

**Lecture 12: SPI and SD cards**  
EE-379 Embedded Systems and Applications  
*Electrical Engineering Department, University at Buffalo*  
*Last update: Cristinel Ababei, March 2013*

## 1. Objective

The objective of this lecture is to learn about Serial Peripheral Interface (SPI) and micro SD memory cards.

## 2. SPI Introduction

Serial Peripheral Interface (SPI) communication was used to connect devices such as printers, cameras, scanners, etc. to a desktop computer; but it has largely been replaced by USB. SPI is still utilized as a communication means for some applications using displays, memory cards, sensors, etc. SPI runs using a master/slave set-up and can run in full duplex mode (i.e., signals can be transmitted between the master and the slave simultaneously). When multiple slaves are present, SPI requires no addressing to differentiate between these slaves. There is no standard communication protocol for SPI.

SPI is used to control peripheral devices and has some advantages over I2C. Because of its simplicity and generality, it is being incorporated in various peripheral ICs. The number of signals of SPI, three or four wires, is larger than I2C'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 I2C). Therefore, often it is used in applications (ADC, DAC or communication between ICs) that require high data transfer rates.

The SPI communication method is illustrated in Fig.1 below. 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). The contents of both 8-bit shift registers are exchanged with the shift clock driven by master IC. An additional fourth signal, **SS** (Slave Select), is utilized to synchronize the start of packet or byte boundary and to facilitate working with multiple slave devices simultaneously. Most slave ICs utilize different pin names (e.g., DI, DO, SCK and CS) to the SPI interface. For one-way transfer devices, such as DAC and single channel ADC, either of data lines may be omitted. The data bits are shifted in MSB first.

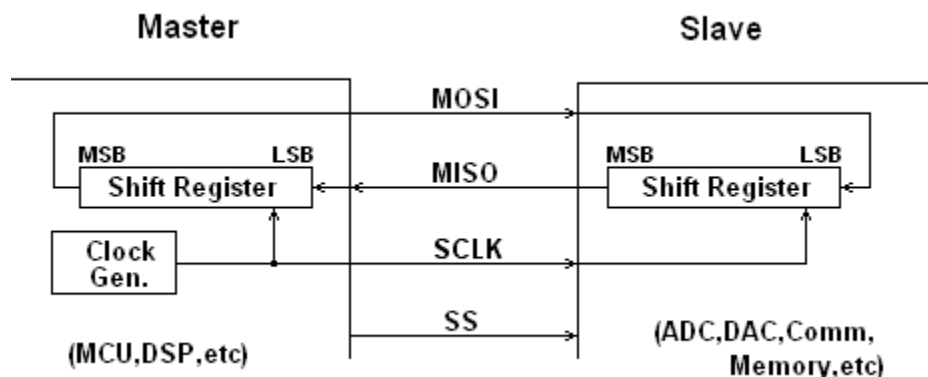
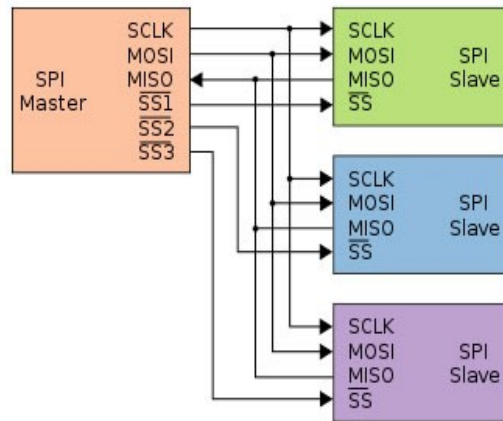


Figure 1 SPI communication method.

When additional slaves are attached to the SPI bus, they are attached in parallel and an individual SS signal must be connected from the master to each of the slaves (as shown in Fig.2). The data output of the slave IC

is enabled when the corresponding SS signal is set; the data output is disconnected from the MISO line when the slave device is deselected.



**Figure 2 Single master multiple slaves configuration.**

In SPI, data shift and data latch are done on opposite clock edges. Consequently, the four different operation modes (as a result of the combination of clock polarity and clock phase) that the master IC can configure its SPI interface are shown in Fig.3 below.

SPI Mode	Timing Diagram
<b>Mode 0</b> Positive Pulse. Latch, then Shift. (CPHA=0, CPOL=0)	
<b>Mode 1</b> Positive Pulse. Shift, then Latch.	
<b>Mode 2</b> Negative Pulse. Latch, then Shift.	
<b>Mode 3</b> Negative Pulse. Shift, then Latch.	

**Figure 3 The four different operation modes of SPI.**

There is a lot of online information on SPI. You should search and read some for more details. Good starting points are the references suggested in [1] from where most of the above material has been adopted. Also, read Chapter 17 of the LPC17xx User Manual for details on the SPI interface available on the LPC1768 MCU that we use in this course.

### **3. MMC and SDC Cards**

#### ***A) Background***

The Secure Digital Memory Card (SDC) is the de facto standard memory card for mobile devices. The SDC was developed as upper-compatible to Multi Media Card (MMC). SDC compliant equipment can also use MMCs in most cases. These cards have basically a flash memory array and a (micro)controller inside. The flash memory controls (erasing, reading, writing, error controls, and wearleveling) are completed inside the memory card. The data is transferred between the memory card and the host controller as data blocks in units of 512 bytes; therefore, these cards can be seen as generic hard disk drives from the view point of upper level layers. The currently defined file system for the memory card is FAT12/16 with FDISK partitioning rule. The FAT32 is defined for only high capacity ( $\geq 4\text{G}$ ) cards.

Please take a while and read the following very popular webpage that describes the use of MMC and SDC cards: [http://elm-chan.org/docs/mmc/mmc\\_e.html](http://elm-chan.org/docs/mmc/mmc_e.html)

A lot of the concepts described in the aforementioned webpage apply also to working with micro SD cards, which we'll use in the examples studied later on in this lecture.

A block diagram of the SD card is shown in Fig.4. It consists of a 9-pin interface, a card controller, a memory interface and a memory core. The 9-pin interface allows the exchange of data between a connected system and the card controller. The controller can read/write data from/to the memory core using the memory core interface. In addition, several internal registers store the state of the card. The controller responds to two types of user requests: control and data. Control requests set up the operation of the controller and allow access to the SD card registers. Data requests are used to either read data from or write data to the memory core.

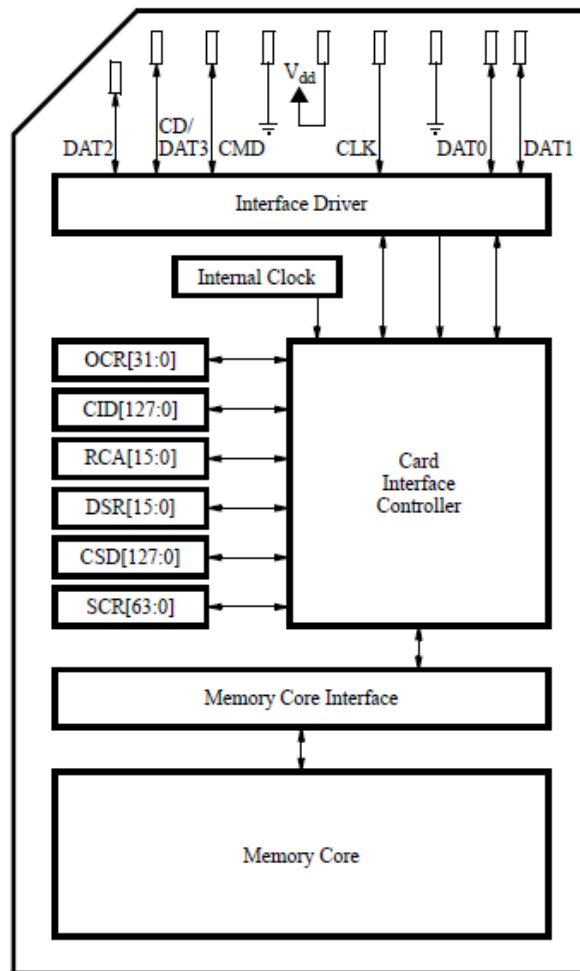


Figure 4 Block diagram of an SD card.

#### B) Communication with SD cards

Communication with an SD card can be done in one of two modes: the SD mode or the SPI mode. By default, the SD card operates in the SD mode. However, we'll work with the SPI mode and communicate with it using the SPI protocol.

Communication with the SD card is performed by sending commands to it and receiving responses from it. A valid SD card command consists of 48 bits as shown in Fig.5. The leftmost two bits are the start bits which we set to (01). They are followed by a 6-bit **command number** and a 32-bit argument where additional information may be provided. Next, there are 7 bits containing a Cyclic Redundancy Check (CRC) code, followed by a single stop bit (1).

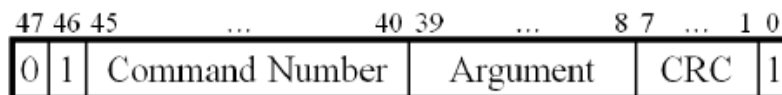


Figure 5 Format of the 48-bit command for an SD card.

The CRC code is used by the SD card to verify the integrity of a command it receives. By default, the SD card ignores the CRC bits for most commands (except CMD8) unless a user requests that CRC bits be checked upon receiving every message.

Sending a command to the SD card is performed in serial fashion. By default, the MOSI line is set to 1 to indicate that no message is being sent. The process of sending a message begins with placing its most-significant bit on the MOSI line and then toggling the SD CLK signal from 0 to 1 and then back from 1 to 0. Then, the second bit is placed on the MOSI line and again the SD CLK signal is toggled twice. Repeating this procedure for each bit allows the SD card to receive a complete command.

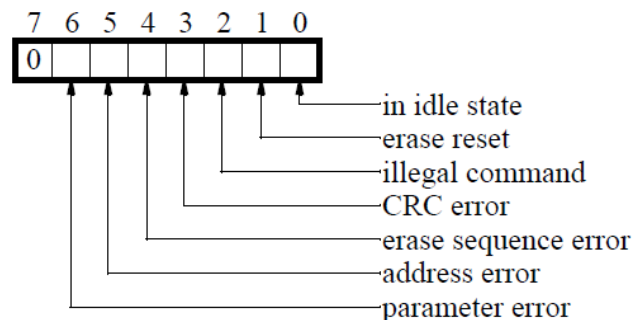
Once the SD card receives a command it will begin processing it. To respond to a command, the SD card requires the SD CLK signal to toggle for at least 8 cycles. Your program will have to toggle the SD CLK signal and maintain the MOSI line high while waiting for a response. The length of a response message varies depending on the command. Most of the commands get a response mostly in the form of 8-bit messages, with two exceptions where the response consists of 40 bits.

To ensure the proper operation of the SD card, the SD CLK signal should have a frequency in the range of 100 to 400 kHz.

To communicate with the SD card, your program has to place the SD card into the SPI mode. To do this, set the MOSI and CS lines to logic value 1 and toggle SD CLK for at least 74 cycles. After the 74 cycles (or more) have occurred, your program should set the CS line to 0 and send the command CMD0:

**01 000000 00000000 00000000 00000000 00000000 1001010 1**

This is the reset command, which puts the SD card into the SPI mode if executed when the CS line is low. The SD card will respond to the reset command by sending a basic 8-bit response on the MISO line. The structure of this response is shown in Fig.6. The first bit is always a 0, while the other bits specify any errors that may have occurred when processing the last message. If the command you sent was successfully received, then you will receive the message (00000001)<sub>2</sub>.



**Figure 6 Format of a basic 8-bit Response to every command in SPI mode.**

To receive this message, your program should continuously toggle the SD CLK signal and observe the MISO line for data, while keeping the MOSI line high and the CS line low. Your program can detect the message, because every message begins with a 0 bit, and when the SD card sends no data it keeps the MISO line high. Note that the response to each command is sent by the card a few SD CLK cycles later. If the expected response is not received within 16 clock cycles after sending the reset command, the reset command has to be sent again.

Following a successful reset, test if your system can successfully communicate with the SD card by sending a different command. For example, send one of the following commands, while keeping the CS at value 0:

(i) Command CMD8

**01 001000 00000000 00000000 00000001 10101010 0000111 1**

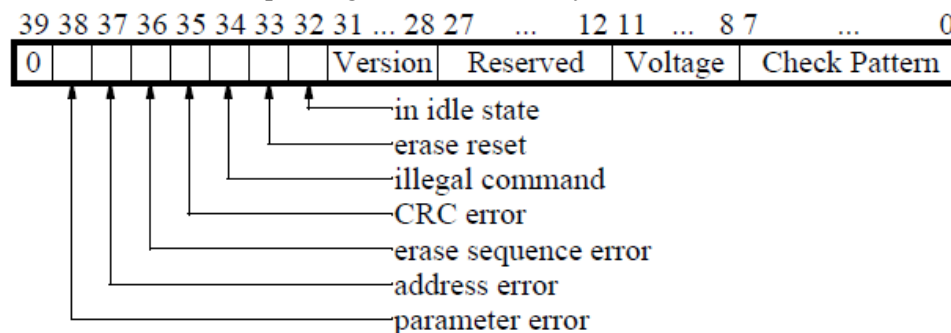
This command is only available in the latest cards, compatible with SD card Specifications version 2.0. For most older cards this command should fail and cause the SD card to respond with a message that this command is illegal.

(ii) Command CMD58

**01 111010 00000000 00000000 00000000 00000000 0111010 1**

This command requests the contents of the operating conditions register for the connected card.

A response to these commands consists of 40 bits (see Fig.7), where the first 8 bits are identical to the basic 8-bit response, while the remaining 32 bits contain specific information about the SD card. Although the actual contents of the remaining 32 bits are not important for this discussion, a valid response indicates that your command was transmitted and processed successfully. If successful, the first 8 bits of the response will be either 00000001 or 00000101 depending on the version of your SD card.



**Figure 7 The format of the 40-bit Response.**

The steps necessary to complete the SD card initialization are shown in the flowchart in Fig.8. The flowchart consists of boxes in which a command number is specified. Starting at the top of the flowchart, each command has to be sent to the SD card, and a response has to be received. In some cases, a response from the card needs to be processed to decide the next course of action, as indicated by the diamond-shaped decision boxes.

This is just one example of such an initialization sequence (credit:

[ftp://ftp.altera.com/up/pub/Altera\\_Material/12.0/Laboratory\\_Exercises/Embedded\\_Systems/DE2/embed\\_la\\_b9.pdf](ftp://ftp.altera.com/up/pub/Altera_Material/12.0/Laboratory_Exercises/Embedded_Systems/DE2/embed_la_b9.pdf), from where portions of this discussion are adopted). It's possible that it may work or not depending

on the type of card you use and on its manufacturer. Other similar such flowcharts exist; for example:

<http://elm-chan.org/docs/mmc/sdinit.png>. Most of the times, you will have to try to implement several of them until you will get it to work; unless you use a known card and an already written program for that card.

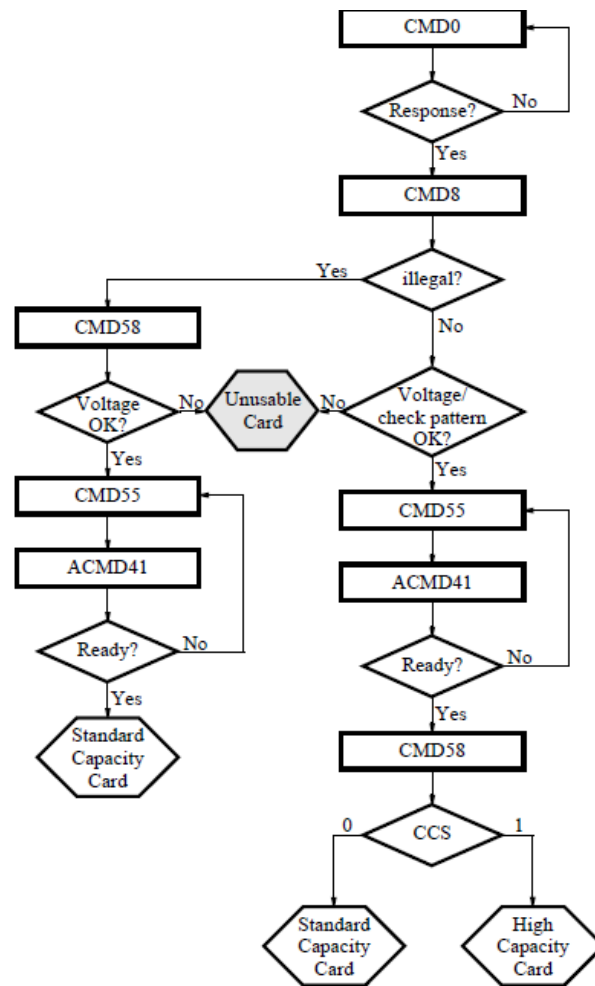


Figure 8 Flowchart of a generic SD card initialization routine in SPI mode.

### C) FAT File system basics

Microcontrollers are relatively memory constrained, e.g., 512K bytes flash, which limits the ability to store large quantities of data either as inputs to or outputs from an embedded program. For example, in a game application it might be desirable to access sound and graphic files or in a data logging application, to store extended amounts of data. In addition, accessing the contents of the flash requires a special interface and software. In such applications it is desirable to provide external storage which the MCU can access while running and the user/programmer can easily access at other times. SD cards provide a cost effective solution which can be accessed by both the processor and user. In practice, these cards have **file systems** (typically FAT) and can be inserted in commonly available adaptors. Furthermore, the physical interface of most SD cards has a SPI mode (described in the previous section and which is demonstrated with an actual implementation in Example 1 in the next section) which is accessible.

Communicating with an SD memory card is relatively simple using some SPI driver like the one described in the previous section. However, controlling the card and interpreting the data communicated requires significant additional software. Such software practically elevates the level of access to the card to a higher level of abstraction as illustrated in Fig.9 (user or application space).

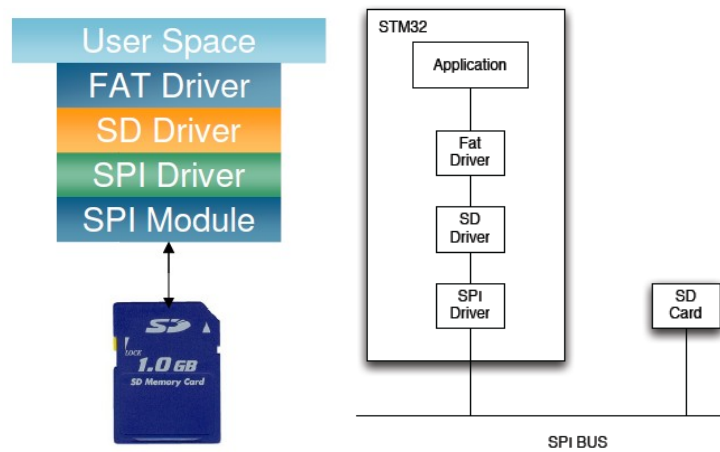


Figure 9 SD card software stack.

The aforementioned software is typically what is referred as a **FAT file system driver**. One such example is the available widely used FatFs module, which with some amount of porting could be utilized with our SPI driver. Note that Keil offers a similar module too, but it's not free; if you tried to compile for example the **SD\_File/** project, you will get an error as you would need an RL license...

The data on an SD card is organized as a file system – cards below 2GB are typically formatted as FAT16 file systems. In a FAT file system, the first several storage blocks are used to maintain data about the file system – for example allocation tables – while the remaining blocks are used to store the contents of files and directories.

An application, which wishes to access data stored on an SD Card, utilizes file level commands such as open, read, and write to access specific files within the SD card file system. These commands are provided by a FAT file system driver. The FAT file system issues commands at the level of block reads and writes without any knowledge of how these commands are implemented. A separate SD driver implements these commands. Finally, the SD driver utilizes the SPI interface to communicate with the SD Card.

Fortunately, it is not necessary to write all of this software. One can use of the FatFs generic file system (or use licensed ones but for pay, like the one from Keil). This open source package provides most of the components required including a “generic” SD driver that is relatively easily modified to utilize with any typical SPI driver.

To understand how an application interacts with FatFs, consider the example derived from the FatFs sample distribution illustrated in Fig.10 (credit: <http://homes.soic.indiana.edu/geobrown/c335> from where portions of this description have been adopted too). This example fragment assumes it is communicating with an SD card formatted with a fat file system which contains file in the root directory called MESSAGE.TXT. The program reads this file, and creates another called HELLO.TXT. Notice the use of relatively standard file system commands.



```

f_mount(0, &Fatfs);/* Register volume work area */

xprintf("\nOpen an existing file (message.txt).\n");
rc = f_open(&Fil, "MESSAGE.TXT", FA_READ);

if (!rc) {
    xprintf("\nType the file content.\n");
    for (;;) {
        /* Read a chunk of file */
        rc = f_read(&Fil, Buff, sizeof Buff, &br);
        if (rc || !br) break;/* Error or end of file */
        for (i = 0; i < br; i++)/* Type the data */
            myputchar(Buff[i]);
    }
    if (rc) die(rc);
    xprintf("\nClose the file.\n");
    rc = f_close(&Fil);
    if (rc) die(rc);
}

xprintf("\nCreate a new file (hello.txt).\n");
rc = f_open(&Fil, "HELLO.TXT", FA_WRITE | FA_CREATE_ALWAYS);
if (rc) die(rc);

xprintf("\nWrite a text data. (Hello world!)\n");
rc = f_write(&Fil, "Hello world!\r\n", 14, &bw);
if (rc) die(rc);
xprintf("%u bytes written.\n", bw);

```

Figure 10 FatFs example.

#### 4. Example 1: Reading from a micro SD card (1GB) raw data from fixed physical addresses

In this example, I simply read raw data from a fixed physical address on the micro SD memory card. The SD card has only one file, which is a small .png image **alfaromeo.png** with size 320x240 pixels. Using the free HxD hexadecimal editor, I first found that the physical address where the file is stored inside the memory is **0x00040000**. This is a neat editor, which also shows the contents of the file as in figure below:

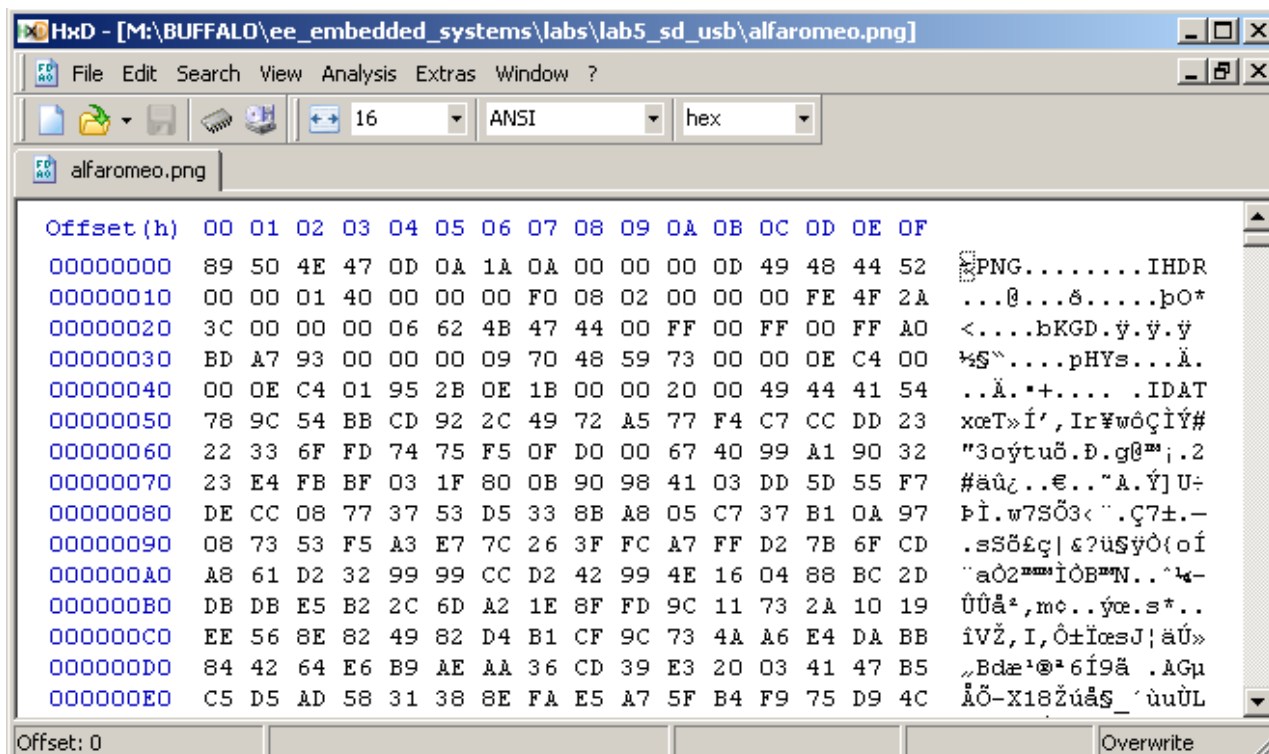
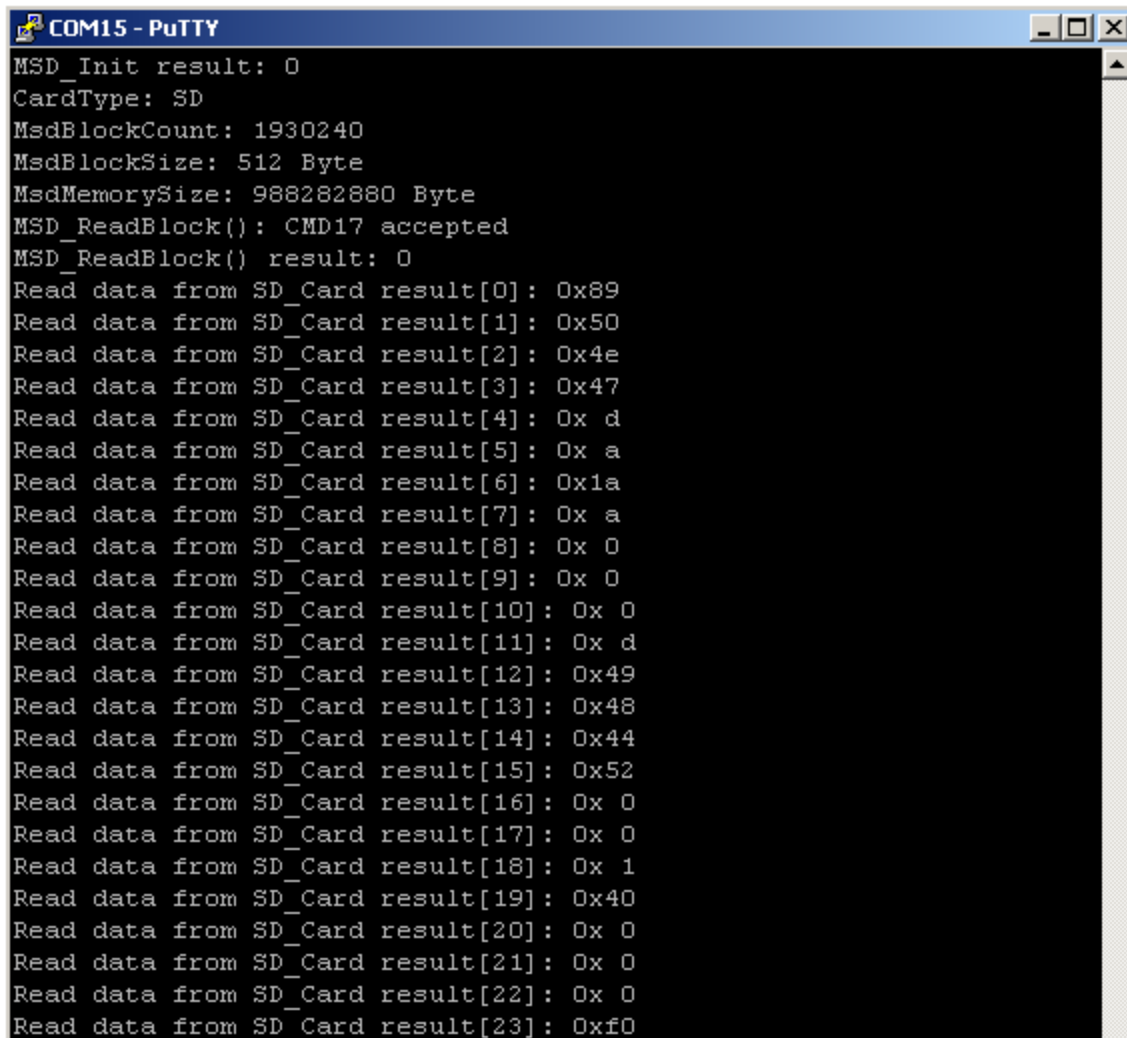


Figure 11 Contents of .png file displayed using HxD editor.

The program in this example (found as the entire uVision project **lab5\_sd\_dbg/** inside the downloadable archive of lab#5) simply: 1) initializes the SPI and the SD card and 2) reads data from the SD card starting at address **0x00040000** and then prints byte by byte to the putty terminal of the host PC. This relatively simple set-up is very useful for debugging purposes and for investigating micro SD cards (which working with is kind of painful as a lot of times SD cards do not behave as theoretically described on [sdcard.org](http://sdcard.org) or for example in SanDisk's ambiguous documentation...).

Use the provided uVision project **lab5\_sd\_dbg/**. Clean and build, then download to the MCB1700 board. Observe operation and comment. Study the source code. The putty terminal should print useful information as in the figure below. Note that the data read from the SD card matches the data shown using the HxD editor.



```
COM15 - PuTTY
MSD_Init result: 0
CardType: SD
MsdblockCount: 1930240
MsdblockSize: 512 Byte
MsdbMemorySize: 988282880 Byte
MSD_ReadBlock(): CMD17 accepted
MSD_ReadBlock() result: 0
Read data from SD_Card result[0]: 0x89
Read data from SD_Card result[1]: 0x50
Read data from SD_Card result[2]: 0x4e
Read data from SD_Card result[3]: 0x47
Read data from SD_Card result[4]: 0x d
Read data from SD_Card result[5]: 0x a
Read data from SD_Card result[6]: 0x1a
Read data from SD_Card result[7]: 0x a
Read data from SD_Card result[8]: 0x 0
Read data from SD_Card result[9]: 0x 0
Read data from SD_Card result[10]: 0x 0
Read data from SD_Card result[11]: 0x d
Read data from SD_Card result[12]: 0x49
Read data from SD_Card result[13]: 0x48
Read data from SD_Card result[14]: 0x44
Read data from SD_Card result[15]: 0x52
Read data from SD_Card result[16]: 0x 0
Read data from SD_Card result[17]: 0x 0
Read data from SD_Card result[18]: 0x 1
Read data from SD_Card result[19]: 0x40
Read data from SD_Card result[20]: 0x 0
Read data from SD_Card result[21]: 0x 0
Read data from SD_Card result[22]: 0x 0
Read data from SD_Card result[23]: 0xf0
```

Figure 12 Output of Putty terminal connected to the MCB170 board.

## 5. Example 2: Read picture from micro SD card and display on the 320x240 LCD display of the MCB1700 board

This example is an improved version of the previous example. Here, I simply read the small .bmp (TODO: make it work with .png format too) image, **alfaromeo.bmp**, and display it on the 320x240 LCD screen of the MCB1700 board. The program in this example can be found inside the uVision project **lab5\_sd\_lcd/** inside the downloadable archive of lab#5.

Clean and build, then download to the MCB1700 board. Observe operation and comment. Study the source code. You should see the image displayed on the LCD screen as shown below (use the micro SD card from the TA).



Figure 13 Cris' Alfa Romeo sports-car shown on the LCD display of the MCB1700 board. ☺

## 6. Example 3: Manipulate files on the SD card using the FatFs file system

The program in this example can be found inside the uVision project **lab5\_sd\_fat/** inside the downloadable archive of lab#5. Clean and build, then download to the MCB1700 board. Observe operation and comment. Study the source code.

## 7. References, credits

[1] SPI related references:

--About SPI; [http://elm-chan.org/docs/spi\\_e.html](http://elm-chan.org/docs/spi_e.html)

--SPI bus (see also the references and external links therein!);

[http://en.wikipedia.org/wiki/Serial\\_Peripheral\\_Interface\\_Bus](http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus)

[2] SD memory cards related references:

--How to use MMC/SDC cards (very popular reference!); [http://elm-chan.org/docs/mmc/mmc\\_e.html](http://elm-chan.org/docs/mmc/mmc_e.html)

--Modified version of the above: [http://users.ece.utexas.edu/~valvano/EE345M/view12\\_SPI\\_SDC.pdf](http://users.ece.utexas.edu/~valvano/EE345M/view12_SPI_SDC.pdf)

--Lab manual; <http://homes.soic.indiana.edu/geobrown/c335>

--Application note; [http://alumni.cs.ucr.edu/~amitra/sdcard/Additional/sdcard\\_appnote\\_foust.pdf](http://alumni.cs.ucr.edu/~amitra/sdcard/Additional/sdcard_appnote_foust.pdf)

--<https://www.sdcard.org/home/>

--SDCARD Physical Layer Simplified Specification;

[https://www.sdcard.org/downloads/pls/simplified\\_specs/archive/part1\\_101.pdf](https://www.sdcard.org/downloads/pls/simplified_specs/archive/part1_101.pdf)

--Toshiba SD Card specification;

[http://embdev.net/attachment/39390/TOSHIBA\\_SD\\_Card\\_Specification.pdf](http://embdev.net/attachment/39390/TOSHIBA_SD_Card_Specification.pdf)

--MultiMediaCard (MMC); <http://en.wikipedia.org/wiki/MultiMediaCard>

--Secure Digital (SD) Card; [http://en.wikipedia.org/wiki/Secure\\_Digital\\_card](http://en.wikipedia.org/wiki/Secure_Digital_card)

--FatFs - Generic FAT File System Module; [http://elm-chan.org/fsw/ff/00index\\_e.html](http://elm-chan.org/fsw/ff/00index_e.html)

[3] PNG and BMP formats:

-- [http://en.wikipedia.org/wiki/Portable\\_Network\\_Graphics](http://en.wikipedia.org/wiki/Portable_Network_Graphics)

-- [http://en.wikipedia.org/wiki/BMP\\_file\\_format](http://en.wikipedia.org/wiki/BMP_file_format)