**INTERFACING SD CARD FAT32 FILE SYSTEM WITH SPARTAN 6 FPGA NUMATO - MIMAS V2 DEVELOPMENT BOARD**

**Scope**

This project explains how to interface the SD card with an FPGA. In this project, Spartan 6 (XC6SLX9-3csg324) FPGA is used. The FPGA runs on 5V power supply with a built in oscillator frequency of 100 MHz. A 4GB micro SDHC card (class 6) from Strontium is used in this particular project. The SD card is formatted with FAT32. The ultimate aim of this project is to read a BMP image file from the SD card. The SD card has been formatted as FAT32 before interfacing. The generalized code for the FAT32 is written to interface the SD card. Explanations of the FAT32 file system and how to access files from these file system is explained in this project.

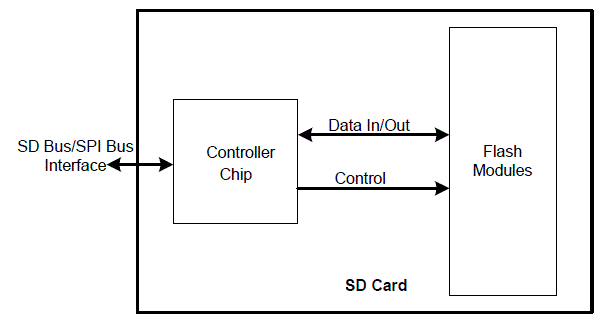
The following section explains the SD card working in detail.

**1  SD CARD**

The SD card is consisting of 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.

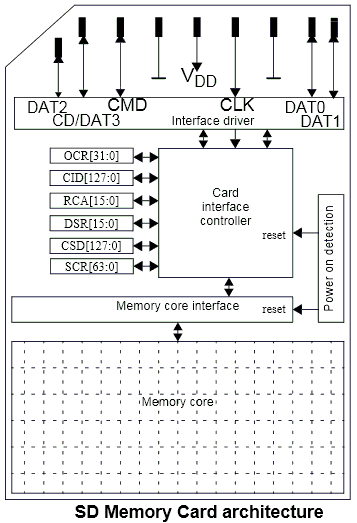
 The ‘SD card controller’ helps to communicate the ‘memory core’ with the external devices like FPGAs, FPGA. It can respond to certain set of standard SD commands and read or write data from the memory core in for the external device.



*Fig. 1: Functional Block Diagram of SD Memory Card*

The capacity of the ‘memory core’ is referred to as the size of the SD card. Other than the ‘memory core’ there are certain registers associated with the ‘SD card controller’. These registers store the status of the SD card. The contents of these registers are read only.

The SD card can be interfaced with the FPGA using serial data bus. It can connect using ‘SD buses’ or ‘SPI buses’. The ‘SD bus’ is designed for high speed whereas the SPI bus can operate with much lower speed only. The FPGA can read or write data the memory core and read the registers using standard SD commands send through these serial buses.



*Fig. 2: Memory Architecture of SD Card*

In this project the memory card is interfaced using the SPI bus. Certain commands are not available for the SPI mode of interfacing and also the speed will be lower than the SD mode. But this kind of interfacing is a lot simpler especially due to the fact that most of the FPGA has I/O and easily interface SPI controller module.

This section summarizes that the SD card has an internal controller chip, a memory core region. The internal controller can decode the commands, provide serial interface while the memory core region is where the file system is implemented.

Based on this knowledge the following section tries to explain the SD card functional layer concept.

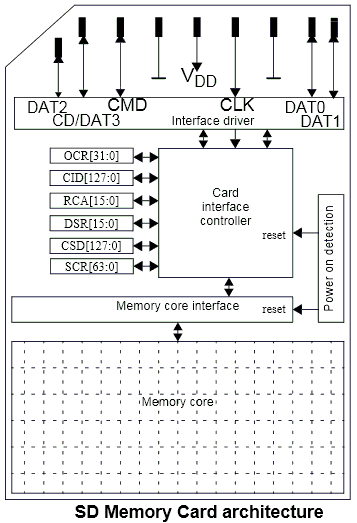
**2 SD CARD FUNCTIONAL LAYERS**

The internals of the SD card can be explained with the help of functional layer concept. Basically there are three layers

1)     Serial interface layer

2)     SD commands layer

3)    File system layer

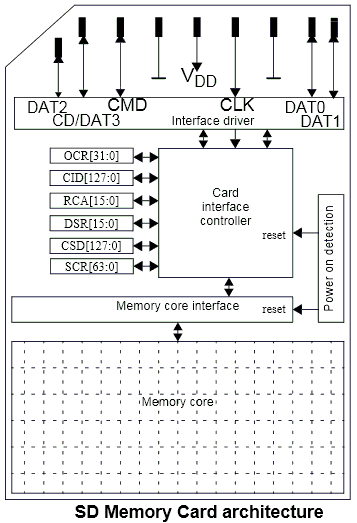


*Fig. 3: Functional Layers of common SD Card*

The ‘Serial interface layer’ and the ‘SD commands layer’ can be viewed as residing inside the ‘SD controller’ and the ‘File system layer’ can found inside the ‘Memory core’. Since the ultimate aim of this project is to read a file from the FAT32 file system of the SD card, it is necessary to access all the three layers in the proper way.

**2. 1    SERIAL INTERFACE LAYER**

This layer provides serial interface of the SD card with the FPGA. In this project the SPI bus is used for the serial interface. The following diagram shows how to interface more than one SD card with a FPGA.

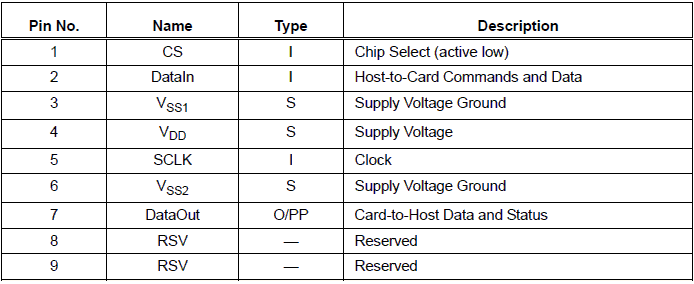


*Fig. 4: Block Diagram interfacing multiple SD card with FPGA*

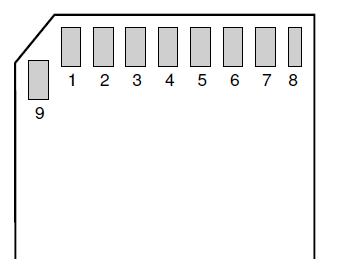
 In this kind of interfacing the FPGA is called the ‘Host’ and the SD card is called the ‘Slave’. The FPGA initiates all the data transfers. The clock is also controlled by the FPGA. The FPGA is free to choose between the SD cards by asserting the respective CS (Chip Select) pin.

The data is transmitted from the FPGA to the SD card using the MOSI (Master Output Slave Input) channel and the data is transferred by the SD card to the FPGA using the MISO (Master Input Slave Output) channel.

The pin out of a SD card for the SPI interfacing mode is shown in the following figure.



*Fig. 5: Pin Numbers of SD card for SPI interfacing mode*

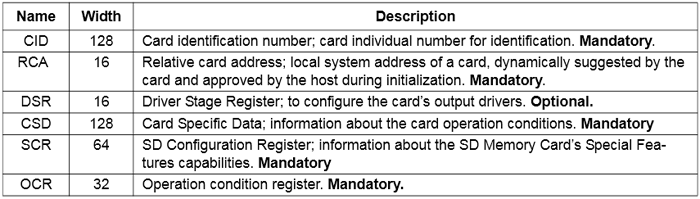


*Fig. 6: SD Card with PIN Out*

**2.2**   **SD COMMANDS LAYER**

The SD card accepts only a set of standard SD commands. Using this commands a FPGA can read the registers of the SD card, and also read/write the ‘Memory Core’.

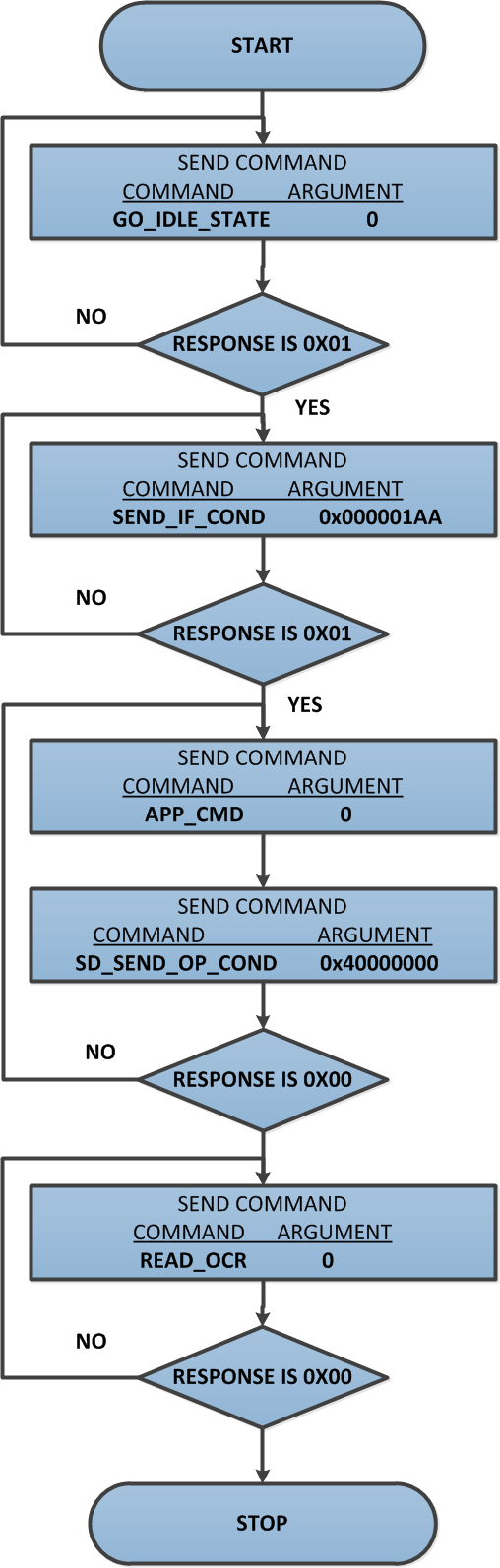
There are six basic registers found in the SD cards, whose details are given in the following figure:



*Fig. 7: Registers in SD card with Description*

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.

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



*Fig. 12: Structure of command block in the SPI interface mode of SD card*

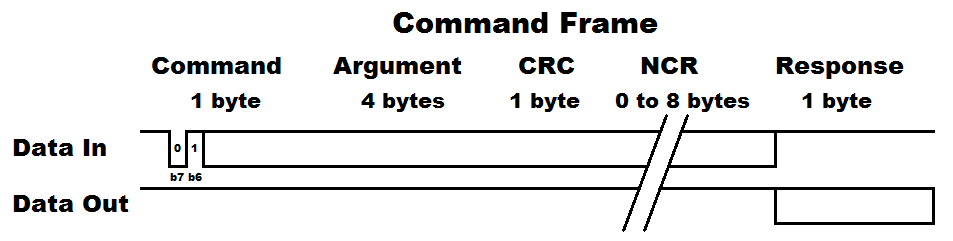
The CRC is mandatory for only a couple of commands in the SPI mode. The 7 – bit CRC forms the first seven bits of the last byte of the command and the eighth bit (end bit) should be always set to one.

Using the above equations and end bit condition, the CRC byte for the 0th command and the 8th command are calculated as 0x95 and 0x87 respectively. For all other commands the CRC is ignored by almost all the SD cards.

**2.2.1 SPI Commands**

I will be running the SD card in SPI mode, the commands have a certain protocol that must be followed. For this project I will only be showing the commands required to initialised, read and write. Firstly the protocol. The commands always have bit 6 set, or decimal 64 added, so for example command 0 would be 64, command 2 would be 65, etc... The argument is our address we want to read or write to. The CRC is a checksum, it does not need to be valid for SPI commands but it must be present. NCR is the processing time of a command before a response is given, the NCR is continuously sent until the response is received. There is one main response (R1) which tells us of any errors, there is another response (R3) which is an extra 4 bytes after the R1 telling us a status register such as the OCR.

The memory core in an SD card is made up of 512byte blocks, a single byte cannot be read.

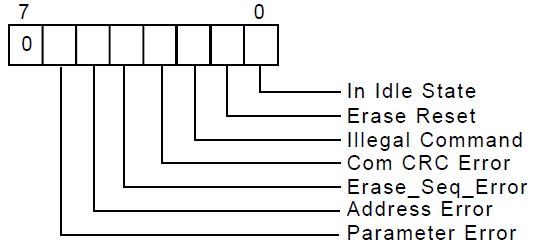


|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Command | Argument | Response | Data | Description |
| CMD0 | None / Zero | R1 | No | Software reset |
| ACMD41 | Zero | R1 | No | Initialise SDHC |
| CM8 | 000001AA | R7 | No | Check card type / voltage |
| CMD12 | None / Zero | R1 | No | Stop reading data |
|  |  |  |  |  |
| CMD17 | Address [31:00] | R1 | Yes | Single block read |
| CMD18 | Address [31:00] | R1 | Yes | Multiple block read |
| ACMD23 | Block quantity [22:00] | R1 | No | Set number of blocks to erase |
| CMD24 | Address [31:00] | R1 | Yes | Single block write |
| CMD25 | Address [31:00] | R1 | Yes | Multiple block write |
| CMD55 | None / Zero | R1 | No | Sent after a ACMD command |
| CMD58 | None / Zero | R3 | No | Read OCR |

**2.2.2 COMMAND RESPONSE**

In SPI mode the SD card response to all the incoming command using three basic types of command response, R1, R2 and R3. Each bit in the response block contains some specific details about the status of the SD card.

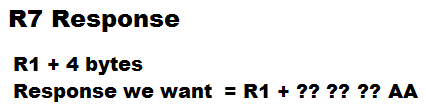
Response – R1



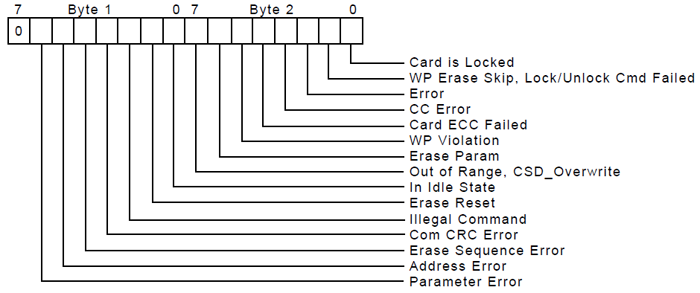
*Fig. 14: SD card command Response 1 in SPI Mode*

 Response – R7

The R7 Response again is a product of the R1 response, it gives very similar information back compared to R3. All we need to ensure is that the last byte is equal to AA, that's it.

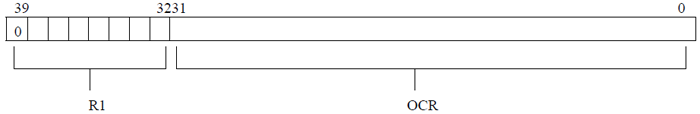


Response – R2



*Fig. 15: SD card command Response 2 in SPI Mode*

Response – R3



*Fig. 16: SD card command Response 3 in SPI Mode*

Response R3 is available only for the 58th command (READ\_OCR).

**2.2.3 SD CARD INITIALIZATION**

There is a specific procedure for initializing an SD card into SPI mode.

All SD card slots will have a switch in the back to say when a card has been inserted.

The cards select pin must be pulled high which deselects the card. At least 76 - 160 pulses must be made to the clock in order for the SD to initialize itself, they have no internal clocking source.

Source code : Sdcard\_controller.v

Command 0 is a software reset which puts the SD card into an idle state, once in this state it can be setup to run in SPI mode. Only one NCR is required.

cmd\_out <= 56'hFF\_40\_00\_00\_00\_00\_95

Command 8 is to check we are using the correct card, if not this particular program will keep looping back to the beginning. This part of the initialization procedure is required.

cmd\_out <= 56'hFF\_48\_00\_00\_01\_AA\_87;

After the R1 response we receive the R3 response, all you need to know is that the final byte to be received must be (hex) AA which tells us that SD card version 2 has been recognised, which is a SDHC, anything else then we start from the very beginning. Only one NCR is required.

Command ACMD41 will finally place the SD card into SPI mode. Only one NCR is required.

cmd\_out <= 56'hFF\_69\_40\_00\_00\_00\_01;

On the first pass the idle flag will still be set, when it is clear the initialisation process is complete. If not then command 55 will be issued, the strange thing about ACMD commands is they are all followed by command 55.

cmd\_out <= 56'hFF\_77\_00\_00\_00\_00\_65;

Note: CRC's, NCR's or zero arguments are generally sent as 0xFF. The reason the data is left high is because this is the way that the SD card works, if the SD card is busy then it's data output pin will be low, when it is ready the data output pin will be high, this is very useful when writing to the card, it means we don't have to send commands to read status registers.

**2.2.4 COMMANDS FOR WRITING DATA**

The data can be written to the ‘Memory Core’ of the SD card using the commands given below followed by the actual data;

WRITE\_BLOCK – Write data to a single block (512bytes)

**WRITE\_BLOCK**

In the SD card a block is always considered as consecutive 512 bytes memory locations. Suppose a block starting from the 2000th memory location need to be written with some data using the WRITE\_BLOCK command. The command packet should be like as shown below;

1st byte (command) – 0x18

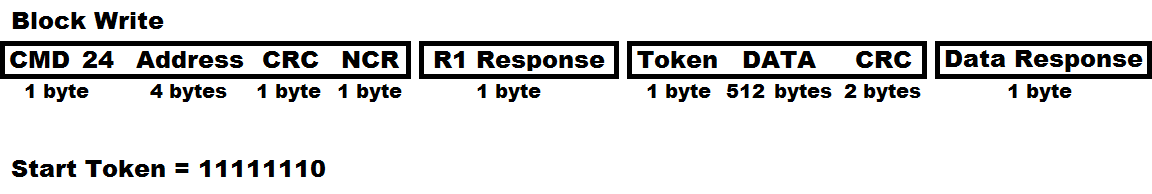
2nd to 5th byte (argument) – 0x000007d0 (Even if there is no arguments for other commands, this field should be set to zero)

6th byte (CRC) – any value

7th byte NCR

Once the command has been send the FPGA should receive the R1 response. All the bits in the response are supposed to be zero. After receiving the zero valued R1 response byte, the FPGA can send the data to be written into the SD card. The length of data should be 512 bytes even though the actual data has less number of bytes.

The 512 byte data should be preceded by a Data Token byte and should be terminated with 16 bit CRC byte. This 1 + 512 + 2 = 515 bytes form a data packet. The Data Token is a byte with all the bits except the LSB is set to 1 (0xFE). The data packet format is shown below:



The command to issue in this case would be 24, dummy bytes are then sent until a clear R1 response is received, a further dummy byte and then the token byte (11111110b) is sent, the 512 bytes of data are sent followed by two CRC's (don't need to be valid), finally a data response is received to say if the data was successful, dummy data is continuously sent until the correct response is received indicating the write has completed, this would be receiving any data apart from zero. No erase command for the SD, it is done by itself in hardware.

cmd\_out <= {16'hFF\_58, address, 8'hFF};

After the data response has been received the SD card is ready to write, in order to do this the SD card has to be de-asserted, clocked eight times (1 byte) and then reasserted before the status can be checked. The above section of program may sometimes work and it is often how datasheets are to be interpreted, however for the correct procedure the SD card has to be deselected to initiate it's write sequence.

When the SD card is busy it will pull it's data output low (only when it has been selected), when it pulls high it means that it is ready. The program below is an alternative method to that above, while they both achieve the same thing we are ideally looking for 0xFF, this is the correct procedure.

For writing the next data block the WRITE\_BLOCK command should be send again.

**2.2.5 COMMANDS FOR READING DATA**

The data can be read from the ‘Memory Core’ of the SD card using the commands given below;

READ\_BLOCK – Read a single block (512 bytes) from sdcard

**READ\_SINGLE\_BLOCK**

In the SD card a block is always considered as consecutive 512 bytes memory locations. Suppose a block starting from the 2000th memory location need to be read using the READ\_SINGLE\_BLOCK command. The command packet should be like as shown below;

1st byte (command) – 0x51

2nd to 5th byte (argument) – 0x000007d0 (Even if there is no arguments for other commands, this field should be set to zero)

6th byte (CRC) – any value

7th byte NCR

Once the command has been send the FPGA should receive the R1 response. All the bits in the response are supposed to be zero. After receiving the zero valued R1 response byte, the FPGA can read the data from the SD card. The data of 512 bytes will be sending by the SD card in response to each READ\_SINGLE\_BLOCK command.

cmd\_out <= {16'hFF\_51,address,8'hFF};

**SDCARD\_CONTROLLER.V working flow chart**

**Input**

Clock - 25Mhz

Reset - Active high reset

Din[7:0] - Data in [ write operation]

Address[31:0] - Sector address [ Read/Write operation]

Wr - Write enable

Rd - Read enable

Multi\_sector\_en - Multiple sectors read enable

I\_blk\_num - Read total number of blocks in multi sector mode

Miso - sdcard send data to the FPGA

**Output**

Cs - chip select

Mosi - FPGA send command/data to the sdcard

Sclk - spi clock

Byte\_counter - byte count upto 512 bytes

Dout, recv\_data - data out from the sdcard

Status - state changes

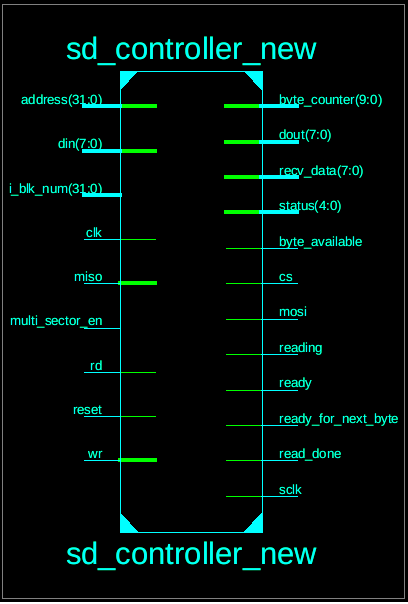
Byte\_available - Ensure the valid data by byte enable

Reading - Data is reading from sdcard

Ready - ready to send the read/write command

Ready\_for\_next\_byte - Each byte have 8 bit. Indicate ready to send next byte

Read\_done - Enable every 512 bytes completed



**State Machine:**

RECV\_DATA[0]==0

NO

YES

IF()

ELSE

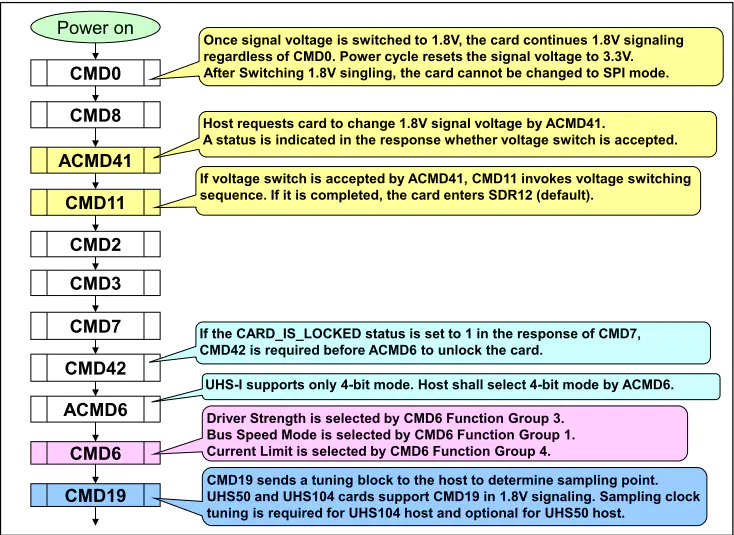
Wr == 1

Rd == 1

WRITE\_BLOCK

READ\_BLOCK

**FIGURE : SDCARD CONTROLLER WORKING FLOW DIAGRAM**

****

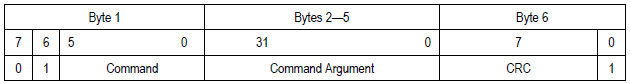
**3 FILE SYSTEM LAYER**

This section explains the FAT32 file system implementation of the SD card in detail. The FAT32 file system is actually written into the ‘Memory Core’ when it was formatted. The FAT32 stands for File Allocation Table 32, means it has a file allocation table of length 32 bits.

The entire data of a file is scrambled across the Memory Core and the FAT (File Allocation Table) holds the location of next block corresponding to the location of the current block.

**SECTORS**

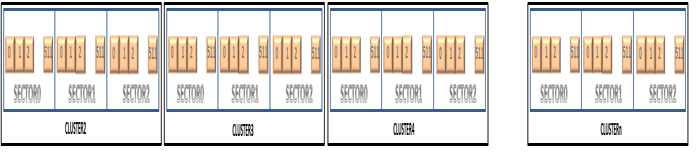
The Memory Core has 8 bit (1 byte) memory locations. The consecutive 8 bit memory locations are grouped into ‘Sectors’. A FAT32 sector usually has 512 bytes per Sector.  The grouping of memory bytes to form Sectors is shown in the following figure.



*Fig. 22: Structure of Sectors in Memory Card*

**CLUSTERS**

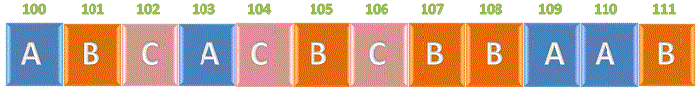
The consecutive Sectors are grouped to form ‘Clusters’. The number of Sectors per Cluster depends on the size of the entire file system. The grouping of Sectors to form Clusters is shown in the following figure.



*Fig. 23: Structure of clusters in Memory Card*

**SCRAMBLED STORAGE**

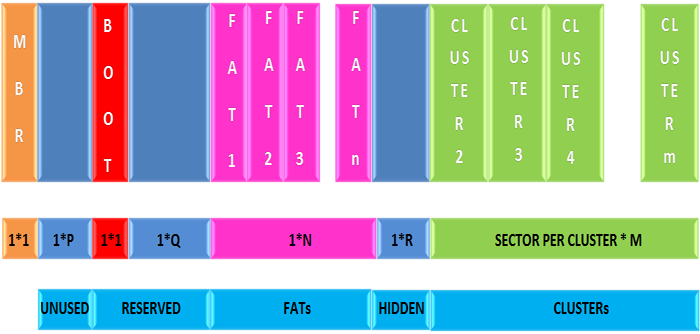
The entire data of a file is scrambled across the Memory Core as Data Clusters. The following figure shows how the data from three files named A, B, C are scrambled across the Memory Core. FAT (File Allocation Table) holds the location of next Cluster corresponding to the location of the current Cluster of the file.



*Fig. 24: Scrambled Data Storage in SD Card*

**3.1 FAT32 FILE SYSTEM FORMAT**

The FAT32 file system is stored or written 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:



*Fig. 25: Format of FAT32 file system in memory Card*

 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.

**3.1.1 Partition**

The MBR (Master Boot Record) is the very first Sector of the ‘Memory Core’ and it is meant to hold the information regarding the partitions inside the file system. The MBR can hold details of four fundamental partitions. The first accessible line of data in an SD card will start at something like address 8192 but in a HEX editor such as WinHex you will not see these first 8192 bytes and it will in fact say sector 0. When in reality reading from sector 0 will actually require address 8192, in general this offset will take this value but of course a computer needs to be certain as it can take other values.

There are a few things first which should be known. Firstly a sector consists of 512 bytes, this is set and cannot be changed. Each bit in the SD card command address corresponds to a sector, therefore single bytes cannot be written or read, only in multiples of 512 byte sectors. A cluster is a multiple of sectors, for example my SD card has been configured to 64 sectors per cluster, this is written into the FAT boot record. Lastly all of the SD card data, not our file data, follows something called "little endian", the data is read backwards, so byte 1, byte 2, byte 3, etc...

Below is an example of the partition, the software does a great job of decoding what all of these bytes mean. The only part of data that we need is the location of the first sector, so we would read from address zero and then continuously read and discard the data until we reach bytes 0x1C6 through to 0x1C9. The first sector is known as the boot sector, this is where most HEX editors will start from, without reading the partition we would be unable to accurately reach this point. The data read is 00 20 00 00 which is actually 0x00002000, following little endian, this corresponds to 8192 decimal.

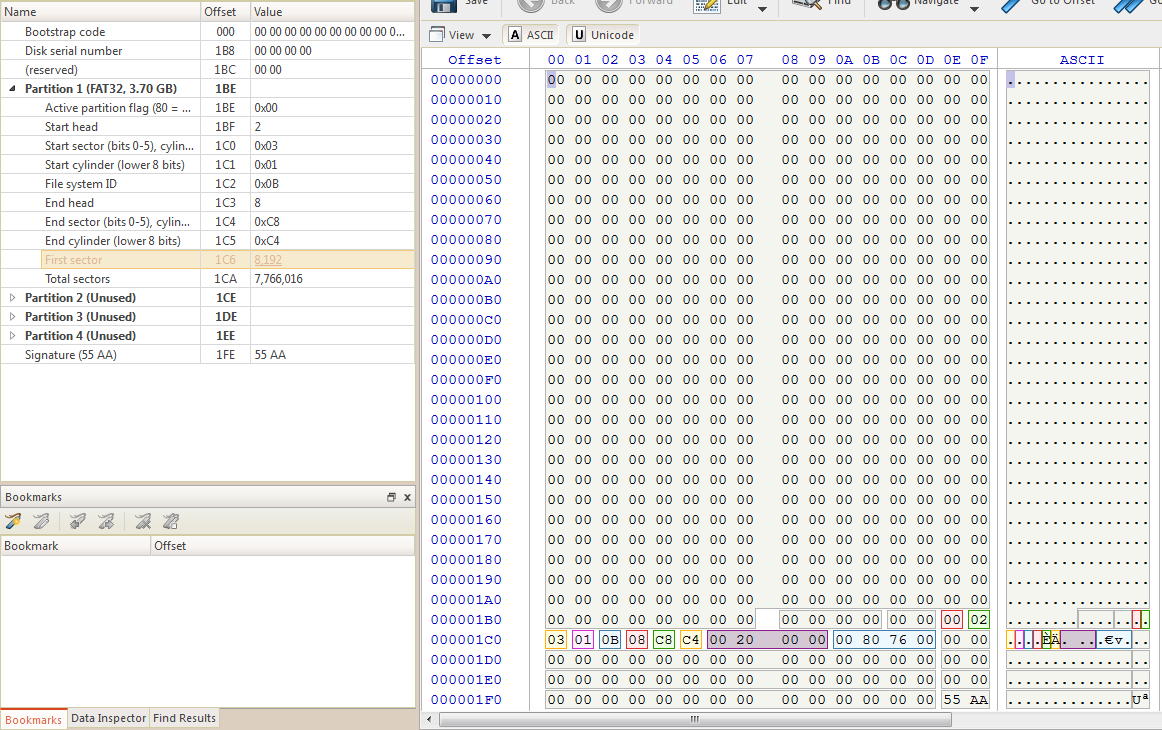
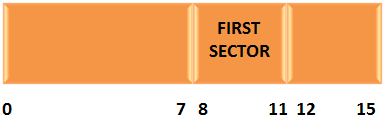


Fig : Partition 512 bytes – Sector 0 of SDCARD

The Partition Info Bytes are 16 bytes long and each of them holds important information regarding the corresponding partitions. The last byte of the MBR is the signature byte which holds a specific value (0xAA55 for FAT32) which can be used to check whether the sector is MBR or not. The four bytes long First Sector number of the partition can be read starting from 8th bit to 11th bit of the Partition Info Bytes as shown below:



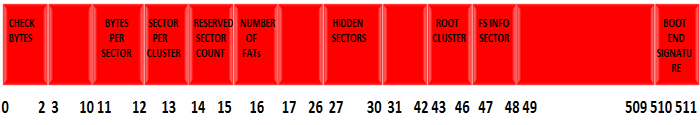
FIRST SECTOR = 00 20 00 00 ( Fig : Partition)

So Boot sector address = 0x00002000 ( in decimal 8192).

*Fig. 27: Partition of First Section in Memory Core*

**3.1.2 MASTER BOOT SECTOR**

The Boot Sector is the very first sector inside a partition. This Sector holds all the valuable details regarding the file system inside that particular partition.



*Fig. 28: Bit Values of Boot Sector in Memory Core*

The first two bytes are Check Bytes which together holds a specific value, usually 0xE9 or 0xEB. The 11th and the 12th bytes when read together give the number of Bytes per Sector for the file system. The FAT32 usually has 512 bytes per Sector. The 13th byte holds the number of Sectors per Cluster for the file system. The 14th and the 15th bytes together read the number of Reserved Sectors before the FATs, starting from the beginning of the partition. The 16th bit holds the value of the number of FATs in the file system. The 4 bytes starting from the 27th byte to 30th byte can be read together to get the number of Hidden Sectors between the FATs and the First Cluster of the partition. The 4 bytes starting from the 43rd byte to the 46th byte holds the address of the root cluster, which is the very first cluster of the partition (usually 2). The 47th and the 48th bytes can be read together to get the location of the Sector where the File Information is stored (Directory Sector).

After finding the start of the boot sector we need to load this value into the read command, so loading 8192 into the SD cards read command will bring us to the boot sector. Here is an example of the boot sector, the main things we need to read from this are, the bytes per sector (this should always be 512), the sectors per cluster, the reserved sectors, the hidden sectors and the sectors per FAT. You may notice the offset in the software can be calculated by sectors x bytes per sector, 8192 x 512 = 4194304, 0x400000.

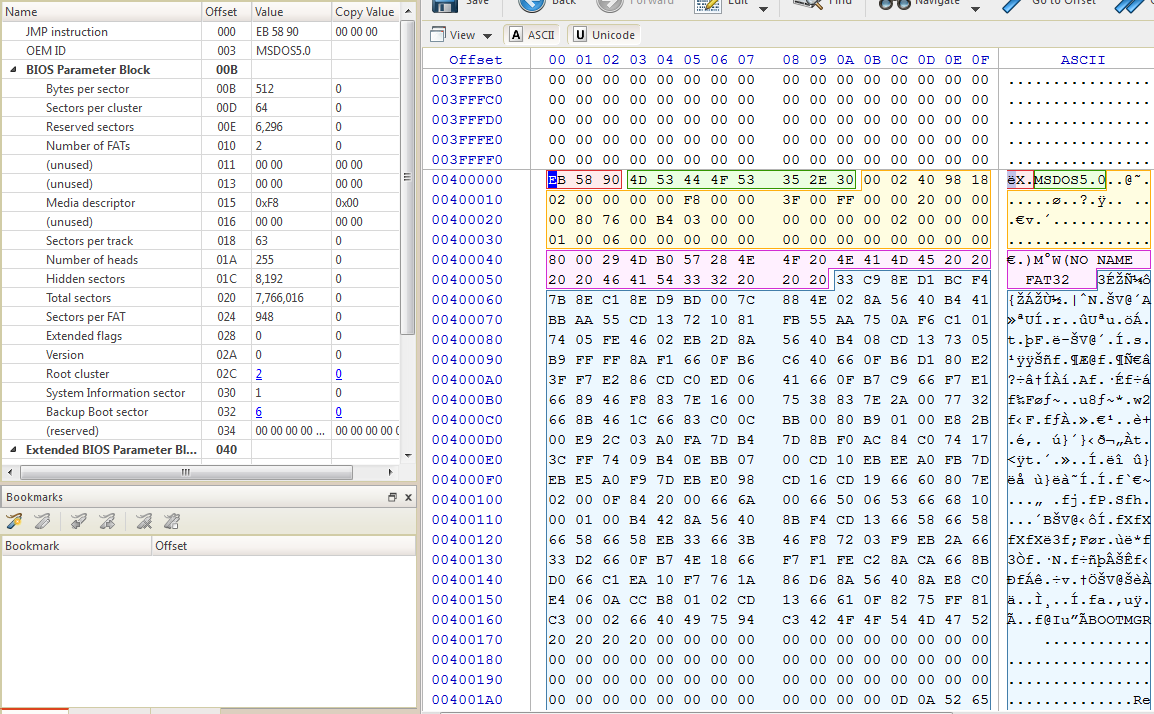


Fig : Master Boot sector/Record

Check Bytes = EB 58 90 ( 0th – 2 bytes)

Bytes per Sector = 00 02 ( 11th – 12th bytes)

0x0200 = 512 bytes

number of Sectors per Cluster = 40 ( 13th byte)

0x40 = 64

number of Reserved Sectors = 98 18 ( 14th& 15th byte)

= 0x1898

= 6296

number of FATs = 02 (16th byte)

number of Hidden Sectors = 00 00 20 00 ( 8192)

**3.1.3 ROOT DIRECTORY**

 To find the start of the root directory we use the formula, "Hidden Sectors + Reserved Sectors + (2 x sectors per FAT)". So using the formula I would get 16384, loading this into the read command would bring me to the root directory as shown in the example below. The root directory contains entries relating to the files saved on the card, each entry takes 32 bytes.

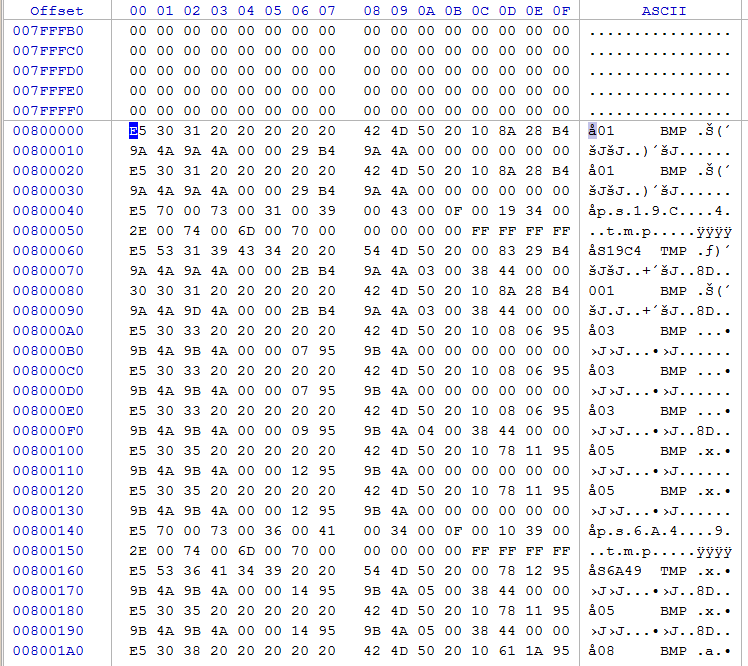


Fig : Root Directory

Taking the first entry from the root directory,

18 Series Microcontroller + SD Card SPI

The entry has been deleted, the file is called "001", the file is an archive, etc... You may also notice that some data is missing such as a the cluster offset and the size of the file, the reason is that this is a deleted file. The following is the actual file, we can see this time that the file is about 17kbyte long and the offset cluster is 3. In the boot sector there was an entry called the "root cluster", it's value was 2, it will always be 2

http://www.edproject.co.uk/_notes/1815Ser5.png

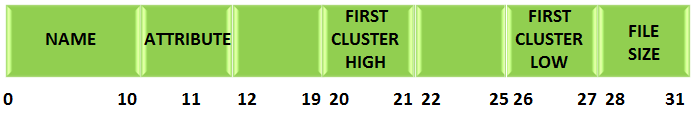
The root cluster is referring to the root directory, so when the cluster offset in the entry above says 3 it means that it is the cluster after the root directory. Our root directory is cluster 2 and our file is in cluster 3. The root directory has the sector address location 16384, since there are 64 sectors per cluster it means that our file is at sector 16448. You will find there are a great deal of entries relating to your file even if you have only ever saved one copy, only one of them will be valid, a disc clean up tool such as a defragmenter will remove these obsolete entries.

**3.1.4 FILESEARCH DIRECTORY SECTOR**

It is the first Sector inside the first Cluster of the file system. The first Cluster always starts with a Cluster number 2. It is also the very first Data Sector of the partition. The FS Directory is 32 bytes long and hence there are 16 FS Directories per FS Directory Sector. The Sector number of the First FS Directory Sector can be found by using the following equation:

DIR\_SECTOR\_NUM = No. RESERVED SECTORS + No. FATS + No. HIDDEN SECTORS

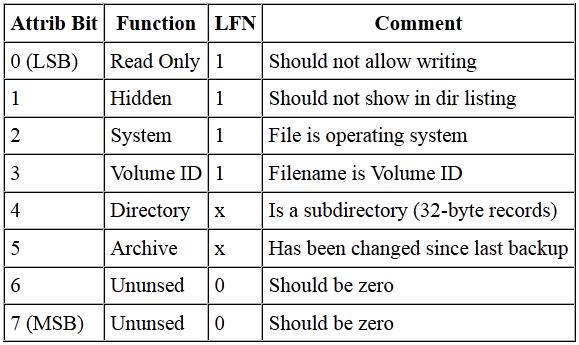
Each FS Directory holds the significant details regarding a single file as shown in the following figure:



*Fig. 30: Structure of FS Directory in SD Card*

|  |  |
| --- | --- |
| File search Directory Entry Data Structure | |
| Bytes | Description |
| 0 | First character of the file name in ASCII, or allocation status 0 = unallocated or E5 = deleted entry |
| 1 to 10 | Remaining characters of the file name |
| 11 | File attributes, see following table |
| 12 | Reserved |
| 13 | File creation time (tenths of seconds) |
| 14 to 15 | File creation time (hours, minutes, seconds) |
| 16 to 17 | File creation date |
| 18 to 19 | Last access date |
| 20 to 21 | High-order cluster offset |
| 22 to 23 | Modification time (hours, minutes, seconds) |
| 24 to 25 | Modification date |
| 26 to 27 | Low-order cluster offset |
| 28 to 31 | File size |

The first 11 bytes holds the Name of the file and the 11th byte holds the attribute of the file. The attribute byte is used to check for a valid file. The content of the attribute byte is shown in the following figure:



*Fig. 31: Content of attribute byte in FS Directory of SD Card*

 The 20th and the 21st byte together hold the higher bytes of the First Cluster number of the File while the 26th and the 27th byte holds the lower bytes of the First Cluster number of the File. The last four bytes hold the size of the file.

http://www.edproject.co.uk/_notes/1815Ser5.png

In this example, 001 BMP is the file name which has 0 to 10 bytes ( 30 30 31 20 20 20 20 20 20 42 4D 50)

11 th byte is the attribute

20th and the 21st byte together hold the higher bytes of the First Cluster number = 00 00

26th and the 27th byte holds the lower bytes of the First Cluster number = 03 00

So cluster number is 03.

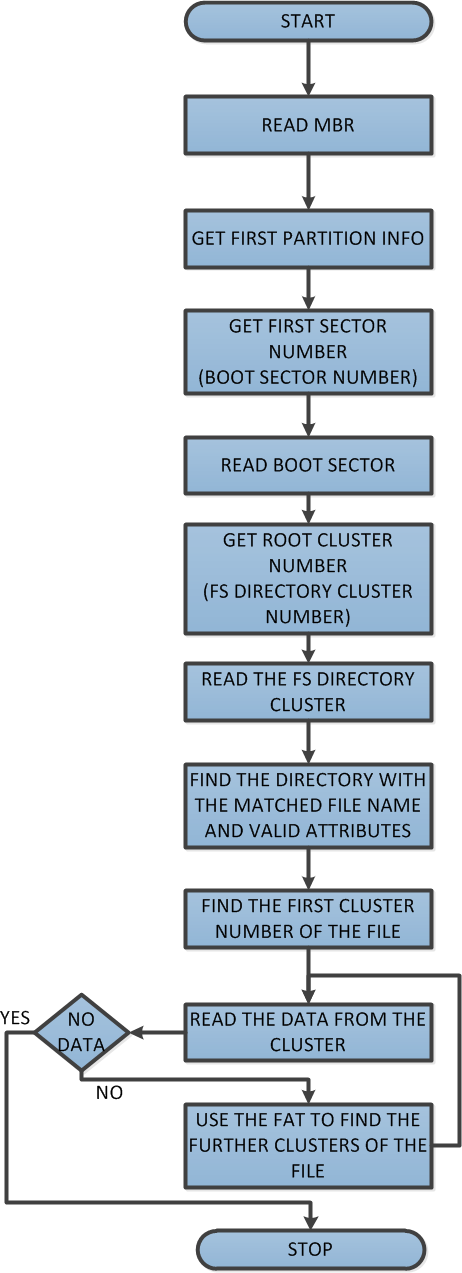
28th to 31th byte is file size = 38 44 00 00

File size = 0x00004438 ( 17kb)

**3.2 THE LOGIC FOR READING A FILE FROM FAT32 FILE SYSTEM**

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 READ\_SINGLE\_BLOCK command from the SD Command Layer alone.

**sdcard\_fat32\_read.V**



*Fig. 32: Algorithm for READING FILE FROM FAT32 FILE SYSTEM*

**3.2.1 USING THE FAT32 (FILE ALLOCATION TABLE 32)**

FAT32 are Sectors in which each consecutive 32 bits together holds the Cluster number of Clusters. Simply each 32 bits point towards a particular Cluster. Since a Cluster normally has 512 bytes, there will be 128 Cluster pointers inside the Sector. This forms the File Allocation Table 32 FAT32.

The number of the next Cluster pointer inside the FAT32 corresponding to a current Cluster number can be calculated by using the following equation

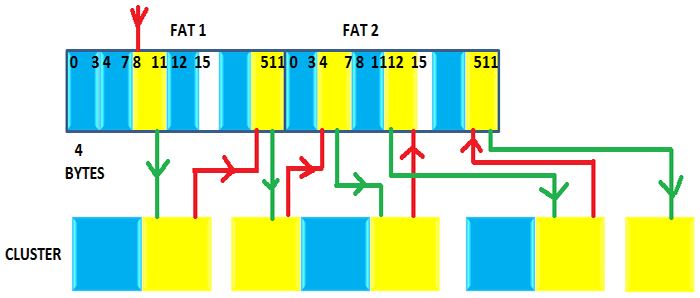
FAT SECTOR NUMBER FOR NEXT CLUSTER POINTER = FIRST SECTOR NUMBER OF THE PARTITION

+ NUMBER OF RESERVED SECTORS

+ ((CURRENT CLUSTER NUMBER \* 4)

/ BYTES PER SECTOR)

The following figure shows the method of reading a file which has been scrambled across the flash ‘Memory Core’ using the FAT32.

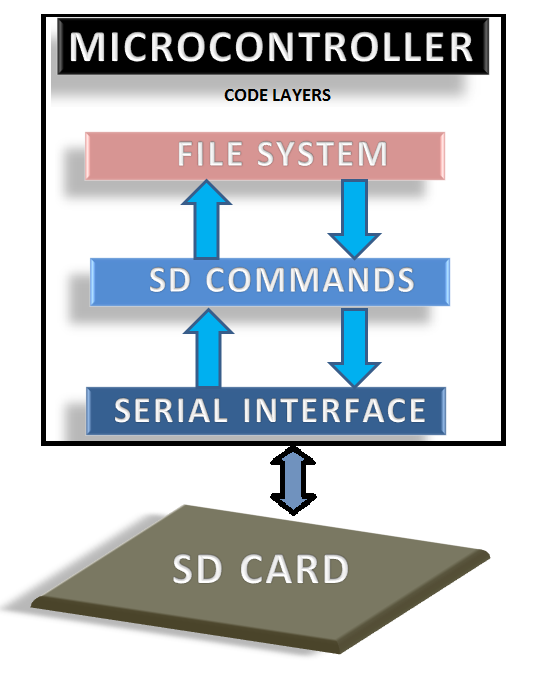


*Fig. 33: Block Diagram to read scrambled file across flash ‘Memory Core’ using FAT32*

The yellow indicates Clusters having the particular file’s data and the corresponding Cluster pointers inside the FAT32. The red lines indicate finding the next Cluster pointer corresponding to the current Cluster and the green line indicate finding the next Cluster using the Cluster number stored inside the Cluster pointers of FAT32.

**4 THE CODING DETAILS**

Since there are three functional layers, namely Serial Interface Layer, SD Commands Layer and File System Layer, the coding is also done for each of the layers separately and then combined together. The code layers are shown in the following figure:



*Fig. 34: Block Diagram of Coding layers in FPGA*

The FPGA read and writes data using the File System Layer. The File System Layer interacts with the SD Commands layer using READ\_SINGLE\_BLOCK command only. The SD Commands Layer interacts with the Serial Interface Layer using SPI transmission and SPI reception function calls. The entire FPGA system interacts with the SD card using the SPI bus interface.

Function pointers have been used for the interaction between different layers effectively. Structures and dynamic memory allocation has been used to store and access 512 bytes of data each time.

**Project Source Code**

Sdcard\_controller.v

Sdcard\_fat32.v

Ram.v (512 )

FAT\_top.v

VGA\_driver.v