How to Use MMC/SDC
Copied from : http://elm-chan.org/docs/mmc/mmc_e.html
References http://www.sdcard.org/home
http://users.ece.utexas.edu/~valvano/EE345M/SD_Physical_Layer_Spec.pdf

Introduction

The *Secure Digital Memory Card* (Figure 1) is the de facto standard memory card for mobile equipments. The SDC was developed as upward-compatible to *Multi Media Card* (also in Figure 1) so that the SDC-compliant equipments can also use an MMC with an appropriate adapter. There are also reduced size versions, such as *RS-MMC, miniSD* and *microSD*, with same functionality. The MMC/SDC has a microcontroller in it, the flash memory controls (erasing, reading, writing and error controls) are completed inside of the memory card. The data is transferred between the memory card and the host controller as data blocks in unit of 512 bytes, so that it can be seen like a generic hard disk drive from view point of upper level layers. The file system for the memory card is FAT12/16 with FDISK partitioning rule. The FAT32 is defined for only high capacity (>= 4G) cards.

Figure 1. Secure digital card and a multimedia card.

SDC-Connector

Figure 2 shows the SDC for Lab 5. The SDC has nine contact pads. The three of the contacts are assigned for power supply so that the number of effective signals are six. Therefore the data transfer between the host and the card is done via a synchronous serial interface. The working supply voltage range is indicated in a special function register and it should be read and confirmed the operating voltage range. However, the supply voltage can be fixed to a proper value because the SDC works at supply voltage of *2.7 to 3.6 volts*. The current consumption can reach up to 15 mA in standby and 50 mA during operation. Combo cards can draw up to 100 mA during operation.

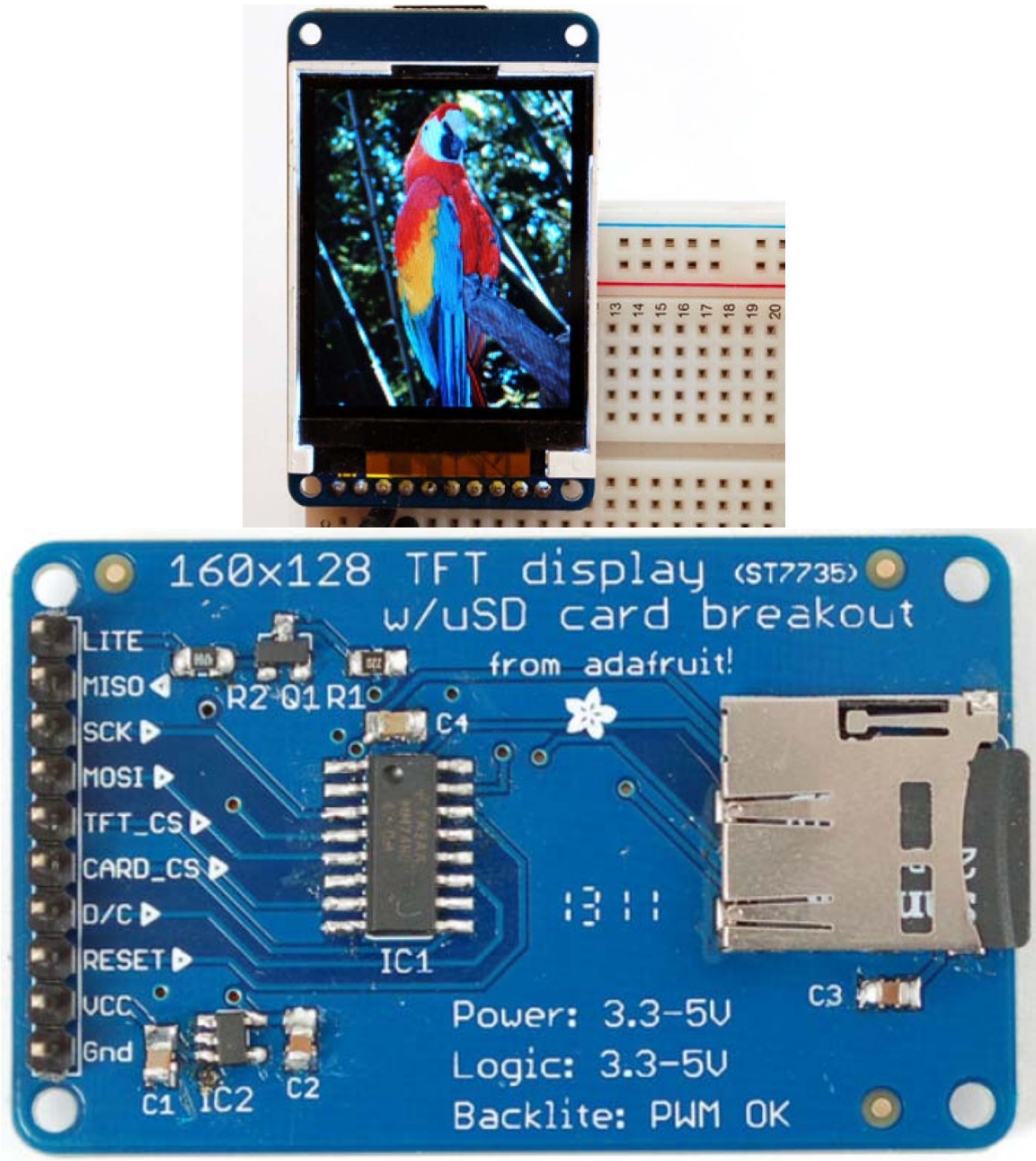


Figure 2. SDC and connector pin-out.

Figure 3a shows the interface logic between the STM32F103 and the SDC. The IRQ line, pin 8, is not connected. PA4 is connected to pin 1 which is card select. PA8 is used to detect if a card is inserted into the SD slot.

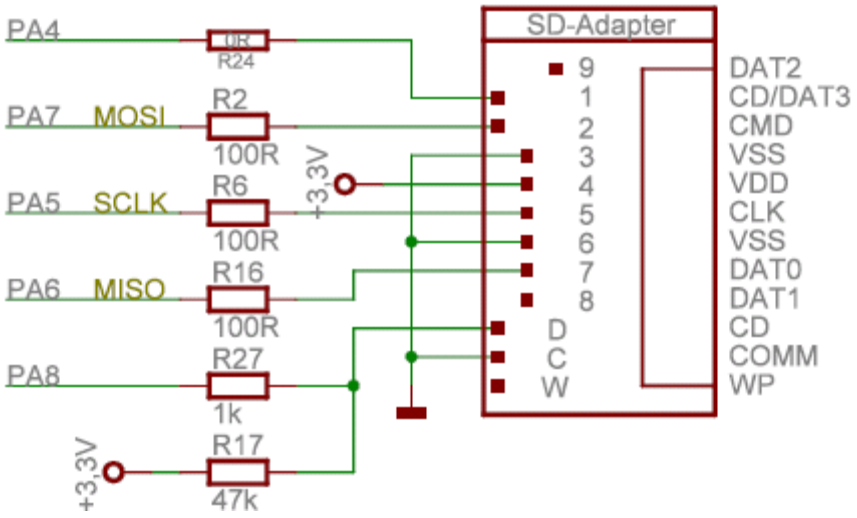


Figure 3a. SDC interface to STM32F103.

Figure 3b shows the interface logic between the LM3S8962 and the SDC. The IRQ line, pin 8, is not connected to the microcontroller, and pulled high. Pin 1 is also pulled high, which is card select. The four SPI signals are connected to the microcontroller.

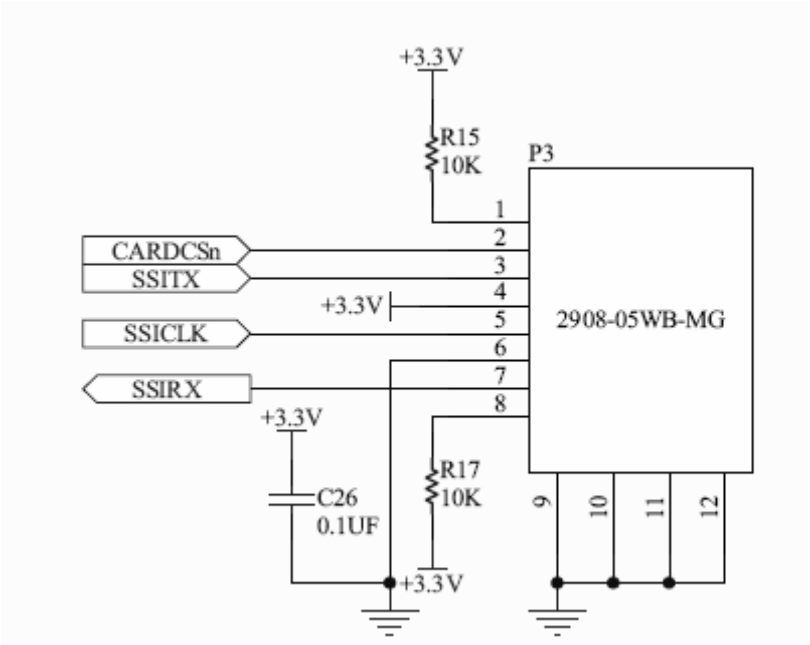


Figure 3b. SDC interface to LM3S8962.

The following shows the interface logic between the TM4C123 and the SDC. The IRQ line, pin 8, is not connected to the microcontroller, and pulled high. Pin 1 is also pulled high, which is card select. The four SPI signals are connected to the microcontroller.

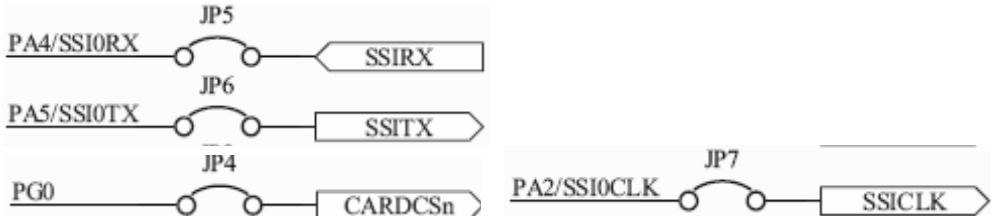
// Backlight (pin 10) connected to +3.3 V

// MISO (pin 9)) connected to PA4 (SSI0Rx) <- new
// SCK (pin 8) connected to PA2 (SSI0Clk)
// MOSI (pin 7) connected to PA5 (SSI0Tx)
// TFT_CS (pin 6) connected to PA3 (SSI0Fss)
// CARD_CS (pin 5) connected to a GPIO <- new
// Data/Command (pin 4) connected to PA6 (GPIO)
// RESET (pin 3) connected to PA7 (GPIO)
// VCC (pin 2) connected to +3.3 V
// Gnd (pin 1) connected to ground
Figure 3c. SDC interface to TM4C123.

SPI Mode Interface of the SD card

SPI (Serial Peripheral Interface) is one of the on-board inter-IC communication interfaces. It was introduced by Motorola, Inc. (Freescale Semiconductor). Because of its simplicity and generality, it is incorporated in many peripheral ICs similar to the Philips *IIC-bus*. The number of signals of SPI, three or four wires, is larger than IIC's two wires, but the transfer rate can rise up to 20 Mbps or higher depends on device's ability (5 - 50 times faster than IIC). Therefore, it is preferable for applications, such as ADC, DAC, or communication, which require high data transfer rate. The basic structure of the SPI is shown in Figure 4. The master IC and the slave IC are tied with three signal lines, SCLK (Serial Clock), MISO (Master-In Slave-Out) and MOSI (Master-Out Slave-In), and contents of both 8-bit shift registers are exchanged with the shift clock driven by master IC. Additionally an SS (Slave Select) signal can be used to synchronize the start of packet or byte boundary, and multi-slave configurations. Most slave ICs assign different pin names, such as DI, DO and CS, to the SPI interface. For one-way transfer device, such as DAC and single channel ADC, only one data line may be used. The data bits are shifted MSB first.

When interfacing multiple slaves to a single master, the slave ICs can be attached in parallel and separate CS signals from master IC are connected to each slave ICs. The data output of slave IC that selected by which CS signal is enabled and deselected devices are disconnected from MISO line.



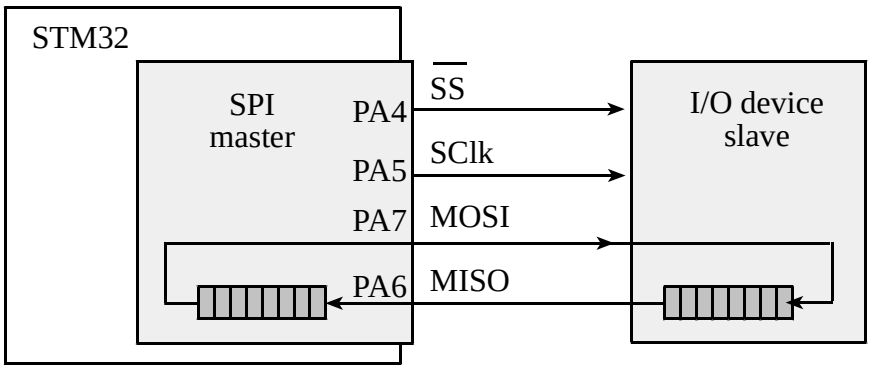


Figure 209. Single master/ single slave application

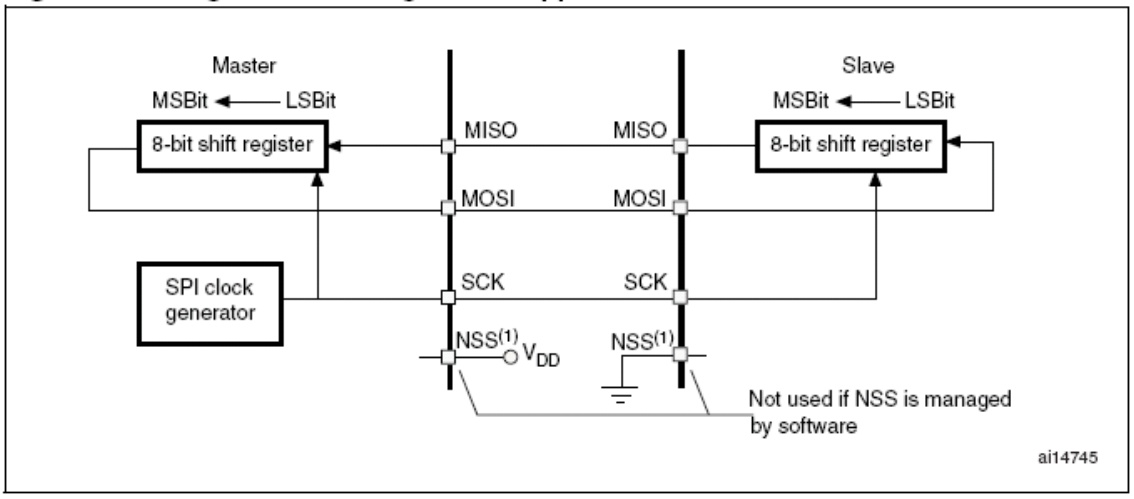


Figure 4. SPI block diagrams.

In SPI, data shift and data latch are done opposite clock edges respectively. There is an advantage that when shift and latch operations are separated, critical timing between two operations can be avoided. Therefore, timing consideration for IC design and board design can be relieved. But on the other hand there are four operation modes due to combination of clock polarity and clock phase, master IC must configure its SPI interface as an SPI mode that slave IC required. The SD card uses CPOL=0, CPHA=0 mode as shown in Figure 5.

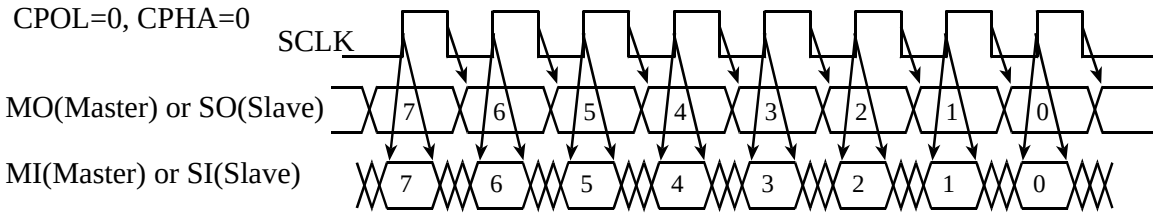


Figure 5. SPI CPOL= 0, SPHA=0 mode.

There are three possible modes to interface the SD card, as shown in Table 1. The communication protocol for the SPI mode is simple compared to the native modes. The MMC/SDC can be attached to the most microcontrollers via a generic SPI interface or GPIO ports. Therefore the SPI mode is suitable for low cost embedded applications.

Especially, there is no reason to attempt to use native mode with a cheap microcontroller that has no native MMC/SDC interface. For SDC, the 'SPI mode 0' is defined for its SPI mode. Thus the *SPI Mode 0* (CPHA=0, CPOL=0) is the proper setting for MMC/SDC interface, but SPI mode 3 also works as well in most cases.

Pin	SD 4-bit mode		SD 1-bit mode		SPI mode	
1	CD/DAT[3]	Data line 3	N/C	Not Used	CS	Card Select
2	CMD	Command line	CMD	Command line	DI	Data input
3	VSS1	Ground	VSS1	Ground	VSS1	Ground
4	VDD	Supply voltage	VDD	Supply voltage	VDD	Supply voltage
5	CLK	Clock	CLK	Clock	SCLK	Clock
6	VSS2	Ground	VSS2	Ground	VSS2	Ground
7	DAT[0]	Data line 0	DATA	Data line	DO	Data output
8	DAT[1]	Data line 1 or Interrupt (optional)	IRQ	Interrupt	IRQ	Interrupt
9	DAT[2]	Data line 2 or Read Wait (optional)	RW	Read Wait (optional)	NC	Not Used

Table 1. Three SD interfacing modes.

Command and Response

In SPI mode, the data direction on the signal line is fixed and the data is transferred in byte oriented serial communication. The command frame from host to card is a fixed length (six bytes) packet that shown below. When a command frame is transmitted to the card, a response to the command (R1, R2 or R3) will able to be read from the card. Because data transfer is driven by serial clock generated by host, the host must continue to read bytes, send a 0xFF and get the received data, until receive any valid response. The command response time (NCR) is 0 to 8 bytes for SDC, 1 to 8 bytes for MMC. The CS signal must be held low during a transaction (command, response and data transfer if exist). The CRC field is optional in SPI mode, but it is required as a bit field to compose a command frame. The DI signal must be kept high during read transfer.

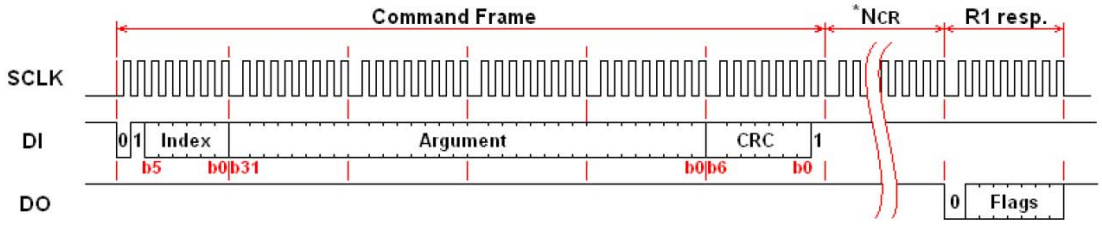


Figure 6. SD command frame.

SPI Command Set

Each command is expressed in abbreviation like

GO_IDLE_STATE or CMD<n>

where <n> is the number of the command index and the value can be 0 to 63. Table 2 describes only commands that to be usually used for generic read/write and card

initialization. For details on all commands, please refer to spec sheets from MMCA and SDCA.

Command Index	Argument	Response	Data	Abbreviation	Description
CMD0	None(0)	R1	No	GO_IDLE_STATE	Software reset.
CMD1	None(0)	R1	No	SEND_OP_COND	Initiate initialization process.
ACMD41(*1)	*2	R1	No	APP_SEND_OP_COND	For only SDC. Initiate initialization process.
CMD8	*3	R7	No	SEND_IF_COND	For only SDC V2. Check voltage range.
CMD9	None(0)	R1	Yes	SEND_CSD	Read CSD register.
CMD10	None(0)	R1	Yes	SEND_CID	Read CID register.
CMD12	None(0)	R1b	No	STOP_TRANSMISSION	Stop to read data.
CMD16	Block length[31:0]	R1	No	SET_BLOCKLEN	Change R/W block size.
CMD17	Address[31:0]	R1	Yes	READ_SINGLE_BLOCK	Read a block.
CMD18	Address[31:0]	R1	Yes	READ_MULTIPLE_BLOCK	Read multiple blocks.
CMD23	Number of blocks[15:0]	R1	No	SET_BLOCK_COUNT	For only MMC. Define number of blocks to transfer with next multi-block read/write command.
ACMD23(*1)	Number of blocks[22:0]	R1	No	SET_WR_BLOCK_ERASE_COUNT	For only SDC. Define number of blocks to pre-erase with next multi-block write command.
CMD24	Address[31:0]	R1	Yes	WRITE_BLOCK	Write a block.
CMD25	Address[31:0]	R1	Yes	WRITE_MULTIPLE_BLOCK	Write multiple blocks.
CMD55(*1)	None(0)	R1	No	APP_CMD	Leading command of ACMD<n> command.
CMD58	None(0)	R3	No	READ_OCR	Read OCR.

*1:ACMD<n> means a command sequence of CMD55-CMD<n>.
*2: Rsv(0)[31], HCS[30], Rsv(0)[29:0]
*3: Rsv(0)[31:12], Supply Voltage(1)[11:8], Check Pattern(0xAA)[7:0]

Table 2. SD commands.

SPI Response

There are three command response formats, *R1*, *R2* and *R3*, depending on the command index. A byte of response R1 is returned for most commands. The bit field of R1 response is shown in Figure 7, the value 0x00 means successful. When any error occurred, corresponding status bit in the response will be set. The R3 response (R1 and trailing 32-bit OCR) is for only CMD58.

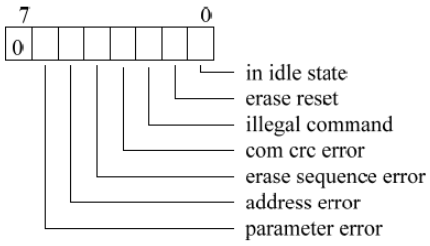


Figure 7-9: R1 Response Format

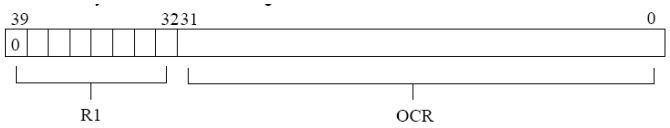


Figure 7-11: R3 Response Format

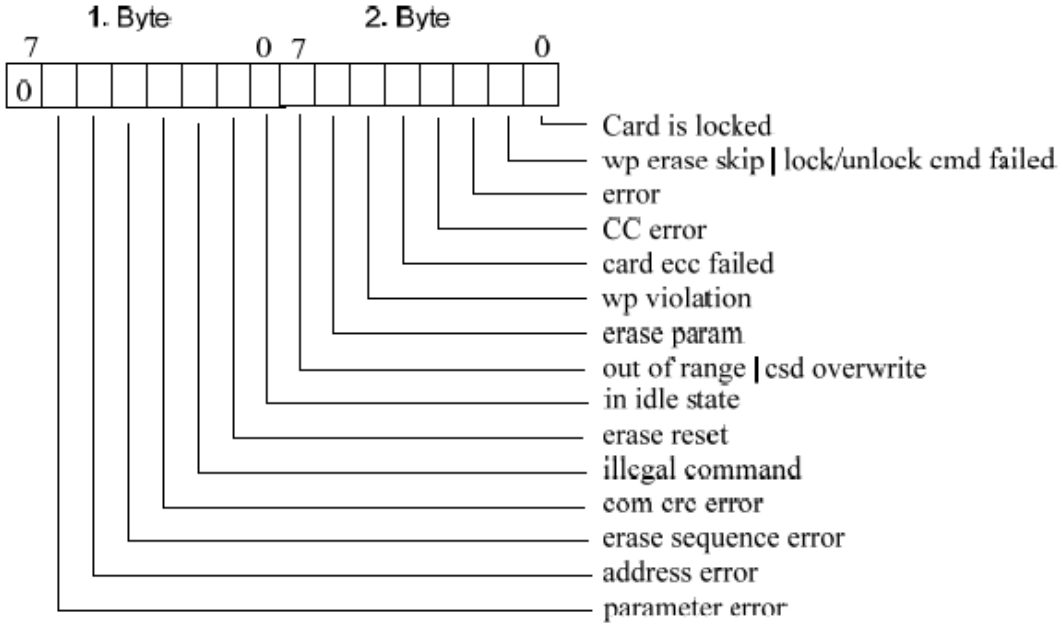


Figure 7-10: R2 Response Format

Figure 7. SD response frames.

Some commands take a time longer than NCR and it responds *R1b*. It is an R1 response followed by busy flag (DO is held low as long as internal process is in progress). The host controller should wait for end of the process until 0xFF is received.

Initialization Procedure for SPI Mode

After power on reset, MMC/SDC enters its native operating mode. To put it SPI mode, following procedure must be performed.

1) Power ON (Insertion)

After supply voltage reached 2.2 volts, wait for a millisecond at least. *Set DI and CS high and apply more than 74 clock pulses to SCLK* and the card will go ready to accept native commands.

2) Software Reset
Set SPI clock rate between 100 kHz and 400 kHz and then send a *CMD0* with CS low to reset the card. The card samples CS signal when a CMD0 is received. If the CS signal is low, the card enters SPI mode. Since the CMD0 must be sent as a native command, the CRC field must have a valid value. When once the card enters SPI mode, the CRC feature is disabled and the CRC is not checked, so that command transmission routine can be written with the hardcoded CRC value that valid for only CMD0 and CMD8. When the CMD0 is accepted, the card will enter idle state and respond R1 response with In Idle State bit (0x01). The CRC feature can also be switched with CMD59.

3) Initialization
In idle state, the card accepts only CMD0, CMD1, ACMD41 and CMD58. Any other commands will be rejected. In this time, read OCR with CMD58, check working voltage range indicated by the OCR. In case of the system supply voltage is out of working voltage range, the card must be rejected. Note that all cards work at voltage range of 2.7 to 3.6 volts at least. The card initiates initialization when a *CMD1* is received. To poll end of the initialization, the host controller must send CMD1 and check the response until end of the initialization. When the card is initialized successfully, In Idle State bit in the R1 response is cleared (R1 resp changes 0x01 to 0x00). The initialization process can take *hundreds of milliseconds* (large cards tend to longer), so that this is a consideration to determine the time out value. After the In Idle State bit cleared, generic read/write commands will able to be accepted. Because *ACMD41* instead of CMD1 is recommended for SDC, trying ACMD41 first and retry with CMD1 if rejected, is ideal to support both type of the cards. The SPI clock rate should be changed to fast as possible to optimize the read/write performance. The TRAN_SPEED field in the CSD indicates the maximum clock rate of the card. The maximum clock rate is 20MHz for MMC, 25MHz for SDC in most case. Note that the clock rate will able to be fixed to 20/25MHz in SPI mode because there is no open-drain condition that restricts the clock rate. The initial block length can be set larger than 512 on 2GB card, so that the block size should be re-initialized with CMD16 if needed.

How to support SDC Ver2 and high capacity cards
After the card enters idle state with a CMD0, send a *CMD8* with argument of 0x000001AA and correct CRC prior to initialization process. When the CMD8 is rejected with an illegal command error (0x05), the card is SDC V1 or MMC. When the CMD8 is accepted, R7 response (R1(0x01) and trailing 32 bit data) will be returned. The lower 12 bits in the return value 0x1AA means that the card is SDC V2 and it can work at voltage range of 2.7 to 3.6 volts. If not the case, the card must be rejected. And then initiate initialization with ACMD41 with HCS (bit 30). After the initialization completed, read OCR and check CCS (bit 30) in the OCR. When it is set, subsequent data read/write operations that described below are commanded in block address instead of byte address. The block size is always fixed to 512 bytes.

Data Transfer
1) Data Packet and Data Response

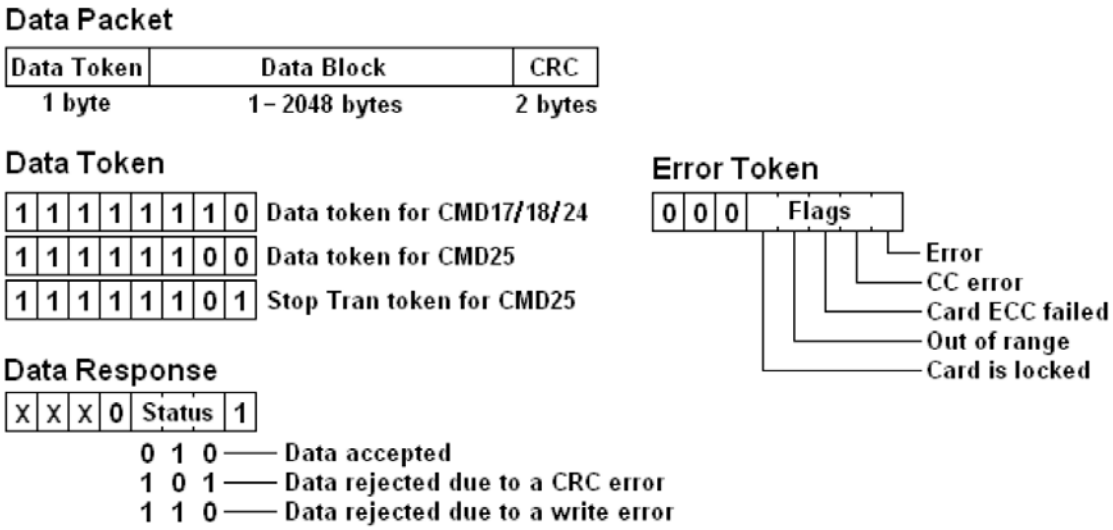


Figure 8. SD data packets.

In a transaction with data transfer, one or more data blocks will be sent/received after command response. The data block is transferred as a data packet that consists of Token, Data Block and CRC. The format of the data packet is shown in Figure 8 and there are three data tokens. As for Stop Tran token that means end of multiple block write, it is used in single byte without data block and CRC.

2) Single Block Read

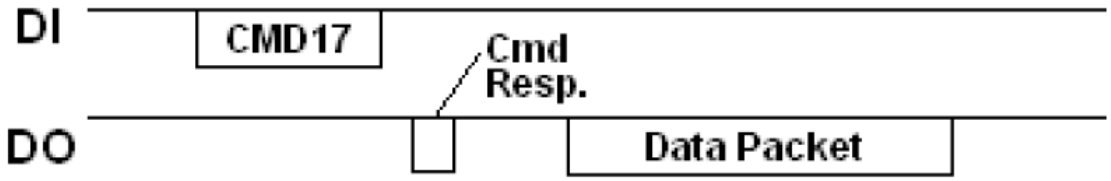
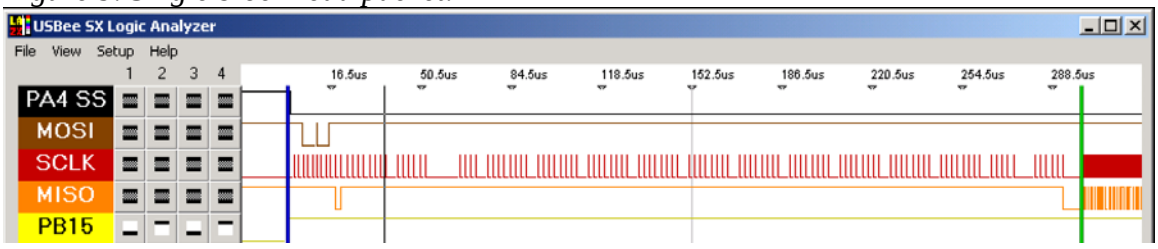
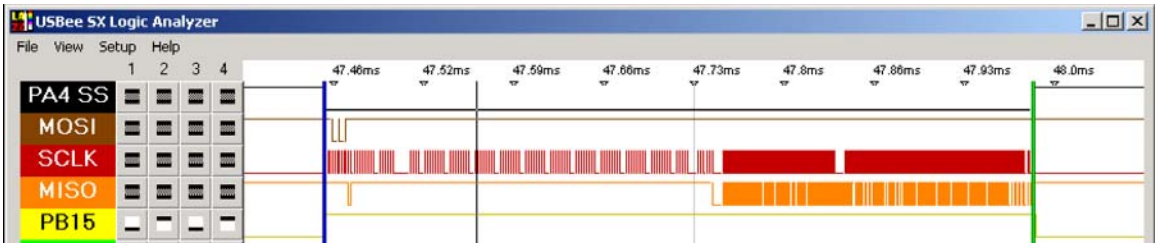


Figure 9. Single block read packet.



It took 300 μs to set up for read (logic analyzer resolution too slow for SCLK)



It took 535 μs to read one block (logic analyzer resolution too slow for SCLK)

The argument specifies the location to start to read *in unit of byte or block*. The sector address specified by upper layer must be scaled properly. When a CMD17 is accepted, a read operation is initiated and the read data block will be sent to the host. After a valid data token is detected, the host controller receives following data field and two byte CRC. The CRC bytes must be flushed even if it is not needed. If any error occurred during the read operation, an error token will be returned instead of data packet.

3) Multiple Block Read

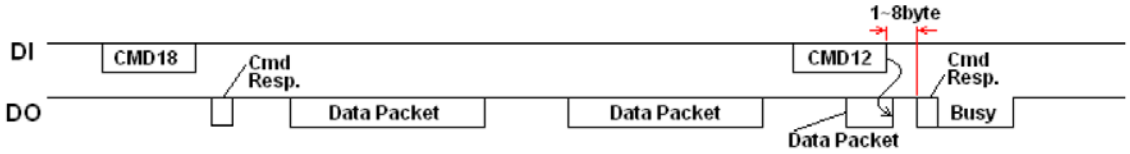


Figure 10. Multiple block read packet.

The Multiple Block Read command reads multiple blocks in sequence from the specified address. When number of transfer blocks has not been specified before this command, the transaction will be initiated as an open-ended multiple block read, the read operation will continue until stopped with a CMD12. The received byte immediately following CMD12 is a stuff byte, it should be discarded before receive the response of the CMD12.

4) Single Block Write

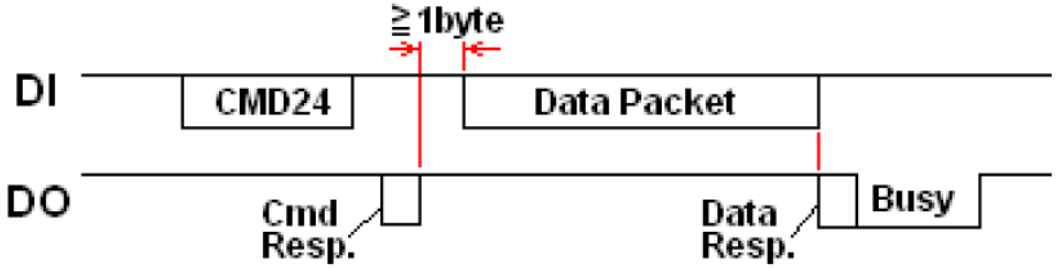


Figure 11. Single block write packet.

It took 272 μs to write one 512 byte block. (logic analyzer resolution too slow for SCLK)

When a write command is accepted, the host controller sends a data packet to the card after a byte space. The packet format is same as Block Read command. The CRC field

can have any invalid value unless the CRC function is enabled. When a data packet has been sent, the card responds a Data Response immediately following the data packet. The data response trails a busy flag to process the write operation. Most cards cannot change write block size and it is fixed to 512. In principle of the SPI mode, the CS signal must be asserted during a transaction; however there is an exception to this rule. When the card is busy, the host controller can de-assert CS to release SPI bus for any other SPI devices. The card will drive DO signal low again when reselect it during internal process is in progress. Therefore a preceding busy check (wait ready immediately before command and data packet) instead of post wait can eliminate waste wait time. In addition the internal process is initiated a byte after the data response, this means eight clocks are required to initiate internal write operation. The state of CS signal during the eight clocks is negligible so that it can done by bus release process described below.

5) Multiple Block Write

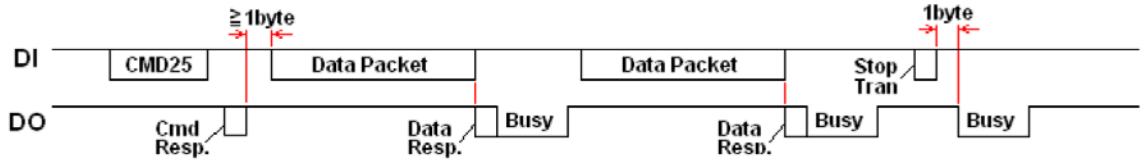


Figure 12. Multiple block write packet.

The Multiple Block Write command writes multiple blocks in sequence from the specified address. When number of transfer blocks has not been specified prior to this command, the transaction will be initiated as an *open-ended multiple block write*, the write operation will continue until it is terminated with a Stop Tran token. The busy flag will appear on the DO line a byte after the Stop Tran token. As for SDC, the multiple block write transaction must be terminated with a Stop Tran token independent of the transfer type, pre-defined or open-ended.

Reading CSD and CID

These are same as Single Block Read except for the data block length. The CSD and CID are sent to the host as *16 byte data block*. For details of the CMD, CID and OCR, refer to http://users.ece.utexas.edu/~valvano/EE345M/SD_Physical_Layer_Spec.pdf.

Consideration to Bus Floating and Hot Insertion

Any signals that can be floated should be pulled low or high properly via a resister. This is a generic design rule on MOS devices. Because DI and DO are normally high, they should be pulled-up. According to SDC/MMC specs, from 50k to 100k ohms is recommended to the value of pull-up registers. However the clock signal is not mentioned in the SDC/MMC specs because it is always driven by host controller. When there is a possibility of floating, it should be pulled to the normal state, low. The MMC/SDC can hot insertion/removal but some considerations to the host circuit are needed to avoid an incorrect operation. For example, if the system power supply (Vcc) is tied to the card socket directly, the Vcc will dip at the instant of contact closed due to a charge current to the capacitor that built in the card. 'A' in the right image is the scope view and it shows that occurring a voltage dip of about 600 millivolts. This is a sufficient level to trigger a brown out detector. 'B' in the right image shows that an inductor is

inserted to block the surge current, the voltage dip is reduced to 200 mV. A low ESR capacitor, such as OS-CON, can eliminate the voltage dip drastically like shown in 'C'. However the low ESR capacitor can cause an oscillation of LDO regulator.

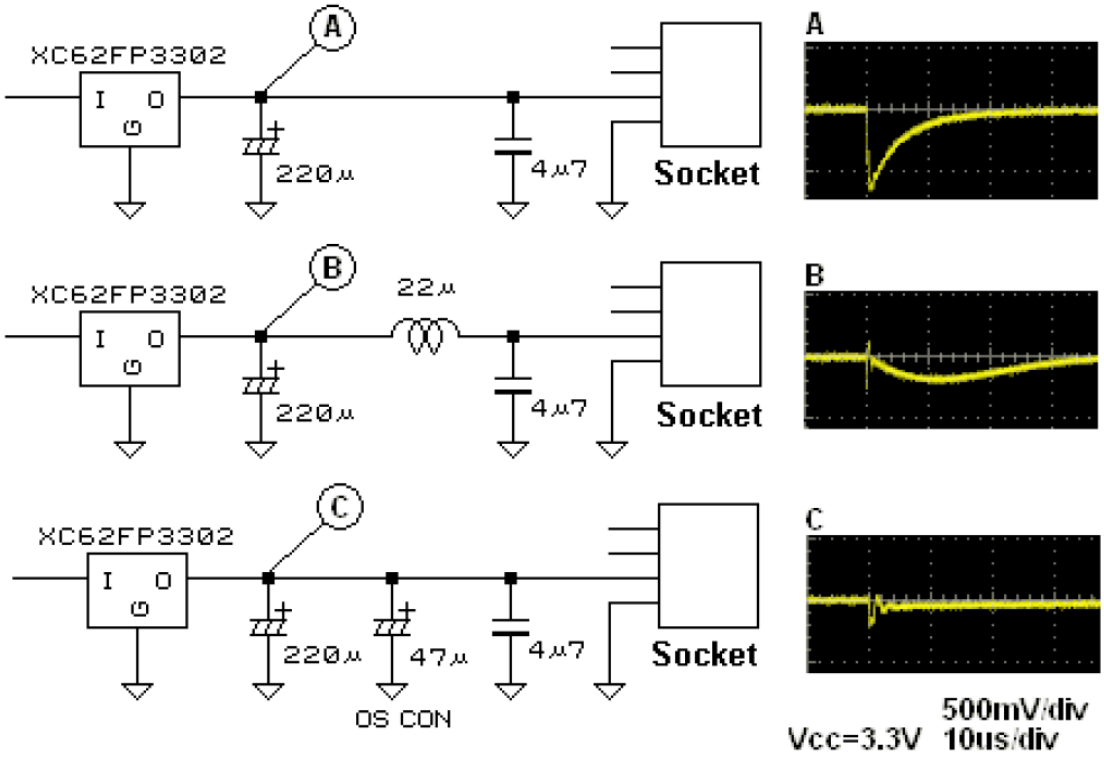


Figure 13. Support for hot insertion.

Consideration on Multi-slave Configuration

In the SPI bus, each slave device is selected with separated CS signals, and plural devices can be attached to an SPI bus. Generic SPI slave device drives/releases its DO signal by CS signal asynchronously to share an SPI bus. However MMC/SDC drives/releases DO signal in *synchronizing to the SCLK*. This means there is a possibility of bus conflict with MMC/SDC and any other SPI slaves that attached to an SPI bus. Right image shows the drive/release timing of the MMC/SDC (the DO signal is pulled to 1/2 Vcc to see the bus state). Therefore to make MMC/SDC release DO signal, the master device must send a byte after CS signal is de-asserted.

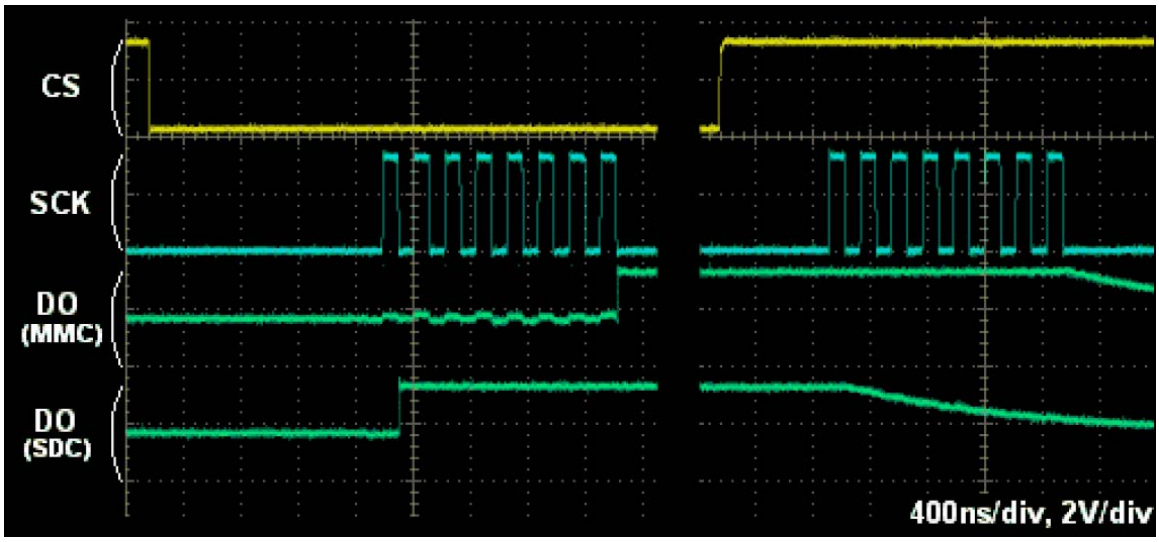


Figure 14. Multi-slave configuration.

Optimization of Write Performance (http://elm-chan.org/docs/sm_e.html)

Most MMC/SDC employs NAND Flash Memory as a memory array. The NAND flash memory is cost effective and it can read/write *large* data fast, but on the other hand, there is a disadvantage that rewriting a *part* of data is inefficient. Generally the flash memory requires to erase existing data before write a new data, and minimum unit of erase operation (called erase block) is larger than write block size. The typical NAND flash memory has a block size of 512/16K bytes for write/erase operation, and recent monster card employs large block chip (2K/128K). This means that rewriting entire data in the erase block is done in the card even if write only a sector (512 bytes).

Benchmark

I examined the read/write performance of two SD cards using the available drivers for the STM32F103. In each case the experiment involved either writing a 512-byte block The sector number, i, was varied from 0 to 100

```
GPIOB->ODR |= 0x1000;    // bit 12 on LED
result = disk_write(0,testBuff,i,1);
GPIOB->ODR &= ~0x1000;   // bit 12 off LED
```

or by reading a 512-byte block

```
GPIOB->ODR |= 0x2000;    // bit 13 on LED
result = disk_read(0,buffer,i,1);
GPIOB->ODR &= ~0x2000;   // bit 13 off LED
```

The results were:

Kingston 2GB SD Memory Card: SD-M02G

Write block time: 2300 µs/block, or about 200 kbytes/sec

Read block time: 532 µs/block, or about 1 Mbytes/sec

Kingston 4GB SD Memory Card: SD-K04G

Write block time: 4000 µs/block, or about 128 kbytes/sec

Read block time: 665 µs/block, or about 0.77 Mbytes/sec

The performance seems to vary considerably. For example, when calling read directly after a write the speed was dramatically faster.

```
result = disk_initialize(0);
```

```
  if(result) printf("SD error=%u\n\r",result);
  while(result==0){
    for(i=0;i<100;i++){
      GPIOB->ODR |= 0x1000;    // bit 12 on LED
      result = disk_write(0,testBuff,i,1);
      GPIOB->ODR &= ~0x1000;   // bit 12 off LED
        for(j=0;j<1000;j++);
      if(result) printf("SD write error=%u block=%u\n\r",result,i);
      GPIOB->ODR |= 0x8000;    // bit 15 on LED
      result = disk_read(0,buffer,i,1);
      GPIOB->ODR &= ~0x8000;   // bit 15 off LED
      if(result) printf("SD read error=%u block=%u\n\r",result,i);
      }
  }
```

Kingston 2GB SD Memory Card: SD-M02G

Write block time: 2300 μs/block, or about 200 kbytes/sec

Read block time: 272 μs/block, or about 1 Mbytes/sec

Links

MMCA - Multimedia Card Association http://www.mmca.org/

SDA - SD Card Association http://www.sdcard.org/

SDHC Physical Layer Spec. http://www.sdcard.org/developers/tech/sdcard/pls/

About SPI http://elm-chan.org/docs/spi_e.html

Generic FAT file system module http://elm-chan.org/fsw/ff/00index_e.html

with sample code to control *MMC/SDSC/SDHC*

Low-level driver (drv = 0) block size is 512 bytes

Files needed

edisk.h	header file for SD interface
edisk.c	implementation file for SD interface
integer.h	

LBA stands for logical block address, meaning they are numbers 0, 1, 2, 3,…

```
// DSTATUS of type BYTE (8 bits)
//  STA_NOINIT   0x01   Drive not initialized
//  STA_NODISK   0x02   No medium in the drive
//  STA_PROTECT  0x04   Write protected

DSTATUS eDisk_Initialize(BYTE drv);

DSTATUS eDisk_Status(BYTE drv);

// DRESULT of type BYTE (8 bits)
//  RES_OK         0: Successful
//  RES_ERROR      1: R/W Error
//  RES_WRPRT      2: Write Protected
```

```
//  RES_NOTRDY    3: Not Ready
//  RES_PARERR    4: Invalid Parameter

DRESULT eDisk_Read (
  BYTE drv,     // Physical drive number (0)
  BYTE *buff,   // Pointer to buffer to read data
  DWORD sector, // Start sector number (LBA)
  BYTE count);  // Sector count (1..255)

//*************** eDisk_ReadBlock ***********
// Read 1 block of 512 bytes from the SD (write to RAM)
// Inputs: pointer to an empty RAM buffer
//         sector number of SD card to read: 0,1,2,...
// Outputs: result
//  RES_OK        0: Successful
//  RES_ERROR     1: R/W Error
//  RES_WRPRT     2: Write Protected
//  RES_NOTRDY    3: Not Ready
//  RES_PARERR    4: Invalid Parameter
DRESULT eDisk_ReadBlock (
    BYTE *buff,   /* Pointer to buffer to store data */
    DWORD sector  /* Start sector number (LBA) */
)

DRESULT eEisk_Write (
  BYTE drv,         // Physical drive number (0)
  const BYTE *buff, // Pointer to the data to be written
  DWORD sector,     // Start sector number (LBA)
  BYTE count);      // Sector count (1..255)
)
//*************** eDisk_WriteBlock ***********
// Write 1 block of 512 bytes of data to the SD card
// Inputs: pointer to RAM buffer with information
//         sector number of SD card to write: 0,1,2,...
// Outputs: result
//  RES_OK        0: Successful
//  RES_ERROR     1: R/W Error
//  RES_WRPRT     2: Write Protected
//  RES_NOTRDY    3: Not Ready
//  RES_PARERR    4: Invalid Parameter
DRESULT eDisk_WriteBlock (
    const BYTE *buff,   /* Pointer to data to be written */
    DWORD sector        /* Start sector number (LBA) */
)
void disk_timerproc(void); // to be called every 10ms
```

High-level FAT16 file system, see StellarisWare sd_card example