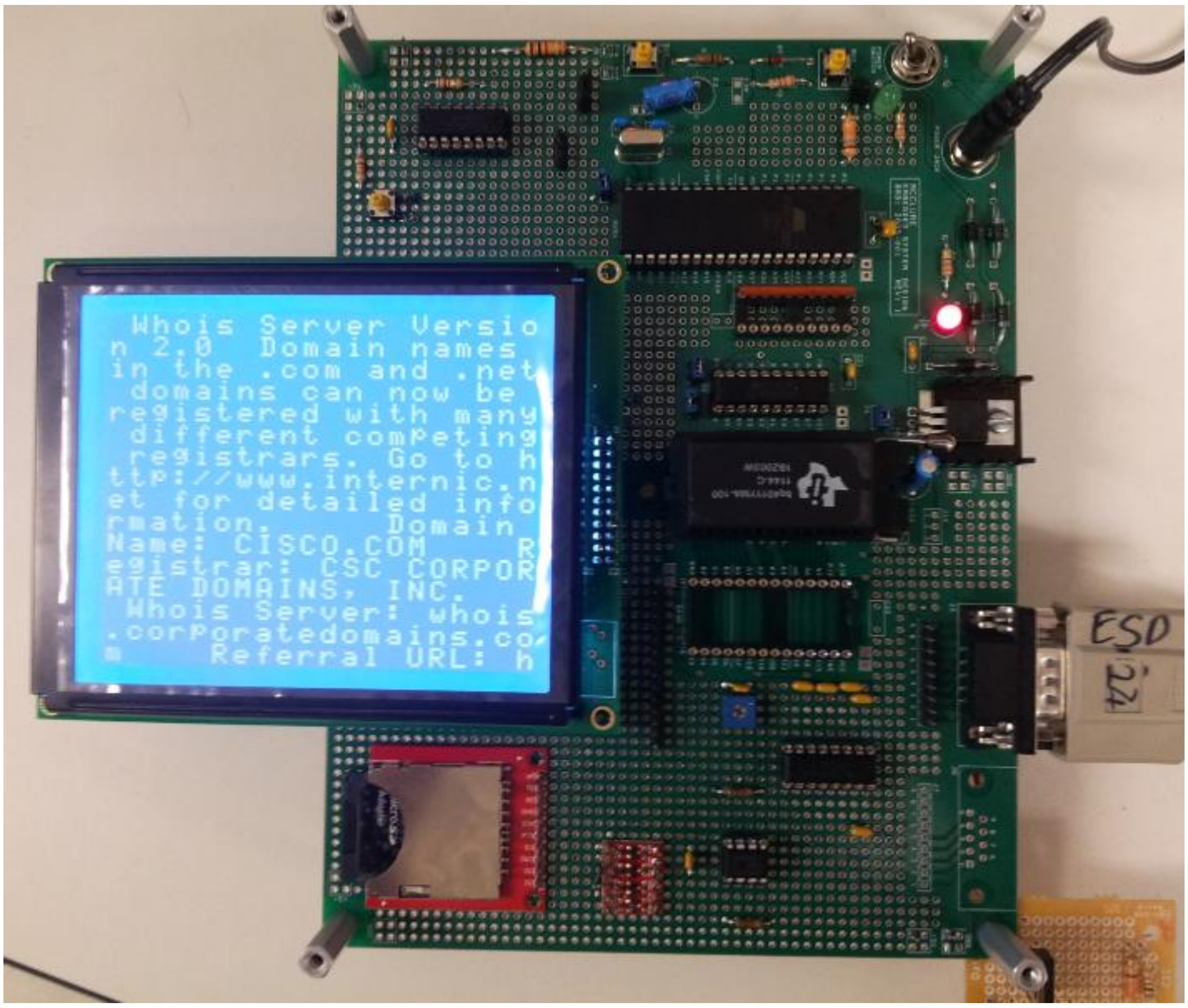

```
 Whois Server Versio
n 2.0  Domain names
in the .com and .net
 domains can now be
registered with many
 different competing
 registrars. Go to h
ttp://www.internic.n
et for detailed info
rmation.    Domain
Name: CISCO.COM   R
egistrar: CSC CORPOR
ATE DOMAINS, INC.
 Whois Server: whois
.corporatedomains.co
m     Referral URL: h
```

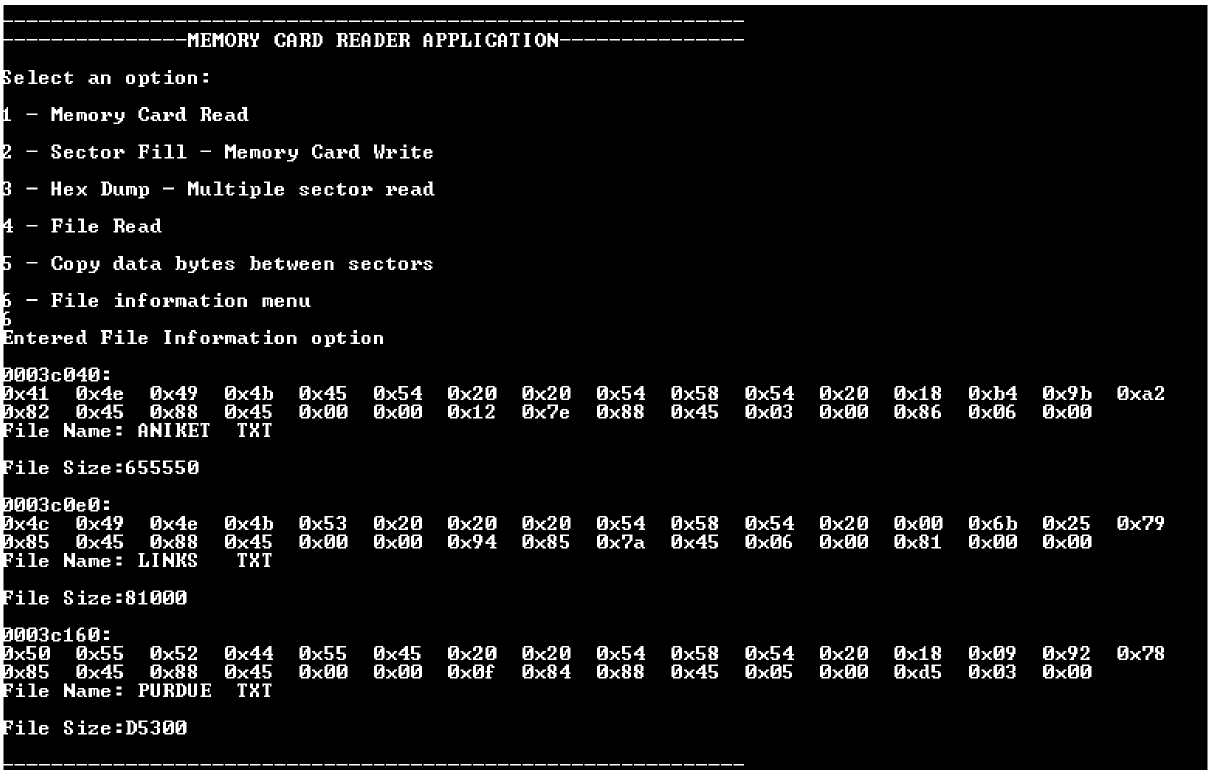
Figure 3.20 File display on Graphic LCD

File information:

The file information feature has been implemented to show the file name and size read from the directory entry structure sector 0x0003C000.

The last four bytes contain the file size and the first 11 bytes contain the filename and file extension.

The **fileopen()** and **getfilename()** functions implements this functionality. The result of the file information menu is shown on the terminal as below:



```
------------------------------------------------------------
---------------MEMORY CARD READER APPLICATION---------------

Select an option:

1 - Memory Card Read

2 - Sector Fill - Memory Card Write

3 - Hex Dump - Multiple sector read

4 - File Read

5 - Copy data bytes between sectors

6 - File information menu
6
Entered File Information option

0003c040:
0x41  0x4e  0x49  0x4b  0x45  0x54  0x20  0x20  0x54  0x58  0x54  0x20  0x18  0xb4  0x9b  0xa2
0x82  0x45  0x88  0x45  0x00  0x00  0x12  0x7e  0x88  0x45  0x03  0x00  0x86  0x06  0x00
File Name: ANIKET  TXT

File Size:655550

0003c0e0:
0x4c  0x49  0x4e  0x4b  0x53  0x20  0x20  0x20  0x54  0x58  0x54  0x20  0x00  0x6b  0x25  0x79
0x85  0x45  0x88  0x45  0x00  0x00  0x94  0x85  0x7a  0x45  0x06  0x00  0x81  0x00  0x00
File Name: LINKS   TXT

File Size:81000

0003c160:
0x50  0x55  0x52  0x44  0x55  0x45  0x20  0x20  0x54  0x58  0x54  0x20  0x18  0x09  0x92  0x78
0x85  0x45  0x88  0x45  0x00  0x00  0x0f  0x84  0x88  0x45  0x05  0x00  0xd5  0x03  0x00
File Name: PURDUE  TXT

File Size:D5300

------------------------------------------------------------
```

Figure 3.21 File information menu

The hex dump shows the file information in the directory entry structure sector.

3.2.4 Graphic LCD driver:

A graphic LCD with T6963C controller has been interfaced to the microcontroller. The main objective of this graphic LCD was to sow the file contents on the LCD screen. This provides a file read feature which can display files selected by the used on the LCD screen. The LCD is 160 X 128 pixels wide.

The graphic LCD driver has four main functions:

- LCD initialization
- Put charater
- Put string
- Goto cursor position

Intilization - glcd_init() :

The initialization function of graphic LCD sends a reset command to the LCD initially. It then gives certain commands to the LCD like set text area, set text home address, set text attribute mode.

The commands to the LCD can be single byte or two byte commands.

The clear screen function **clrscr()** clears the LCD screen.

Put character – glcd_putch():

This function writes the character passed to it on the LCD. The cursor position needs to be set where the character needs to be written. An auto increment feature increments the cursor to the next position automatically.

Put string – glcd_putstr():

The put string function calls the put char function till the string ends i.e. a NULL character is encountered.

Set cursor position - glcd_gotoxy():

This function sets the desired cursor position on the LCD screen.

The LCD interface with the host MCU is a parallel one. The control signals of the LCD are connected to the port 2 pins and are controlled by the microcontroller.

Back light is provided from a separate LM7805 so that power drawn from primary 7805 is minimized.

3.2.5 Serial driver:

Serial communication through RS232 port with the computer terminal has been used as a user interface in this project. The serial communication driver has been leveraged from my own code written in previous labs of embedded system design.

The serial port initialization function **serinit()** initializes the serial port at the 9600 baud rate. The **putstr(), putchar()** and **getchar()** functions have been used to get characters or print characters to the terminal.

3.2.6 Application program

The application program provides the user with features of memory read, sector fill – memory write, multiple sector read, hex dump, file read, memory copy and file information.

The user interface is coded in the application program.

All the driver functions are called from the main program and the above features are implemented.

Complete project set up:

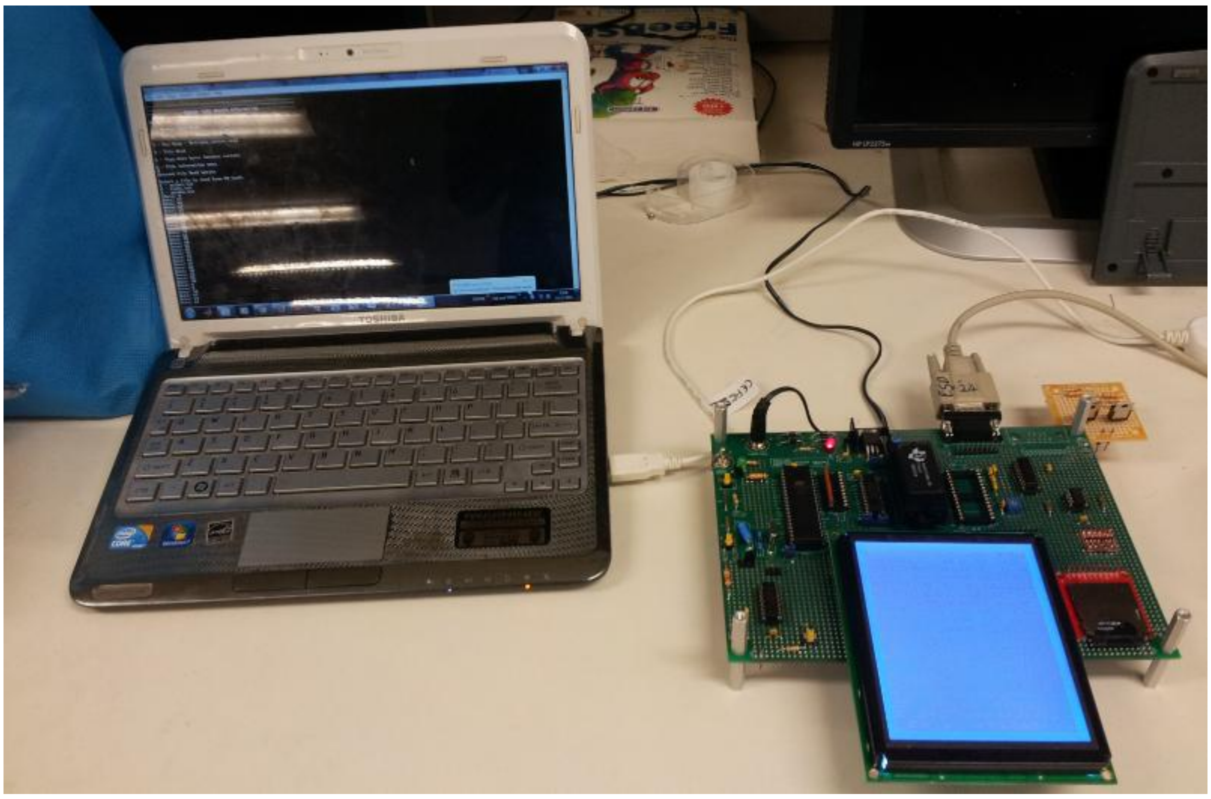


Figure 3.22 Complete set up

4. Conclusion

The objective of this project was to implement a memory card reader on 8051 application board. This objective was achieved successfully with the desired features for the card reader application. The graphic LCD interfaced to the board provided a complete interface for the user to view the contents of the files.

Reading from the memory card was a big challenge. Debugging problems in this project was difficult. The responses of the SD card were viewed on a logic analyzer. This gave a good idea if the SD card was accepting commands.

The goal of writing a portable sd card driver was achieved through this project. By the end of this project, I learnt significant things about the FAT file system and the SD card commands as a whole. The graphic LCD interface was a new experience.

5. Future development ideas

Implementing file write on the 8051 board would be a positive step ahead. This would provide a complete file system interface to the user.

To implement a file write driver, the directory structure sector needs to be edited to include the file structure information of the newly written file.

With file write, file erase functionality can also be implemented.

The memory card can be used as data memory storage for applications that acquire high data and needs to be stored for analysis.

6. References

1] SanDisk SD card – Product Manual Version 2.2 Document No. 80-13-00169

2] SD specifications – Part A2 – SD Host Controller

3] Specifications – FAT16 file system

http://www.maverick-os.dk/FileSystemFormats/FAT16_FileSystem.html#StatingCluster

4] SD Card Map

5] Specifications for LCD Module - DS-G160128STBWW

6] T6963C Graphic LCD Controller datasheet

7] Writing Software for T6963C based Graphic LCDs

8] Application Note AN-029 Interfacing and set-up of Toshiba T6963C

Note: The files used for references have been provided as a part of submission in the section "Reference Material"

7. Appendices

7.1 Appendix A – Bill of Materials

Part	Cost
Memory card module	$9
Level converter	$3
Graphic LCD	$50
Header strips	$3
Total	**$65**

7.2 Appendix B – Source Code

7.2.1 Application program

```
/*
Program: Memory card reader 8051

Description: Memory Card reader application on 8051.
The memory card reader application provides sector read, write, multiple sector read
copy byte and file information functionality. This program is the application program
which runs over the sd card driver. It uses serial communication, SPI interface for
SD Card and graphic LCD display libraries to implement the functionality.

Author: Aniket Kumar

Date: 11/26/2014
*/

#include <mcs51/8051.h>
#include <at89c51ed2.h> //also includes 8052.h and 8051.h
#include <stdio.h>
#include <stdlib.h>
#include <mcs51/8052.h>
#include <string.h>
#include "serialinit.h" // Serial initialization routines
#include "sdcdriver.h" // SD card driver
#include "integer.h" // Data types
#include "glcddriver.h" // Graphic LCD driver

#define WR P2_0
#define CE P2_2
#define HALT P2_5

#define ADDR1 0x00044000
#define ADDR2 0x00050000
#define ADDR3 0x0004C000

__sfr __at (0xC3) SPCON ;
__sfr __at (0xC4) SPSTA ;
__sfr __at (0xC5) SPDAT ;

_sdcc_external_startup()
{
AUXR &= 0xFD;
AUXR |= 0xC0;
return 1;
}

void int_SPI(void) __interrupt (9);
void spi_init();
char fetch_data();

typedef union
{
BYTE b[4];
ULONG ul;
} b_ul;

extern char serial_data;
extern xdata BYTE *databuff;
extern bit transmit_completed;

void main()
{
int pg,k;
```

```
char op,databyte,gch;
xdata b_ul ch,ch1,ch2,ch3,ch4,ch5;
ULONG sec_no1,sec_no2,sec_no3;


spi_init(); // Initializes SPI registers in 8051
serinit(); // Initializes 8051 serial port
glcd_init(); // Initializes graphic lcd

EA=1; // Enable global interrupt bit in IE
EX0=1; // Enable INT0 interrupt
IE0=0;
sd_init(); // initialize SD Card

P2=0xFF;
CE=1;WR=1;

while(1)
{

printf_tiny("\n\r------------------------------------------------------------\n\r");
printf_tiny("------------------------------------------------------------\n\r");
printf_tiny("---------------MEMORY CARD READER APPLICATION---------------\n\r");
printf_tiny("\n\rSelect an option:\n\r");
printf_tiny("\n\r1 - Memory Card Read\n\r");
printf_tiny("\n\r2 - Sector Fill - Memory Card Write\n\r");
printf_tiny("\n\r3 - Hex Dump - Multiple sector read\n\r");
printf_tiny("\n\r4 - File Read\n\r");
printf_tiny("\n\r5 - Copy data bytes between sectors\n\r");
printf_tiny("\n\r6 - File information menu\n\r");

// Get input from user
op = getchar();

switch(op)
{

case '1': // Memory read option
 {
 printf_tiny("\n\rEntered Memory Read option\n\r");
 printf_tiny("\n\rEnter sector address (4 bytes) to read from: \n\r");
 ch.b[3] = fetch_data();
 ch.b[2] = fetch_data();
 ch.b[1] = fetch_data();
 ch.b[0] = fetch_data();

 sd_single_read(ch.ul); // Call single sector read function

 break;
 }
 case '2': //Memory write - sector fill
 {
 printf_tiny("\n\rEntered Sector Fill \n\r");
 printf_tiny("\n\rEnter sector address to be written to memory card: \n\r");
 ch1.b[3] = fetch_data();
 ch1.b[2] = fetch_data();
 ch1.b[1] = fetch_data();
 ch1.b[0] = fetch_data();

 printf_tiny("\n\rEnter data to be written to memory card: \n\r");
 databyte = getchar(); //get databyte from user for sector fill
 printf_tiny("\n\rWriting Data....");
 sd_single_write(databyte,ch1.ul); // SD single sector write
 printf_tiny("\n\rWrite complete....");
 break;
 }
 case '3': // Multiple sector read
 {
 printf_tiny("\n\rHex dump - Multiple sector read option\n\r");
 printf_tiny("\n\rEnter start sector address\n\r");
 ch2.b[3] = fetch_data();
 ch2.b[2] = fetch_data();
```

```
ch2.b[1] = fetch_data();
ch2.b[0] = fetch_data();

printf_tiny("\n\rEnter end sector address\n\r");
ch3.b[3] = fetch_data();
ch3.b[2] = fetch_data();
ch3.b[1] = fetch_data();
ch3.b[0] = fetch_data();

sd_multiple_read(ch2.ul,ch3.ul); // multiple sector read
break;
}

case '5': //Memory Copy from one sector to another
{
printf_tiny("\n\rEntered Memory copy option\n\r");
printf_tiny("\n\rEnter source sector address:\n\r");
ch4.b[3] = fetch_data();
ch4.b[2] = fetch_data();
ch4.b[1] = fetch_data();
ch4.b[0] = fetch_data();

printf_tiny("\n\rEnter destination sector address:\n\r");
ch5.b[3] = fetch_data();
ch5.b[2] = fetch_data();
ch5.b[1] = fetch_data();
ch5.b[0] = fetch_data();

printf_tiny("\n\rCopying data...:\n\r");
sd_copy(ch4.ul,ch5.ul); // Call copy function from sdc driver
break;

}
case '4': //File read option
{
printf_tiny("\n\rEntered File Read option\n\r");
printf_tiny("\n\rSelect a file to read from SD Card:\n\r");
printf_tiny("1 - aniket.txt\n\r");
printf_tiny("2 - Links.txt\n\r");
printf_tiny("3 - purdue.txt\n\r");


gch = getchar();

switch(gch)
{
case '1': // Read file 1
sec_no1=ADDR1; // Address of file 1
for(pg=0;pg<5;pg++)
{
sd_single_read_glcd(sec_no1+(pg*0x140)); // single sector read
while(IE0 != 1); // wait for INT0
// IF INT0=0 goto next pag
glcd_init(); // clear LCD
}
break;
case '2': // Read file 2
sec_no2=ADDR2; // Address of file 2
for(pg=0;pg<1;pg++)
{
sd_single_read_glcd(sec_no2+(pg*0x140)); // read file
while(IE0 != 1); // wait for INT0
// IF INT0=0 goto next page
glcd_init(); // clear LCD
}
break;
case '3': // Read file 3
sec_no3=ADDR3; // Address of file 3
for(pg=0;pg<3;pg++)
{
sd_single_read_glcd(sec_no3+(pg*0x140));
while(IE0 != 1); // wait for INT0
// IF INT0=0 goto next page
glcd_init(); // clear LCD
}

break;
}
break;

}

case '6': // File information menu
{
printf_tiny("\n\rEntered File Information option\n\r");
for(k=0;k<3;k++)
fileopen(k+1); // Read file information from sector 0x0003C000

break;
}

}

}
}

// SPI interrupt service routine
void int_SPI(void) __interrupt (9) /* interrupt address is 0x004B */
{
switch (SPSTA) /* read and clear spi status register */
{
case 0x80:
serial_data=SPDAT; /* read receive data */
transmit_completed=1;/* set software flag */
delay(100);
break;

case 0x10:
/* put here for mode fault tasking */
printf_tiny("Mode fault\n\r");
break;

case 0x40:
/* put here for overrun tasking */
printf_tiny("Overrun tasking\n\r");
break;
}
}

// Initialize SPI module of 805
void spi_init()
{
SPCON &= 0x00; /* Fclk Periph/2 */
SPCON |= 0x10; /* Master mode */
IEN1 |= 0x04; /* enable spi interrupt */
}

// Fetch single byte data from terminal in hex format
char fetch_data()
{
int i1,i;
char ch;
char finalval;
char bufsiz1[2];

do
{
memset(bufsiz1,'\0',2);

for(i1=0;i1<2;i1++)
{
ch = getchar();
if(((ch > 47) && (ch < 59)) || ((ch > 96) && (ch < 103)) || ((ch > 64) && (ch
< 71)))
```

```c
{
bufsiz1[i1] = ch;
}
else if(!(((ch > 47) && (ch < 59)) || ((ch > 96) && (ch < 103)) || ((ch > 64) && (ch < 71))))
{
printf_tiny("\n\rPlease enter hexadecimal input. Special characters are not allowed.\n\r");
break;
}

}
}while(i1<2);

finalval=0;

// Convert ascii string to hex number
for(i=0;i<2;i++)
{

if((bufsiz1[i] >= '0') && (bufsiz1[i] <= '9'))
{
bufsiz1[i] -= 48;
if(i==0)
{
finalval += (bufsiz1[i]*16);
}
else if(i==1)
{
finalval += bufsiz1[i];
}

}


else if((bufsiz1[i] >= 'A') && (bufsiz1[i] <= 'F'))
{
switch(bufsiz1[i])
{
case 'A':
if(i==0)
finalval += (10*16);
else if(i==1)
finalval += 10;
break;
case 'B':
if(i==0)
finalval += (11*16);
else if(i==1)
finalval += 11;
break;
case 'C':
if(i==0)
finalval += (12*16);
else if(i==1)
finalval += 12;
break;
case 'D':
if(i==0)
finalval += (13*16);
else if(i==1)
finalval += 13;
break;
case 'E':
if(i==0)
finalval += (14*16);
else if(i==1)
finalval += 14;
break;
case 'F':
if(i==0)
finalval += (15*16);
else if(i==1)
finalval += 15;
break;
}
}

else if((bufsiz1[i] >= 'a') && (bufsiz1[i] <= 'f'))
{
switch(bufsiz1[i])
{
case 'a':
if(i==0)
finalval += (10*16);
else if(i==1)
finalval += 10;
break;
case 'b':
if(i==0)
finalval += (11*16);
else if(i==1)
finalval += 11;
break;
case 'c':
if(i==0)
finalval += (12*16);
else if(i==1)
finalval += 12;
break;
case 'd':
if(i==0)
finalval += (13*16);
else if(i==1)
finalval += 13;
break;
case 'e':
if(i==0)
finalval += (14*16);
else if(i==1)
finalval += 14;
break;
case 'f':
if(i==0)
finalval += (15*16);
else if(i==1)
finalval += 15;
break;
}

}

}
return finalval; // return databyte
}

```

7.2.2 SD Card driver

```c
/*
Library: SD Card functionality drivers

Description: SD card drivers for initialization, sector read, sector write,
multiple sector read, hex dump, copy byte have been coded
in this library.

Author: Aniket Kumar

Date: 11/26/2014
*/

#include "sdcdriver.h"
#include "glcddriver.h"

#define CE P2_2
#define WR P2_0
#define HALT P2_5

xdata char serial_data;
xdata char data_save;
bit transmit_completed;

BYTE *filenm,*filesiz;

xdata BYTE *buff,*tempbuf;

typedef union //Union to send address bytes one by one
{
BYTE b[4];
ULONG ul;
} b_ul;

b_ul fsz;

// File information retrived using fileopen() function
void fileopen(int fnum)
{
int i1,i2;

getfilename(fnum); // Retrieve file name and file size

printf_tiny("\n\rFile Name: ");

for(i1=0;i1<11;i1++)
{
printf("%c",filenm[i1]); // Print file name
}

printf_tiny("\n\r");
printf("\n\rFile Size:");
for(i2=0;i2<4;i2++)
{
printf_tiny("%x",(filesiz[i2] & 0xff)); // Print file size
}

printf_tiny("\n\r");

}

// Read file from a specific location
void fileread(int fl)
{
int i;
ULONG filesector,filesector_start;

filesector_start = 0x00040000;

filesector = filesector_start + (fl*0x400);
```

```c

sd_single_read(filesector); // Call single sector read dependig upon file number

printf("\n\rData received: \n\r");

for(i=0;i<320;i++)
printf("%c",buff[i]);
}

// Get the name of file from FAT
void getfilename(int ctrl)
{
int i;
ULONG file_sector_start,file_sector_end;

switch(ctrl)
{
case 1: // File name 1
file_sector_start = 0x0003C040;
file_sector_end = 0x0003C05F;
sd_multiple_read(file_sector_start,file_sector_end);
break;
case 2: // File name 2
file_sector_start = 0x0003C0E0;
file_sector_end = 0x0003C0FF;
sd_multiple_read(file_sector_start,file_sector_end);
break;
case 3: // File name 3
file_sector_start = 0x0003C160;
file_sector_end = 0x0003C17F;
sd_multiple_read(file_sector_start,file_sector_end);
break;
}


for(i=0;i<11;i++)
{
filenm[i] = buff[i];
}

for(i=0;i<4;i++)
{
filesiz[i] = buff[i+28];
}
}

// Copy data from one sector to another
void sd_copy(ULONG srcsect,ULONG destsect)
{

sd_single_read(srcsect); // read single sector
memcpy(tempbuf,buff,512); // Copy data to temporary buffer
sd_block_write(destsect); // Write data to destination sector

}

// Initialize SD Card
int sd_init()
{
int i,counter;
SPCON |= 0x40; /* Run SPI */

for(i=0;i<8;i++)
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
}

P1_4=0; // Make slave select SS low

for(i=0;i<2;i++)
{
```

```
wait_tx_end(); // Dummy byte & Wait for end of transmission
}

wait_tx_end(); // Send Dummy byte & Wait for end of transmission
P1_4=0;
wait_tx_end(); // Dummy byte & Wait for end of transmission

// CMD0 Idle Mode
SPDAT = 0x40;
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */

counter = 0;

while(counter <= 3)
{
SPDAT = 0x00;
counter++;
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */
}

SPDAT = 0x95;
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */


wait_tx_end(); // Dummy byte & Wait for end of transmission

wait_tx_end(); // Dummy byte & Wait for end of transmission

//CMD 1
while(data_save != 0x00)
{
SPDAT = 0x41; // Start byte
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */

counter = 0; // Argument 4 bytes

while(counter <= 3)
{
SPDAT = 0x00;
counter++;
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */
}
for(i=0;i<3;i++)
wait_tx_end(); // Dummy byte & Wait for end of transmission

delay(5);
}


//CMD58 Read OCR
SPDAT = 0x7A; // Start byte
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */

counter = 0; // Argument 4 bytes

while(counter <= 3)
{
SPDAT = 0x00;
counter++;
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */
}

wait_tx_end(); // Dummy byte & Wait for end of transmission

for(i=0;i<6;i++)
```

```
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
data_save = serial_data;
}


//CMD 13
SPDAT = 0x4D; // Start byte
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */

counter = 0; // Argument 4 bytes

while(counter <= 3)
{
SPDAT = 0x00;
counter++;
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */
}

for(i=0;i<4;i++)
wait_tx_end(); // Dummy byte & Wait for end of transmission
data_save = serial_data;

return 0;
}

// Single byte read function
BYTE sd_single_read_glcd(ULONG sectnum)
{
int countera,ii;
char save_data1,cli;
b_ul temp2;

temp2.ul = sectnum;

//CMD 17 Single Data block read

wait_tx_end(); // Dummy byte & Wait for end of transmission

SPDAT = 0x51; // Start byte
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */

countera = 3; // Argument 4 bytes
while(countera >=0)
{
SPDAT = temp2.b[countera];
countera--;
while(!transmit_completed);
transmit_completed = 0;
}


wait_tx_end(); // Dummy byte & Wait for end of transmission
wait_tx_end(); // Dummy byte & Wait for end of transmission


for(ii=0;ii<4;ii++)
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
save_data1 = serial_data;
}


for(ii=0;ii<320;ii++)
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
save_data1 = SPDAT;
printf_tiny("Data: %x\n\r",(save_data1 & 0xFF));
cli = glcd_format(SPDAT);
```

```
glcd_putch(cli);
}

for(ii=0;ii<4;ii++)
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
save_data1 = serial_data;
}
return 1;
}

// Single sector read function
BYTE sd_single_read(ULONG sector)
{
int counter ,i,ctr0,ctr1,ctr2,ctr3;

char save_data;
b_ul temp1;

temp1.ul = sector;

//CMD 17 Single Data block read

wait_tx_end(); // Dummy byte & Wait for end of transmission
SPDAT = 0x51; // Start byte
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */

counter = 3; // Argument 4 bytes
while(counter >=0)
{
SPDAT = temp1.b[counter ];
counter --;
while(!transmit_completed);
transmit_completed = 0;
}

for(i=0;i<6;i++)
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
save_data = serial_data;
}

ctr0=(temp1.b[1] & 0xff)-1;
ctr1= (temp1.b[1] & 0xff);
ctr2= (temp1.b[2] & 0xff);
ctr3= (temp1.b[3] & 0xff);

// Print address
printf("\n\r%02x%02x%02x%02x: \n\r",(temp1.b[3] & 0xff),(temp1.b[2] &
0xff),(temp1.b[
1] & 0xff),(temp1.b[0] & 0xff));

for(i=0;i<512;i++)
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
save_data = SPDAT;

delay(5);

buff[i] = save_data;

ctr0++;
if(i == 256)
{
ctr1++;
ctr0=0;
}
if(((i%16)==0) && (i!=0))
{
putchar('\n');
putchar('\r');
```

```c

printf("0x%02x ",SPDAT);
}
else
{
printf("0x%02x ",SPDAT);
}

}

for(i=0;i<4;i++)
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
save_data = serial_data;
}
return 1;
}

// Single sector write function
BYTE sd_single_write(BYTE databyte,ULONG sector)
{
int i2,counterw;
b_ul tempw;
char data2;

tempw.ul = sector;

//CMD 24 Single Data block write
wait_tx_end(); // Dummy byte & Wait for end of transmission

SPDAT = 0x58; // Start byte
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */

counterw = 3; // Argument 4 bytes
while(counterw >= 0)
{
SPDAT = tempw.b[counterw]; // CRC
counterw--;
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */
}

for(i2=0;i2<4;i2++)
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
data2 = serial_data;
}

// Write data byte

counterw = 0; // Reset byte counter;
// Write <current_blklen> bytes to MMC;
while(counterw < 512)
{
SPDAT = databyte; // Write data byte out through SPI;
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */
counterw++; // Increment byte counter;

}

wait_tx_end(); // Dummy byte & Wait for end of transmission

do // Read Data Response from card;
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
data2 = SPDAT;
printf_tiny("0xFF0\n\r");
} // When bit 0 of the MMC response
// is clear, a valid data response
// has been received;
```

```c
while((data2 & 0x01) != 1);

do // Wait for end of busy signal;
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
printf_tiny("0xFF1\n\r");
}
while(SPDAT == 0x00); // When a non-zero token is returned,
// card is no longer busy;

wait_tx_end(); // Dummy byte & Wait for end of transmission

return 1;
}

// Single block write function for copy data byte
BYTE sd_block_write(ULONG sector)
{
int i,counter1;
b_ul tempw1;
char data3;

tempw1.ul = sector;

//CMD 24 Single Data block write
wait_tx_end(); // Dummy byte & Wait for end of transmission

SPDAT = 0x58; // Start byte
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */

counter1 = 3; // Argument 4 bytes
while(counter1 >= 0)
{
SPDAT = tempw1.b[counter1]; // CRC
counter1--;
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */
}


for(i=0;i<4;i++)
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
data3 = serial_data;
}

// Write data byte

counter1 = 0; // Reset byte counter;
// Write <current_blklen> bytes to MMC;
while(counter1 < 512)
{
SPDAT = tempbuf[counter1]; // Write data byte out through SPI;
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */
counter1++; // Increment byte counter;

}

wait_tx_end(); // Dummy byte & Wait for end of transmission


do // Read Data Response from card;
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
data3 = SPDAT;
printf_tiny("0xFF0\n\r");
} // When bit 0 of the MMC response
// is clear, a valid data response
// has been received;
while((data3 & 0x01) != 1);
```

```

do // Wait for end of busy signal;
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
printf_tiny("0xFF1\n\r");
}
while(SPDAT == 0x00); // When a non-zero token is returned,
// card is no longer busy;

wait_tx_end(); // Dummy byte & Wait for end of transmission

return 1;
}

// Multiple sector read function
BYTE sd_multiple_read(ULONG start_sector,ULONG end_sector)
{
//CMD 16 Block length
int i,counter,ctr2,ctr3;
char data1;
xdata b_ul temp_start,temp_end;
ULONG bytetot;

temp_start.ul = start_sector;
temp_end.ul = end_sector;
bytetot = end_sector-start_sector;


SPDAT = 0x50; // Start byte
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */

counter = 0; // Argument 4 bytes

while(counter <= 3)
{
if(counter == 2)
{
SPDAT = 0x02;
counter++;
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */
}
else
{
SPDAT = 0x00;
counter++;
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */
}
}

for(i=0;i<3;i++)
wait_tx_end(); // Dummy byte & Wait for end of transmission
data1 = serial_data;


//CMD 18 Multiple Data block read

wait_tx_end(); // Dummy byte & Wait for end of transmission
SPDAT = 0x52; // Start byte
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */

counter = 3; // Argument 4 bytes
while(counter >= 0)
{
SPDAT = temp_start.b[counter]; // CRC
counter--;
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */
}
```

```
wait_tx_end(); // Dummy byte & Wait for end of transmission
wait_tx_end(); // Dummy byte & Wait for end of transmission


for(i=0;i<4;i++)
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
data1 = serial_data;
}

ctr2= (temp_start.b[2] & 0xff);
ctr3= (temp_start.b[3] & 0xff);

printf("\n\r%02x%02x%02x%02x: \n\r",(temp_start.b[3] & 0xff),(temp_start.b[2] & 0xff
),(temp_start.b[1] & 0xff),(temp_start.b[0] & 0xff));

for(i=0;i<bytetot;i++)
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
data1 = SPDAT;

delay(5);
buff[i] = data1;

if(((i%16)==0) && (i!=0))
{
putchar('\n');
putchar('\r');

printf("0x%02x ",SPDAT);
}
else
{
printf("0x%02x ",SPDAT);
}


}

sd_stop_transmission(); // Send stop trnsmission command

for(i=0;i<4;i++)
{
wait_tx_end(); // Dummy byte & Wait for end of transmission
data1 = serial_data;
}

return 1;
}

// Stop transmission command
void sd_stop_transmission()
{
int i,counter;

SPDAT = 0x4C; // Start byte
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */

counter = 0; // Argument 4 bytes

while(counter <= 3)
{
SPDAT = 0x00;
counter++;
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */
}

for(i=0;i<3;i++)
wait_tx_end(); // Dummy byte & Wait for end of transmission
data_save = serial_data;
```

```c
}

// Send dummy byte and wait for end of transmission
// Useful when reading a byte or eaiting for response
void wait_tx_end()
{
SPDAT = 0xFF; // Wait for response
while(!transmit_completed);/* wait for end of transmition */
transmit_completed = 0; /* clear software transfer flag */
}

// Arbitrary delay
void delay(unsigned int number)
{
unsigned int i,j;
for(i=0;i<number;i++)
{
for(j=0;j<100;j++)
{
}
}
return;
}
```

7.2.3 SD card driver header file

```c
/*
Header File: SD Card functionality drivers header file

Description: SD card drivers for initialization, sector read, sector write,
multiple sector read, hex dump, copy byte have been coded
in this library.

Author: Aniket Kumar

Date: 11/26/2014
*/

#include <mcs51/8051.h>
#include <at89c51ed2.h> //also includes 8052.h and 8051.h
#include <stdio.h>
#include <stdlib.h>
#include <mcs51/8052.h>
#include <string.h>
#include <malloc.h>
#include "serialinit.h"
#include "integer.h"

#ifndef SDCDRIVER_H_INCLUDED
#define SDCDRIVER_H_INCLUDED

int sd_init();

BYTE sd_single_read(ULONG sector);
BYTE sd_single_read_glcd(ULONG sectnum);
BYTE sd_multiple_read(ULONG start_sector,ULONG end_sector);

BYTE sd_single_write(BYTE databyte,ULONG sector);
BYTE sd_block_write(ULONG sector);

void sd_stop_transmission ();
void getfilename(int ctr1);
void fileopen(int fnum);
void fileread(int f1);
void delay(unsigned int number);
void sd_copy(ULONG srcsect,ULONG destsect);

void wait_tx_end();

#endif


```

7.2.4 Graphic LCD driver code

```
/*
Library: Graphic LCD functionality drivers

Description: Graphic LCD initialization, cursor set, putchar and put str drivers
have been coded in this library

Author: Aniket Kumar

Date: 11/26/2014
*/

#include "glcddriver.h"

// PIN ASSIGNMENTS

// WR - P2.0
// RD - P2.1
// CE - P2.2
// CD - P2.3
// RST - P2.4
// HALT - P2.5

#define WR P2_0
#define RD P2_1
#define CE P2_2
#define CD P2_3
#define RST P2_4
#define HALT P2_5


xdata char CDAT1,CDAT2,COM ,DAT,DAT_R,CLR1,CLR2;

// LCD put character
void glcd_putch(char ch)
{
DAT = ch; // Put character on data lines
wrdat();
busycheck();
WR=1;
CE=1;
}

// Initialize graphic LCD
void glcd_init()
{
RST = 0;
delay_lcd(10);
RST = 1;


CDAT1 = 0x00;
CDAT2 = 0x00;
COM = 0x42; // Graphic Home address set command
wrcom ();

CDAT1 = 0x00;
CDAT2 = 0x00;
COM = 0x43; // Graphics area set command
wrcom ();

CDAT1 = 0x00;
CDAT2 = 0x00;
COM = 0x40; // Text Home address set command
wrcom ();

CDAT1 = 0x14;
CDAT2 = 0x00;
COM = 0x41; // Text area set command
wrcom (); // Write two byte command
```

```c

busycheck();
COM = 0x80;
wrcom1(); // Write single byte command
busycheck();

busycheck();
COM = 0xA0;
wrcom1();
busycheck();

busycheck();
COM = 0x94; // Text attribute mode
wrcom1();
busycheck();

clrscr(400); // Clear LCD screen

// Init write sequence
CDAT1=0x00;
CDAT2=0x00;
COM=0x24; // Set address pointer
wrcom();
busycheck();
COM=0xB0; // Auto write set
wrcom1();
busycheck();

CE=1;
}

// Delay for LCD routines
void delay_lcd(unsigned int number)
{
unsigned int i,j;
for(i=0;i<number;i++)
{
for(j=0;j<100;j++)
{
}
}
return;
}

// Write Command - two bytes
void wrcom()
{
busycheck();
DAT = CDAT1;
wrdat();
busycheck();
DAT = CDAT2;
wrdat();
busycheck();
CD = 1;
CE = 0;
RD = 1;
P0=COM;
WR=0;
}

// Busycheck
void busycheck()
{
CD = 1;
CE = 0;
WR = 1;
RD = 0;
P0 = 0xFF;
delay_lcd(1);

delay_lcd(1);
```

```
RD = 1;
}

//Busycheck3
void busycheck3()
{
CD = 1;
CE = 0;
WR=1;
RD=0;
P0=0xff;
delay_lcd(1);
RD = 1;

}

//Read Data
void rddat()
{
CD = 0;
CE = 0;
P0=0xff;
WR=1;
RD=0;
DAT_R=P0;
RD=1;
}

// Write command - single byte
void wrcom1()
{
CD = 1;
CE = 0;
RD = 1;
WR=0;
P0=COM;
delay_lcd(1);
WR=1;
}

//Write data
void wrdat()
{
CD = 0;
CE = 0;
RD = 1;
WR=0;
P0=DAT;
delay_lcd(1);
WR=1;
}

// Clear LCD screen
void clrscr(int n)
{
int i;
CDAT1=0x00;
CDAT2=0x00;
COM=0x24; // Set address pointer
wrcom();
busycheck();
COM=0xB0; // Set auto write set
wrcom1();
busycheck();
CLR1=0x00;
CLR2=0x20;
DAT=0x00;

for(i=0;i<n;i++)
{
busycheck3();
wrdat();
```

```c
}
busycheck();
COM=0xB2; // Auto mode reset
wrcom1();
busycheck();
}

// Goto X,Y cursor position
void glcd_gotoxy(char x,char y)
{
CDAT1 = 0x00;
CDAT2 = 0x00;
COM = 0x40;
wrcom();
busycheck();
CDAT1=x;
CDAT2=y;
COM=0x21; // Set cursor pointer
wrcom();
busycheck();
}

// LCD put string function
void glcd_putstr(char *lcdstr)
{
int lcount = 0;
char lch;
while(lcdstr[lcount] != '\0')
{
lch = glcd_format(lcdstr[lcount++]);
glcd_putch(lch);
}
}

// COnvert ASCII data to display in T6963C LCD format character map
char glcd_format(char tbyte)
{
char rbyte;

// Format ASCII characters according to the character map in T6963C controller

if((tbyte >= 'A') && (tbyte <= 'Z'))
{
rbyte = tbyte - 0x20;
}
else if((tbyte >= 'a') && (tbyte <= 'z'))
{
rbyte = tbyte - 0x20;
}
else if((tbyte >= '0') && (tbyte <= '9'))
{
rbyte = tbyte - 0x20;
}
else if((tbyte >= ' ') && (tbyte <= '/'))
{
rbyte = tbyte - 0x20;
}
else if((tbyte >= ':') && (tbyte <= '@'))
{
rbyte = tbyte - 0x20;
}
else
{
rbyte = 0x5F;
}
return rbyte;
}

```

7.2.5 Graphic LCD Header file

```c
/*
Header: Graphic LCD functionality drivers

Description: Graphic LCD initialization, cursor set, putchar and put str drivers
have been coded in this header file

Author: Aniket Kumar

Date: 11/26/2014
*/
#include <mcs51/8051.h>
#include <at89c51ed2.h> //also includes 8052.h and 8051.h
#include <stdio.h>
#include <stdlib.h>
#include <mcs51/8052.h>
#include <string.h>

#ifndef GLCDDRIVER_H_INCLUDED
#define GLCDDRIVER_H_INCLUDED

void glcd_init();
void delay_lcd(unsigned int number);
void wrcom();
void busycheck();
void busycheck3();
void rddat();
void wrdat();
void clrscr(int n);
void wrcom1();
void glcd_putch(char ch);
char glcd_format(char tbyte);
void glcd_putstr(char *lcdstr);
void glcd_gotoxy(char x,char y);

#endif

```

7.2.6 Serial Initialization driver code

```c
/*
Program: Timer program

Description: This file consists of serial port initialisation and write functions.

Author: Aniket Kumar

Date: 11/20/2014
*/

#include <mcs51/8051.h>
#include <at89c51ed2.h> //also includes 8052.h and 8051.h
#include <mcs51reg.h>
#include <stdio.h>
#include <string.h>
#include "serialinit.h"

/* Initialize Serial Port */
void serinit()
{
SCON = 0x50;
TMOD |= 0x20;
TH1 = 0xFD;
TR1 = 1;
TI = 1;
}

/* Print a string to serial port */
int putstr (char *s)
{
int i = 0;
while (*s){ // output characters until NULL found
putchar(*s++);
i++;
}
return i+1;
}

/* Print a character to serial port */
void putchar (char c)
{
while (!TI);
SBUF = c; // load serial port with transmit value
TI = 0; // clear TI flag
}

/* Receive a character from serial port */
char getchar ()
{

while (!RI);
RI = 0; // clear RI flag
return SBUF; // return character from SBUF
}

```

7.2.7 Serial driver header file

```c
/*
Program: Serial Initialization program

Description: This file consists of serial port initialisation and write functions.

Author: Aniket Kumar

Date: 11/20/2014
*/


#ifndef SERIALINIT_H_INCLUDED
#define SERIALINIT_H_INCLUDED

void serinit(); // Initialize serial port
int putstr (char *s); // Put str function on terminal
void putchar (char c); // Put character function for terminal
char getchar(); // Get character function for terminal

#endif // SERIALINIT_H_INCLUDED
```

7.2.8 Integer header file

```c
/*-------------------------------------------*/
/* Integer type definitions for FatFs module */
/*-------------------------------------------*/

#ifndef _FF_INTEGER
#define _FF_INTEGER

#ifdef _WIN32 /* FatFs development platform */

#include <windows.h>
#include <tchar.h>

#else /* Embedded platform */

/* This type MUST be 8 bit */
typedef unsigned char BYTE;

/* These types MUST be 16 bit */
typedef short SHORT;
typedef unsigned short WORD;
typedef unsigned short WCHAR;

/* These types MUST be 16 bit or 32 bit */
typedef int INT;
typedef unsigned int UINT;

/* These types MUST be 32 bit */
typedef long LONG;
typedef unsigned long ULONG;

#endif

#endif

```