

[Arduino Projects \(http://www.engineersgarage.com/microcontroller/arduino-projects\)](http://www.engineersgarage.com/microcontroller/arduino-projects) | [Raspberry Pi \(http://www.engineersgarage.com/embedded/raspberry-pi\)](http://www.engineersgarage.com/embedded/raspberry-pi) | [Electronic Circuits \(http://www.engineersgarage.com/electronic-circuits\)](http://www.engineersgarage.com/electronic-circuits) | [AVR \(http://www.engineersgarage.com/embedded/avr-microcontroller-projects\)](http://www.engineersgarage.com/embedded/avr-microcontroller-projects) | [PIC \(http://www.engineersgarage.com/embedded/pic-microcontroller-projects\)](http://www.engineersgarage.com/embedded/pic-microcontroller-projects) | [8051 \(http://www.engineersgarage.com/microcontroller/8051projects\)](http://www.engineersgarage.com/microcontroller/8051projects) | [Electronic Projects \(http://www.engineersgarage.com/contribution\)](http://www.engineersgarage.com/contribution)

Configure your [account time zone setting](#).

Interfacing SD Card with AVR Microcontroller

AVR028

Summary | **Description** | Circuit Diagram | Video | Code | Components | Browse 4m+ Electronic Products

VCON Controls& Components

Terminal Blocks/ Strips, Enclosures Barrier Single/Double Row EuroStyle



This project explains **how to interface the SD card with an [AVR microcontroller \(http://www.engineersgarage.com/articles/avr-microcontroller\)](http://www.engineersgarage.com/articles/avr-microcontroller)**. In this project an **[ATMEGA16 \(http://www.engineersgarage.com/electronic-components/atmega16-microcontroller\)](http://www.engineersgarage.com/electronic-components/atmega16-microcontroller)** microcontroller is used. The microcontroller runs on 5V power supply with a built in crystal frequency of 8 MHz. A 2GB SDSC card from Transcend is used in this particular project, but the code will work with most of the SD cards. The SD card is formatted with FAT32. The ultimate aim of this project is to read a file from the FAT32 file system of 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.

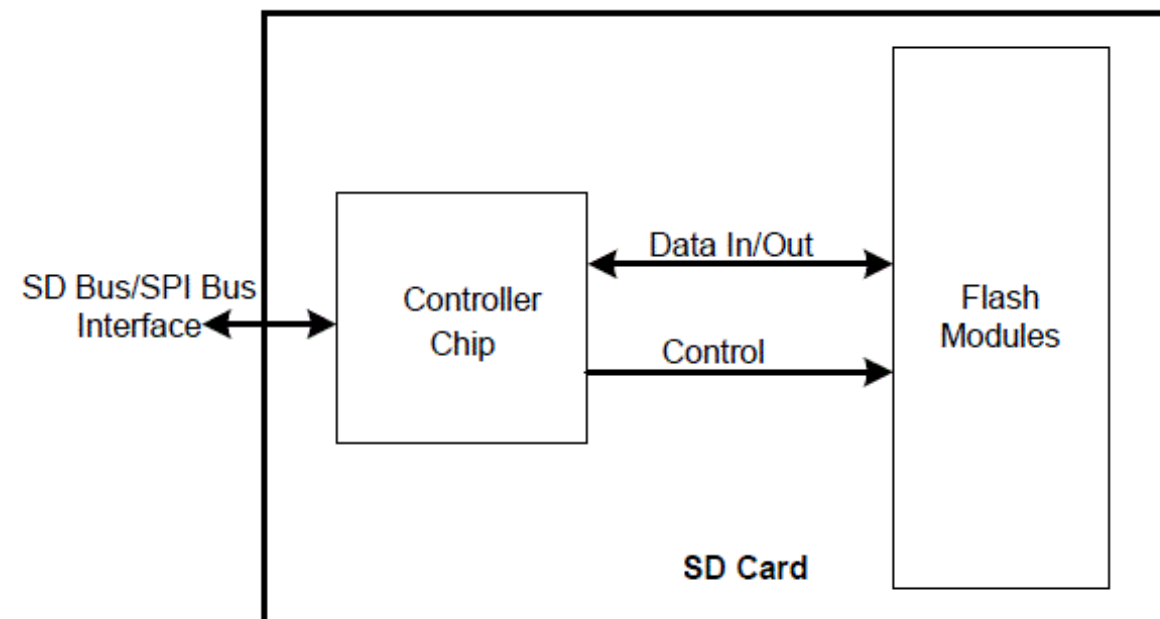
The following section explains the SD card working in detail.

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.

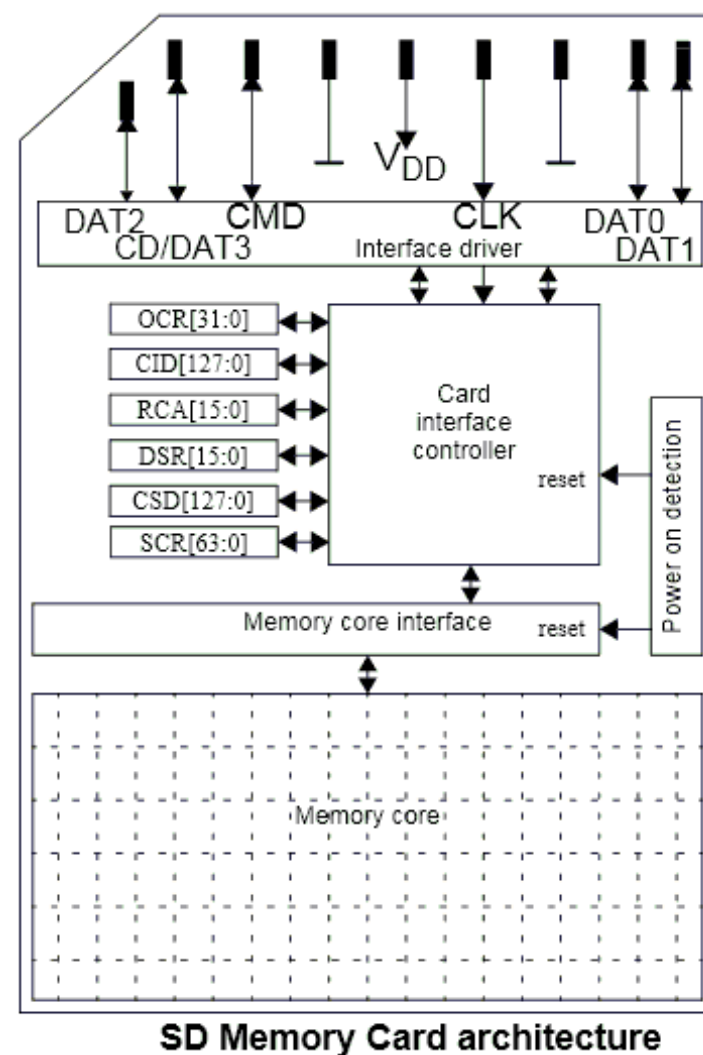
The 'SD card controller' helps to communicate the 'memory core' with the external devices like microcontrollers. It can respond to certain set of standard SD commands and read or write data from the memory core in for the external device.



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 microcontroller 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 microcontroller can read or write data the memory core and read the registers using standard SD commands send through these serial buses.



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 microcontroller has built in SPI hardware 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.

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

- {C} 1) Serial interface layer
- {C} 2) SD commands layer
- {C} 3) File system layer

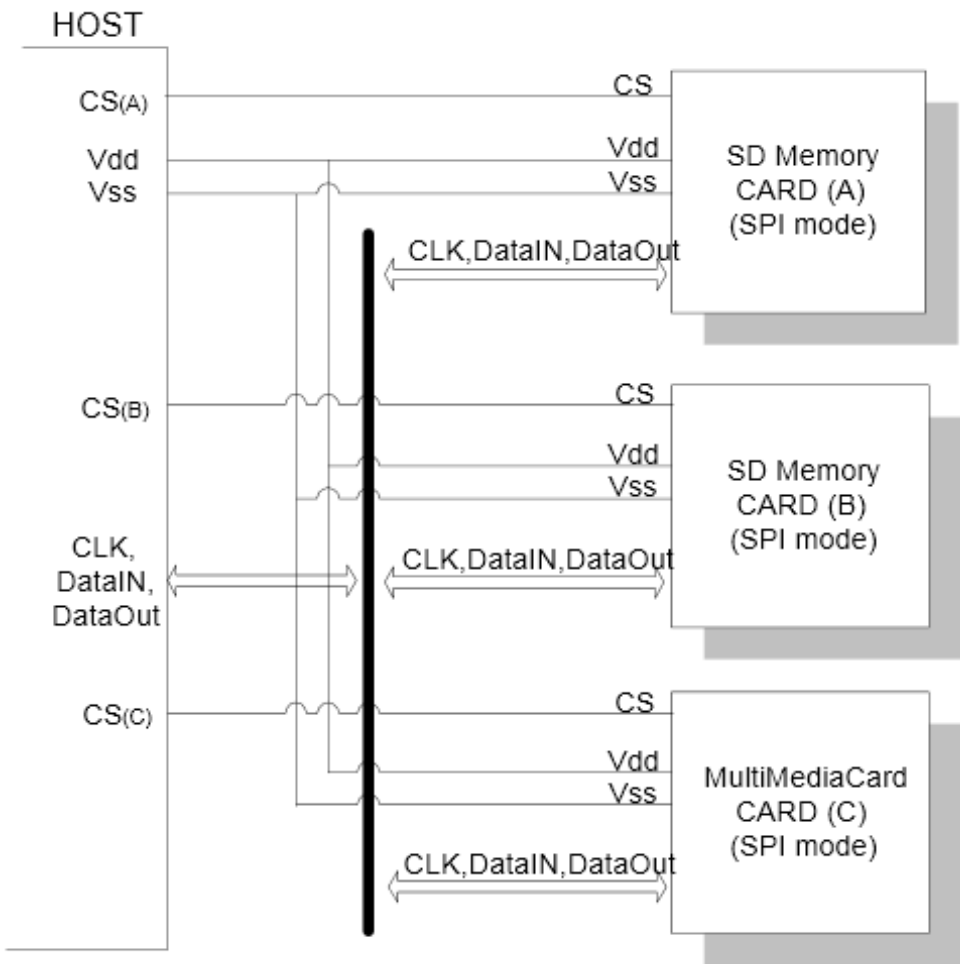


the 'SD commands layer' can be viewed as residing inside the 'SD controller' and the 'File system layer' can found inside the 'Memory core'. Since the ultimate aim of this project is to

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.

{C 1) SERIAL INTERFACE LAYER

This layer provides serial interface of the SD card with the microcontroller. 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 microcontroller.



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

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

The microcontroller used in this project runs on 5V power supply but the SD card can take only up to 3.3V. The logic levels of the 5V and 3.3V are different and these can cause problems. A 3.3V

logic level device may read the logic 0 of a 5V device as logic 1. Similarly the 5V logic level device may read both the logic 0 and logic 1 of a 3.3V logic level device as logic 0 only. This problem can be solved by using a bi-directional level converter introduced between the SPI pins of the microcontroller and the SD card.

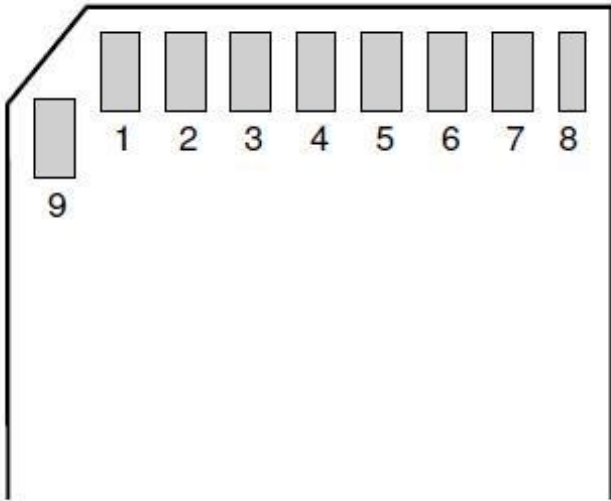
The circuit diagram for the **serial interfacing of a SD card with the microcontroller** including the level converter is shown in the circuit diagram

Pin No	Name	Type	Description
--------	------	------	-------------

tab.

Pin No.	Name	Type	Description
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	SCLK	I	Clock
6	V _{SS2}	S	Supply Voltage Ground
7	DataOut	O/PP	Card-to-Host Data and Status
8	RSV	—	Reserved
9	RSV	—	Reserved

The programmer is 'USBasp' which is also a SPI device. It is connected to the SPI bus along with the SD card. LM1117 is used for providing 3.3V power to the SD card and also to the low voltage side of the level converter. The level converter is



a four channel device with a MOSFET controlling each channel. This kind of level converter is suitable for SPI, I2C buses. The image of the level converter used in this project is shown below.



The image of the AVR SD Card set up

[header= SD Commands Layer]

{C

2)

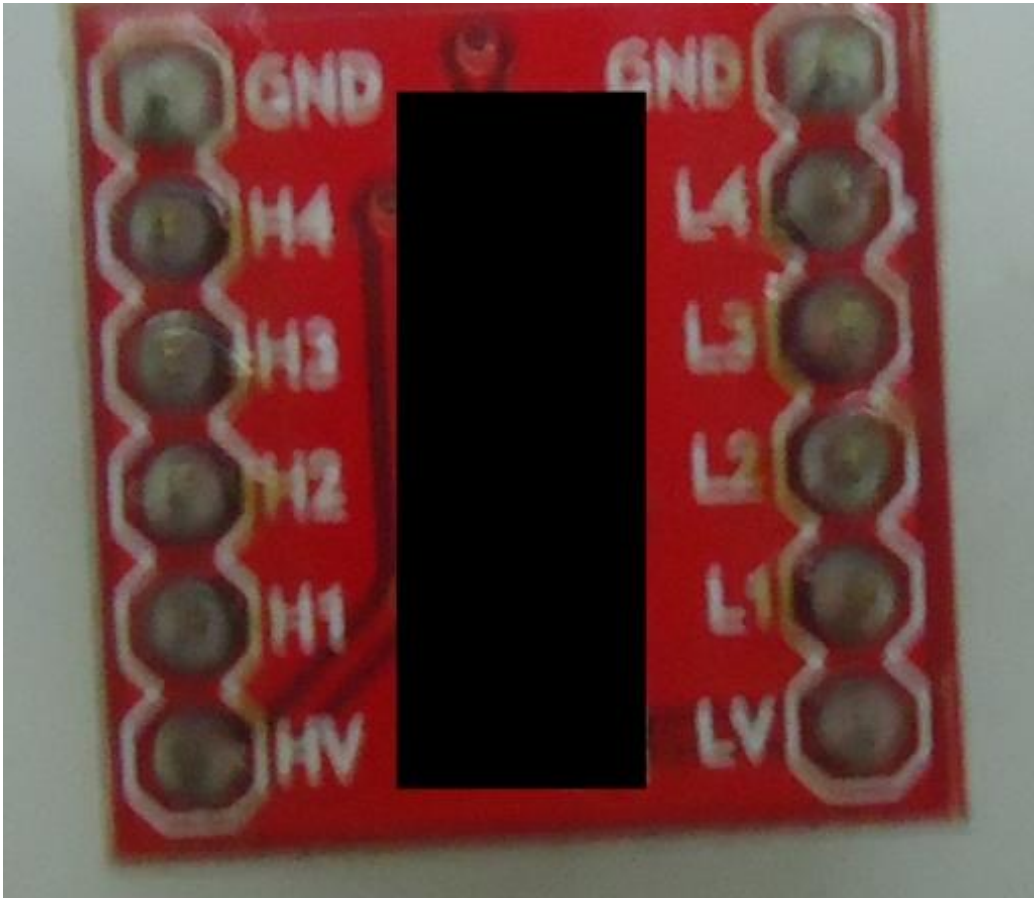
SD

COMMANDS LAYER

The SD card accepts only a set of standard SD commands. Using this commands a microcontroller 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:



details are given in the following figure.

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

The CRC is mandatory for only a couple of commands in the SPI mode. The CRC used is 7 - bit CRC, with a generator polynomial given by

$$x^7 + x^3 + 1$$

The CRC calculation is shown below;

$$G(x) = x^7 + x^3 + 1$$

$$M(x) = (\text{start bit}) \cdot x^{39} + (\text{host bit}) \cdot x^{38} + \dots + (\text{last bit}$$

before CRC) $\cdot x^0$

$$\text{CRC [6...0]} = \text{Remainder [(M(x) \cdot x^7) / G(x)]}$$

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.

The standard SD commands, their arguments, response type and description are given in the following figure.

COMMAND RESPONSE

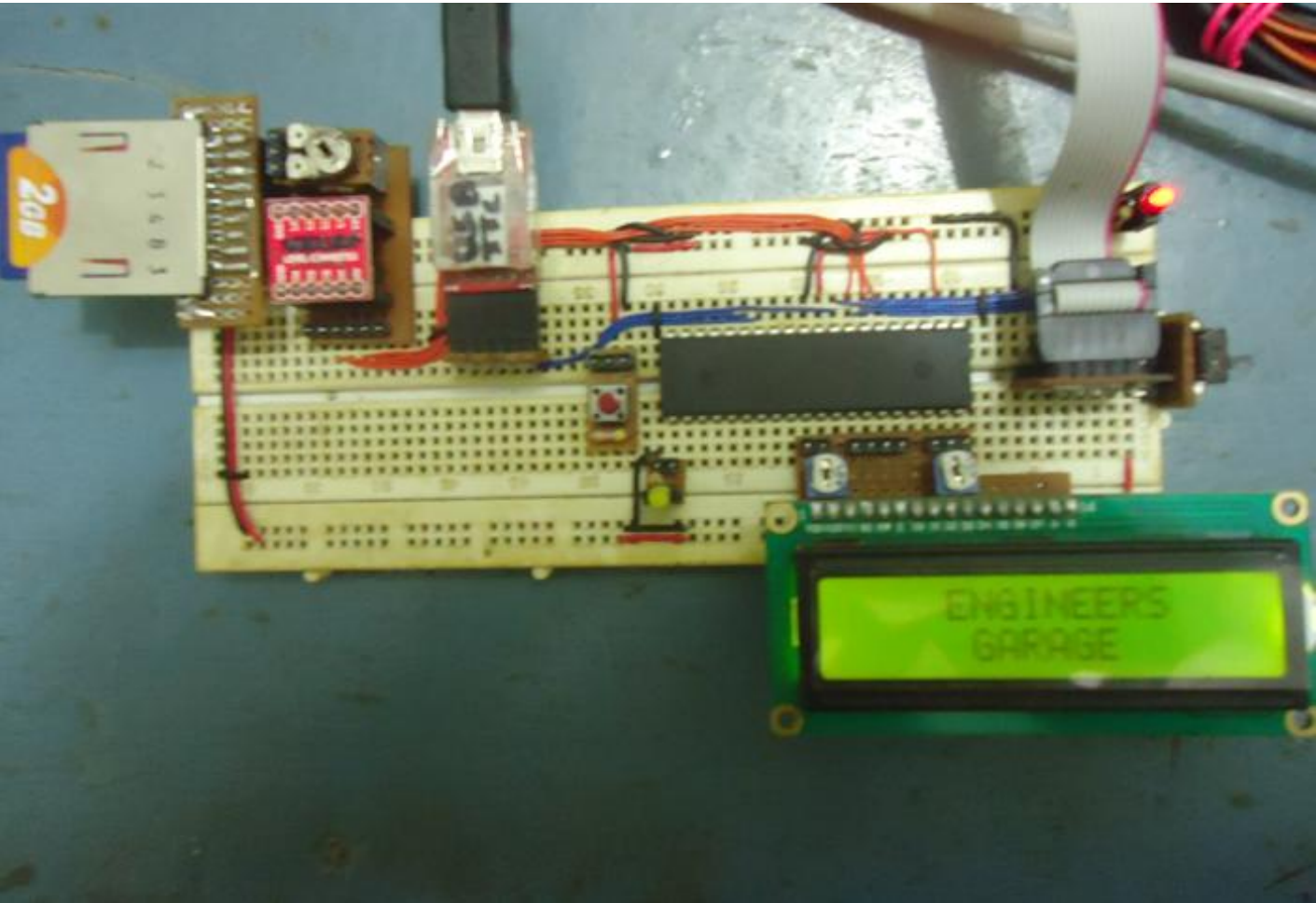
In SPI mode the SD card response to all the incoming command using three basic types of command response, R1, R2 and R3. Each bit in the response block contains some specific details about the status of the SD card.

Response – R1



Response – R2

Response – R3



Name	Width	Description
CID	128	Card identification number; card individual number for identification. Mandatory.
RCA	16	Relative card address; local system address of a card, dynamically suggested by the card and approved by the host during initialization. Mandatory.
DSR	16	Driver Stage Register; to configure the card's output drivers. Optional.
CSD	128	Card Specific Data; information about the card operation conditions. Mandatory
SCR	64	SD Configuration Register; information about the SD Memory Card's Special Features capabilities. Mandatory
OCR	32	Operation condition register. Mandatory.

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

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

Once the command has been send the microcontroller 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 microcontroller can send the data to be written into the SD card. The length of data should be 512

CMD26	No				
CMD27	Yes	None	R1	PROGRAM_CSD	Programming of the programmable bits of the CSD.
CMD28	Yes	[31:0] data address	R1h	SFT WRITE PROT	If the card has write protection features, this

Response – R3

Response R3 is available only for the 58th command (READ_OCR).

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

WRITE_MULTIPLE_BLOCK – Write data to multiple blocks

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

bytes even though the actual data has less number of bytes.

					In the SD card, write protection features, the command sets the write protection bit of the addressed group. The properties of write protection are coded in the card specific data (WP_GRP_SIZE).
CMD29	Yes	[31:0] data address	R1b	CLR_WRITE_PROT	If the card has write protection features, this command clears the write protection bit of the addressed group.
CMD30	Yes	[31:0] write protect data address	R1	SEND_WRITE_PROT	If the card has write protection features, this command asks the card to send the status of the write protection bits.
CMD31	Reserved				
CMD32	Yes	[31:0] data address	R1	ERASE_WR_BLK_START_ADDR	Sets the address of the first write block to be erased.
CMD33	Yes	[31:0] data address	R1	ERASE_WR_BLK_END_ADDR	Sets the address of the last write block in a continuous range to be erased.
CMD34 ... CMD37	Reserved				
CMD38	Yes	[31:0] don't care*	R1b	ERASE	Erases all previously selected write blocks.
CMD39	No				
CMD40	No				
CMD41 ... CMD54	Reserved				
CMD55	Yes	[31:0] stuff bits	R1	APP_CMD	Notifies the card that the next command is an application specific command rather than a standard command.
CMD56	Yes	[31:0] stuff bits [0]: RD/WR.	R1	GEN_CMD	Used either to transfer a Data Block to the card or to get a Data Block from the card for general purpose/application specific commands. The size of the Data Block is defined with SET_BLOCK_LEN command.
CMD57	Reserved				
CMD58	Yes	None	R3	READ_OCR	Reads the OCR register of a card.
CMD59	Yes	[31:1] don't care [0:0] CRC option	R1	CRC_ON_OFF	Turns the CRC option on or off. A '1' in the CRC option bit will turn the option on, a '0' will turn it off.
CMD60-63	No				

COMMANDS FOR READING DATA

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

READ_SINGLE_BLOCK – Read data from a single block

READ_MULTIPLE_BLOCK – Read data from multiple blocks

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) – 0x11

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

CMD INDEX	SPI Mode	Argument	Resp	Abbreviation	Command Description
ACMD6	No				

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:

For writing the next data block the WRITE_BLOCK command should be send again.

WRITE_MULTIPLE_BLOCK

WRITE_MULTIPLE_BLOCK command differs from the WRITE_BLOCK in such a way that this command needs to be send only once and the following 512 bytes long data packets will be written to the consecutive memory blocks.

DATA RESPONSE

For each data block written, the SD card sends a response byte to the microcontroller. The content of the response byte is given as follows:

6th byte (CRC) – any value

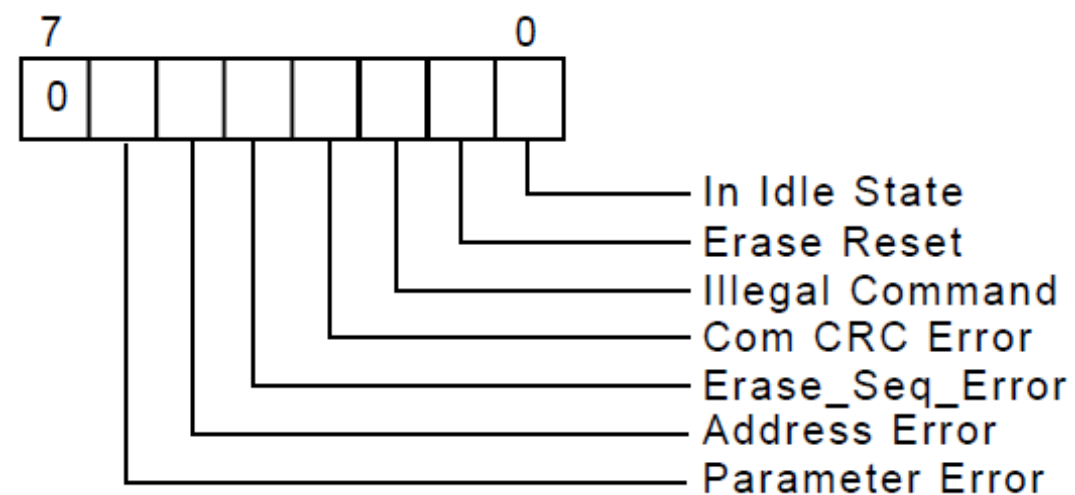
Once the command has been send the

ACMD13	Yes	[31:0] stuff bits	R2	SD_STATUS	Send the SD Card status. The status fields are given in Table 4-21
ACMD17	Reserved				
ACMD18	Yes	--	--	--	Reserved for SD security applications ¹
ACMD19 to ACMD21	Reserved				
ACMD22	Yes	[31:0] stuff bits	R1	SEND_NUM_WR_BLOCKS	Send the numbers of the well-written (without errors) blocks. Responds with 32bit+CRC data block.
ACMD23	Yes	[31:23] stuff bits [22:0]Number of blocks	R1	SET_WR_BLK_ERASE_COUNT	Set the number of write blocks to be pre-erased before writing (to be used for faster Multiple Block WR command). "1"=default (one wr block)(2).
ACMD24	Reserved				
ACMD25	Yes	--	--	--	Reserved for SD security applications ¹
ACMD26	Yes	--	--	--	Reserved for SD security applications ¹
ACMD38	Yes	--	--	--	Reserved for SD security applications ¹
ACMD39 to ACMD40	Reserved				
ACMD41	Yes	None	R1	SEND_OP_COND	Activates the card's initialization process.
ACMD42	Yes	[31:1] stuff bits [0]set_cd	R1	SET_CLR_CARD_DETECT	Connect[1]/Disconnect[0] the 50KOhm pull-up resistor on CD/DAT3 (pin 1) of the card. The pull-up may be used for card detection.
ACMD43 ... ACMD49	Yes	--	--	--	Reserved for SD security applications. ¹
ACMD51	Yes	[31:0] stuff bits	R1	SEND_SCR	Reads the SD Configuration Register (SCR).

microcontroller 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 microcontroller 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.

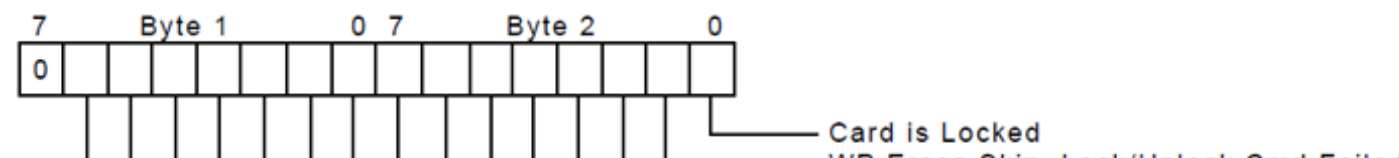
DATA TOKEN

The 512 byte data read 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:

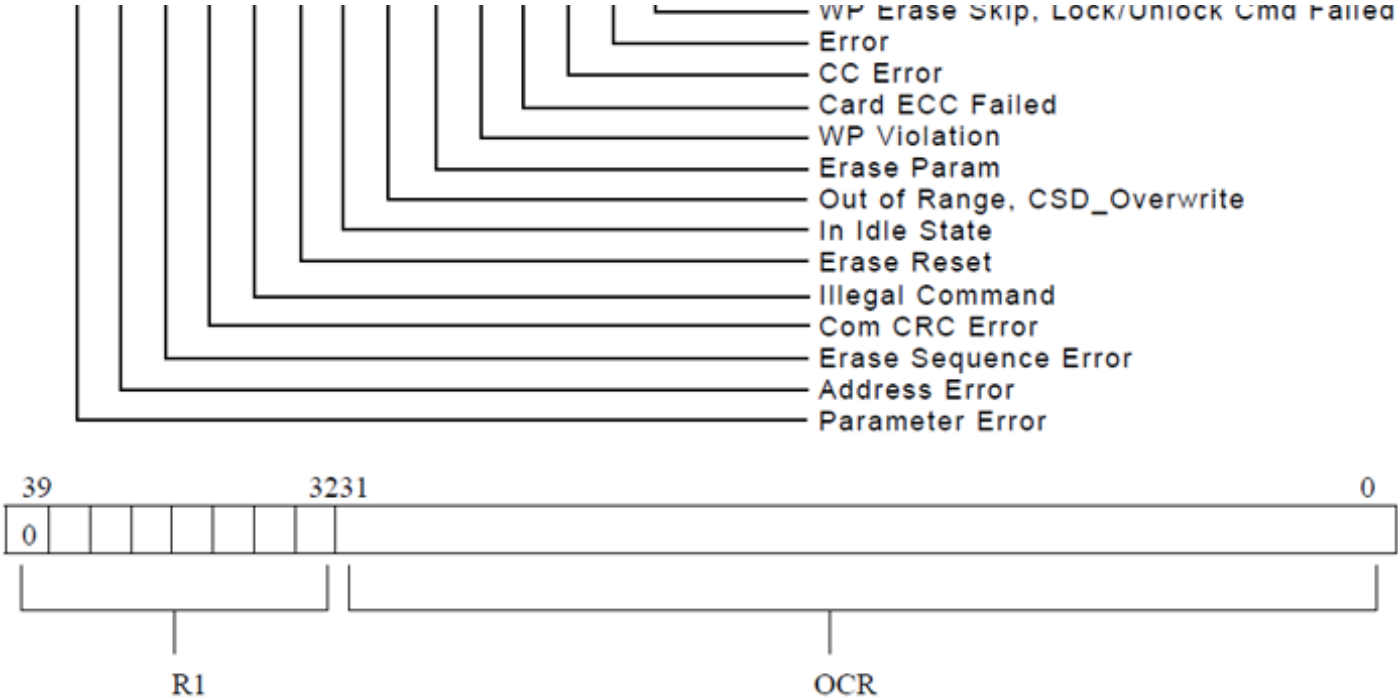


READ_SINGLE_BLOCK in such a way that this command needs to be send only once and the consecutive 512 bytes memory locations will be read continuously.

COMMANDS FOR INITIALIZING THE MEMORY CARD



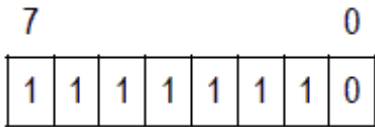
Before the memory card can respond to these commands, the memory card should be



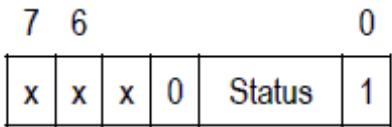
initializes in SPI mode. Certain commands should be send one after the other to initialize the SD card. The SD card will be in SD interfacing mode on reset. Hence the first command send to the SD card should have the correct CRC byte included.

The steps required to initialize the SD card in the SPI mode using commands are as follows:

- First byte: Start Block.



- Bytes 2-513 (depends on the data block length): User data.
- Last two bytes: 16-bit CRC.



The meaning of the status bits is defined as follows:

- '010'—Data accepted.
- '101'—Data rejected due to a CRC error.
- '110'—Data Rejected due to a Write Error

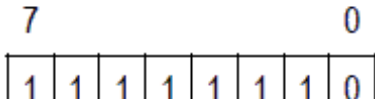
Allocation Table 32, means it has a file allocation table of length 32 bits.

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

- First byte: Start Block.



Once the SD card is initialized, we can start using the read and write commands. The commands from the SD commands layer like READ_SINGLE_BLOCK, READ_MULTIPLE_BLOCK, WRITE_BLOCK etc. are mostly used to interact with the File System layer.

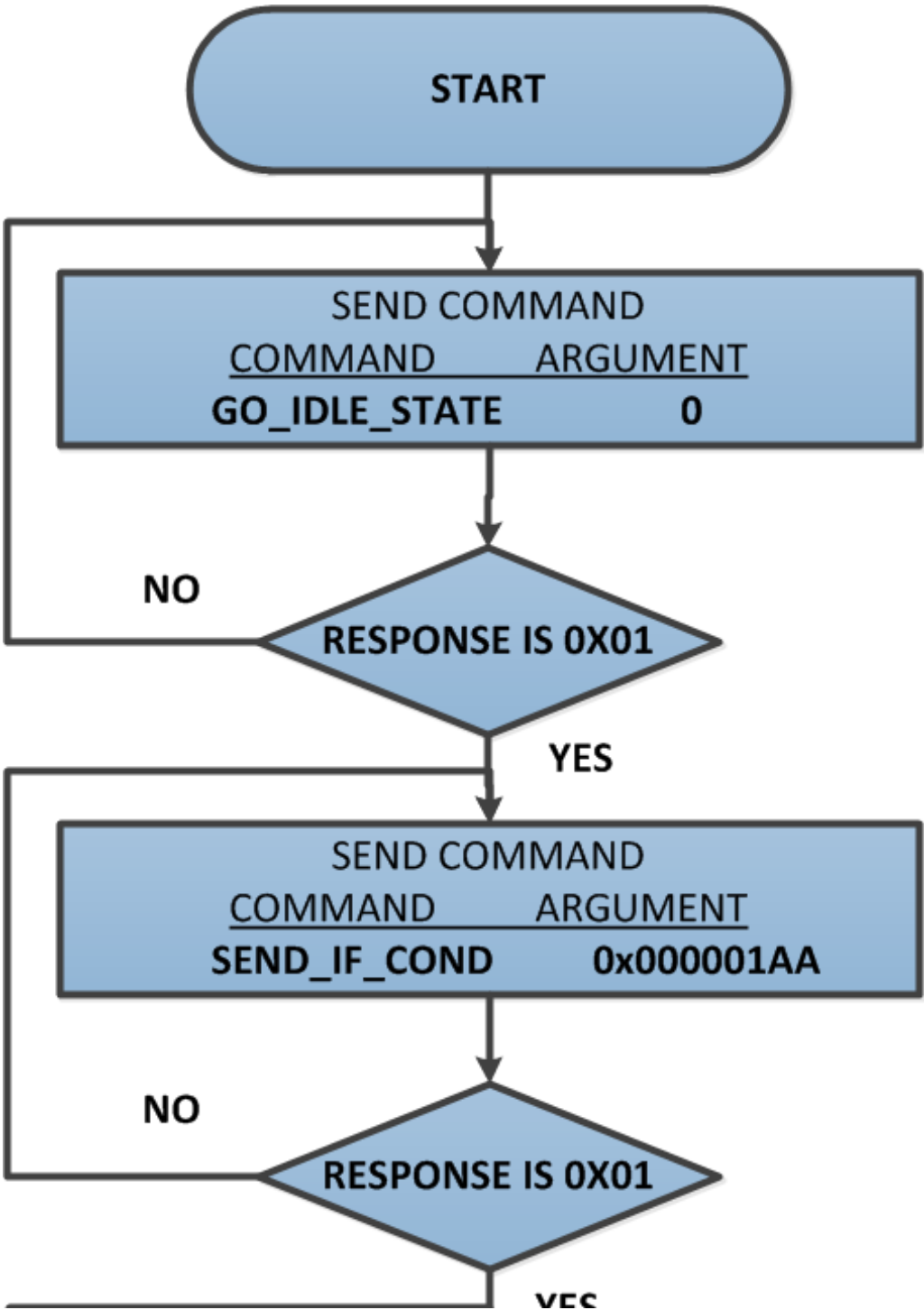
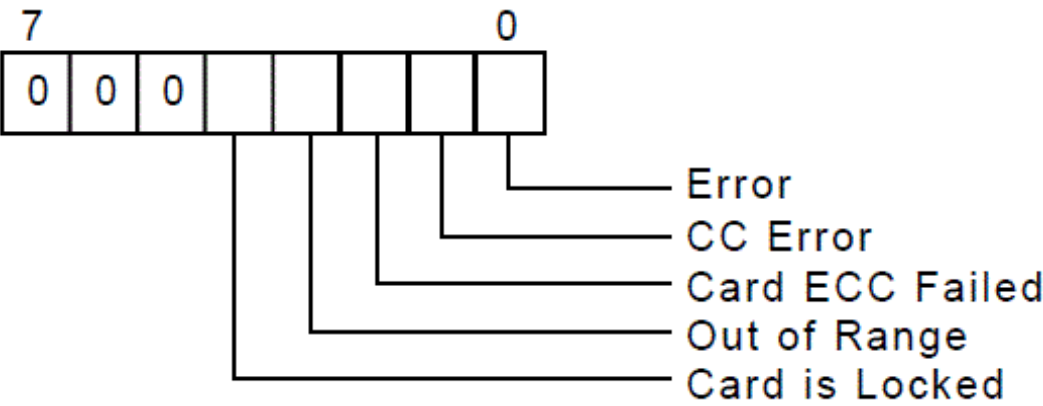
The following section explains the features of FAT32 file system and shows how the data can be read from such a file system implementation.

{C}{C}{C}{C}{C}{C}{C}{C} 3) {C}{C}{C}{C}{C}{C}{C}{C}
{C}{C}{C}{C}{C}{C}{C}{C} **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

has 512 bytes per Sector. The grouping of memory bytes to form Sectors is shown in the following figure.

- Bytes 2-513 (depends on the data block length): User data.
- Last two bytes: 16-bit CRC.



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.

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.

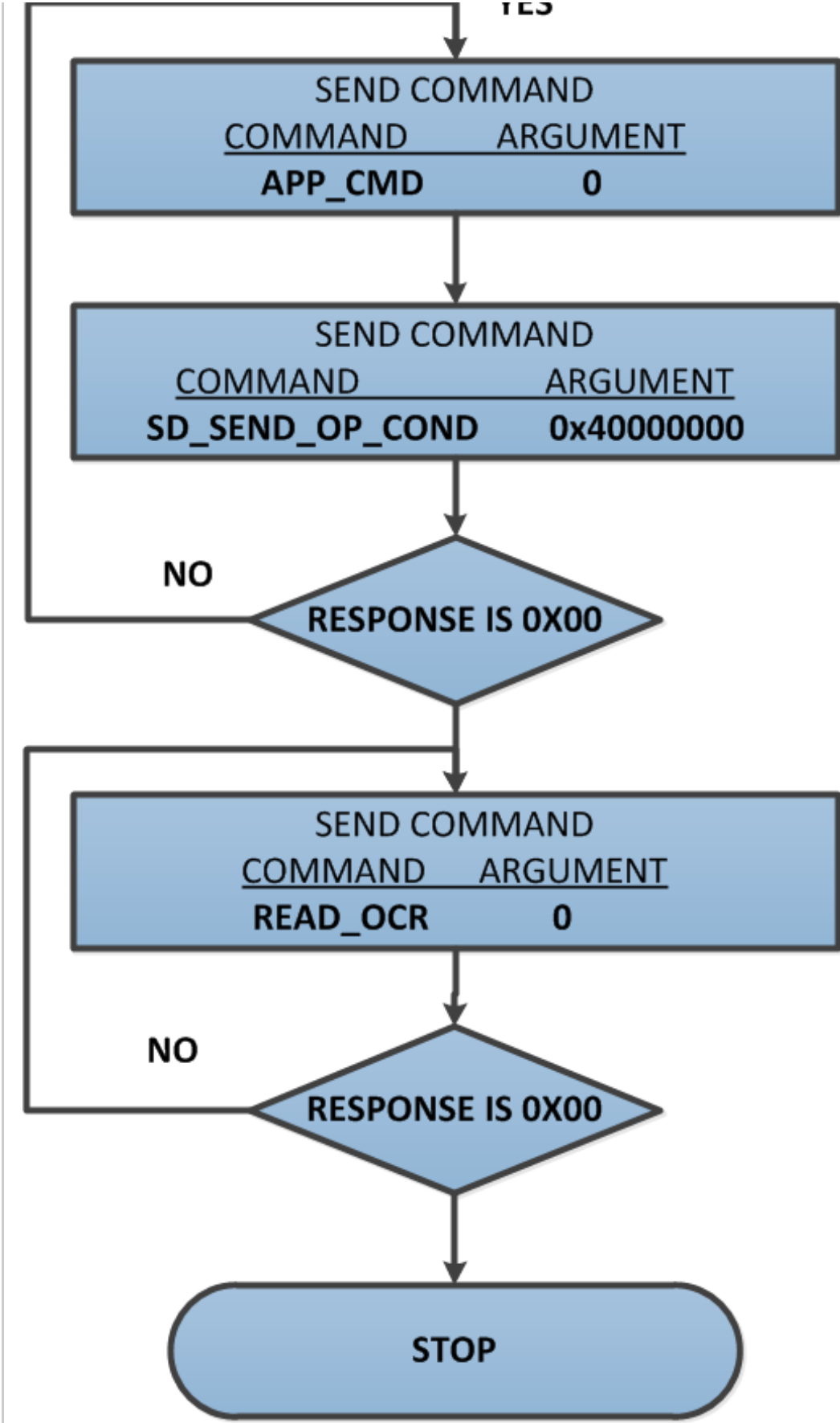
FAT32 FILE SYSTEM FORMAT

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:

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.

MBR

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 MBR is one sector long and the important bytes of the MBR are shown below.



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:

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.

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).

FS INFO SECTOR

The Sector number of the FS Info Sector is obtained from the 47th and 48th byte of the Boot Sector. It holds the current status of the Clusters. It can hold the value of total number of free





clusters at the moment and also the Cluster number of the next free Cluster inside the file system as shown below:

FS 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:

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:

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

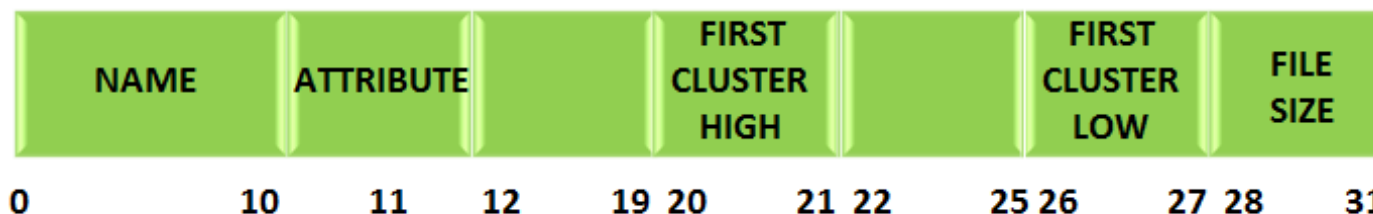
byte holds the lower bytes of the



First Cluster number of the File. The last four bytes hold the size of the file.

0 487 488 491 492 495 496 511

THE LOGIC FOR READING A FILE FROM FAT32 FILE SYSTEM



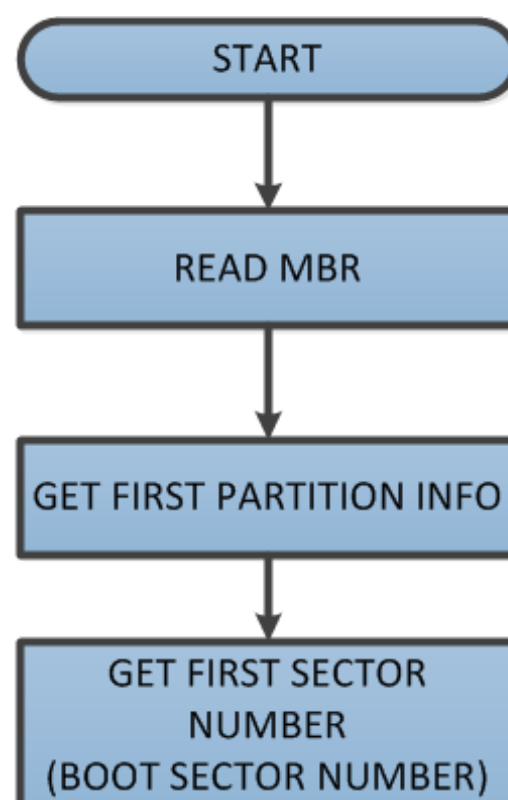
A File with a specific name can be read from the FAT32 formatted file system using the logic shown below; Take a closer look and it can be found that

Attrib Bit	Function	LFN	Comment
0 (LSB)	Read Only	1	Should not allow writing
1	Hidden	1	Should not show in dir listing
2	System	1	File is operating system
3	Volume ID	1	Filename is Volume ID
4	Directory	x	Is a subdirectory (32-byte records)
5	Archive	x	Has been changed since last backup
6	Ununsed	0	Should be zero
7 (MSB)	Ununsed	0	Should be zero

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.

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

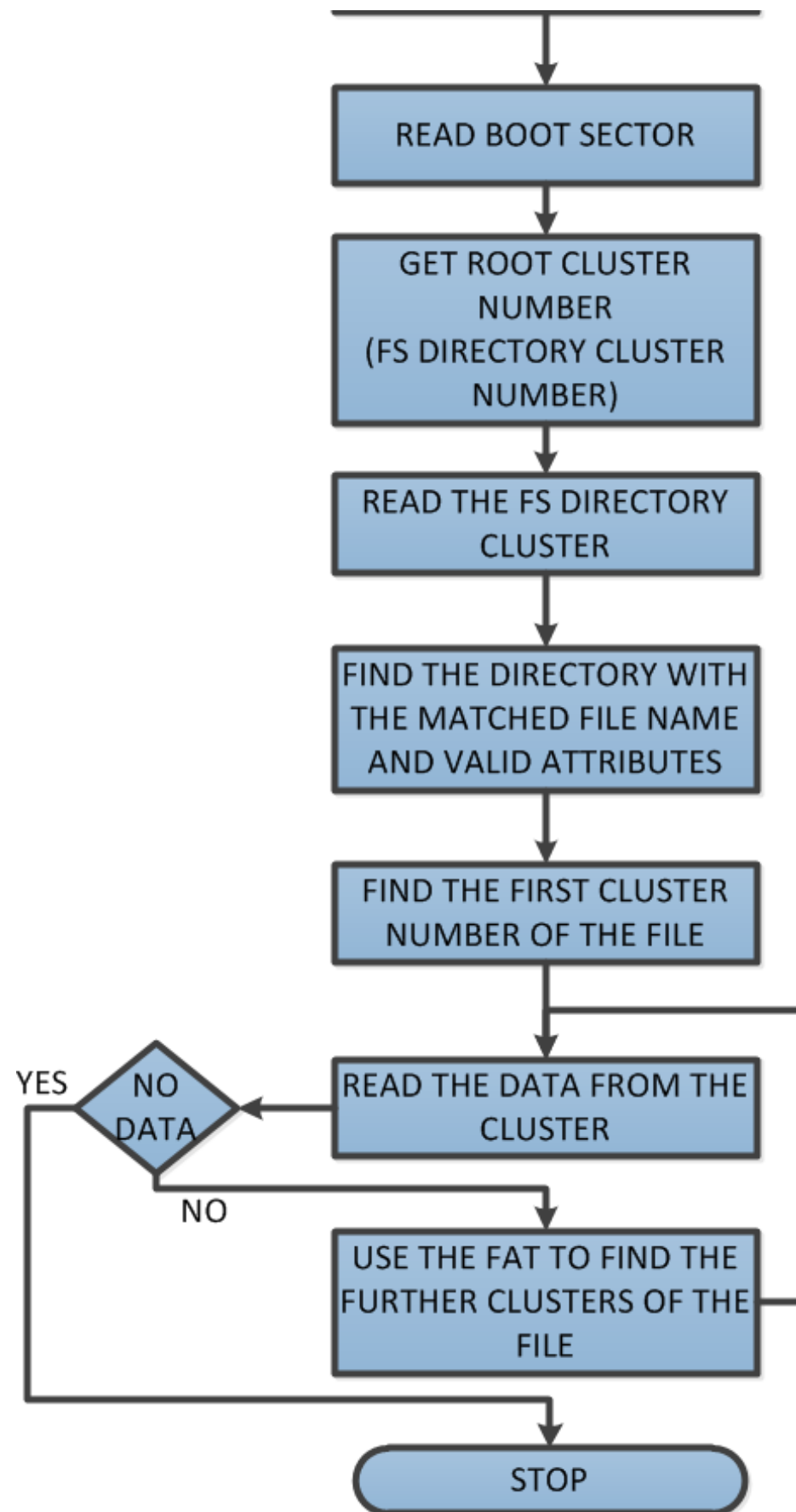
FAT SECTOR NUMBER FOR NEXT CLUSTER POINTER = FIRST SECTOR NUMBER OF THE PARTITION

+ 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.



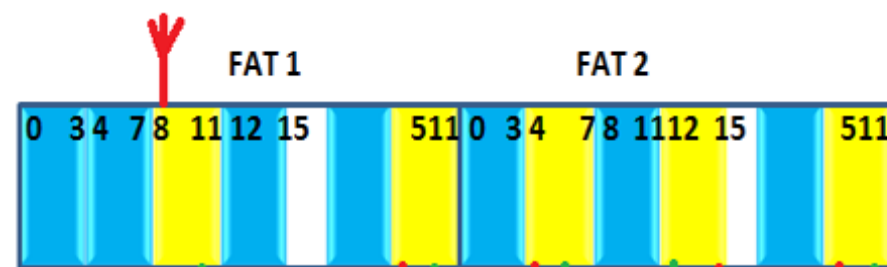
The yellow indicates Clusters having the particular file's data and the corresponding Cluster pointers inside the FAT32. The red lines indicate finding the next Cluster pointer corresponding to the current Cluster and the green line indicate finding the next Cluster using the Cluster number stored inside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

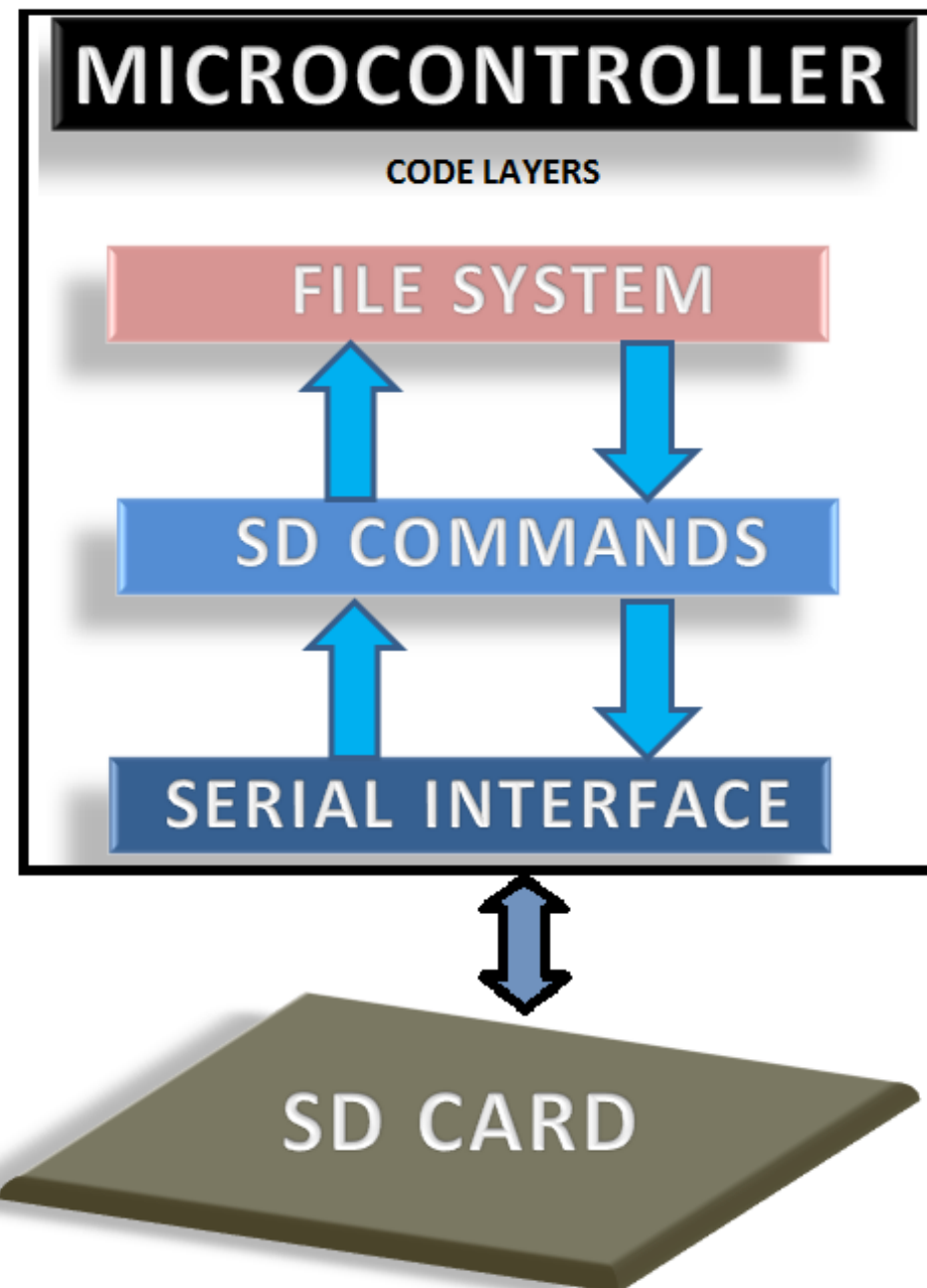
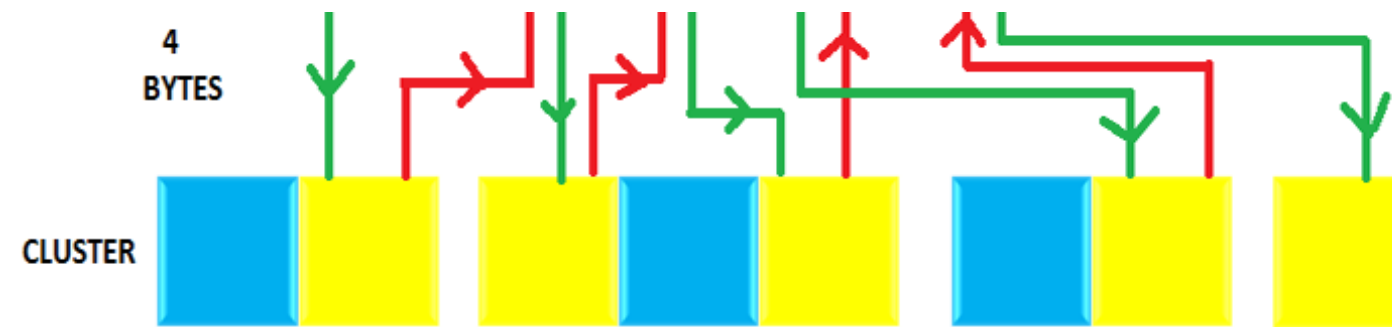
THE CODING DETAILS

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:

The microcontroller 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 microcontroller 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.





Like 38

g+1 2

Tweet 2

Find Parts Fast

Find Electronic Components Up-To-Date pricing and inventory



VCON Controls& Components

Terminal Blocks/ Strips, Enclosures Barrier Single/Double Row EuroStyle



Copyright © 2012 EngineersGarage. All rights reserved. **Privacy Policy (<http://www.engineersgarage.com/privacy-policy>)** | **Refund Policy (<http://www.engineersgarage.com/subscriptions-refund-policy>)** | **About Us (<http://www.engineersgarage.com/about-us>)**