# AMS Lab exercise 9
# SD card driver



**Purpose**
To understand how to interface a typical SD Flash card from a microcontroller.
Part 1: A driver is implemented using basic SD access.
Part 2: A FAT32 API for the SD card is implemented based on the "FatFs" generic framwork.

**Literature**
- "ITDB02 Arduino Mega shield" data sheet.
- "ITDB02 Arduino Mega shield" schematic.
- SD specification for the physical layer.
- SANDISK SD card data sheet.
- SD card info.
- FatFs: http://elm-chan.org/fsw/ff/00index_e.html.

**Material**
- Arduino Mega2560 board.
- ITDB02 Arduino Mega shield.
- ITDB02 TFT display module (version 2). We will use the SD card interface only.
- An SD memory card.

The relevant documents are available at AMS Brightspace.

**Exercise setup**
In this exercise, we will write and test a C driver for an SD memory card inserted in the SD cardholder of the ITDB02 TFT display module.

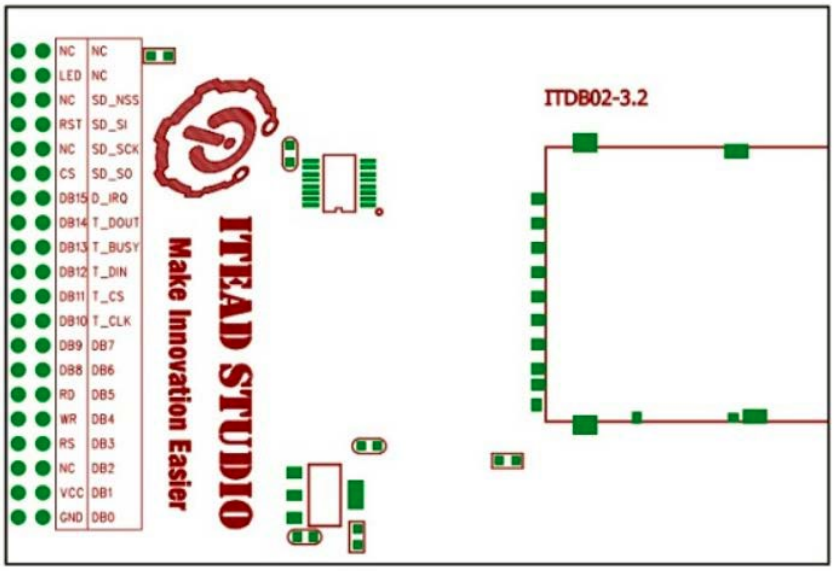
Start by mounting the Arduino Mega shield and the Display module on the Mega2560 Arduino.

*Notice: Do not mount on top of the "Arduino Mega2560 I/O Shield" (PR5824).*
*Some signals will then interfere.*

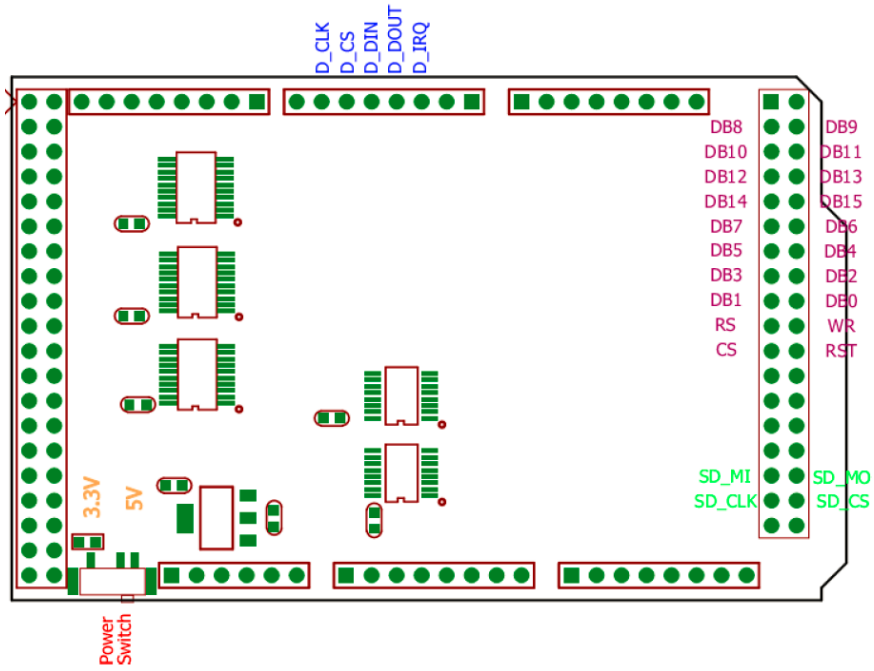
To avoid damaging the hardware (by shortcuttings), remember to place some nonconduction material (eg. plastic foam) between the display mudule and the Arduino shield!

The position of the power switch (3,3 volt / 5 volt) does not matter.

The SD socket is positioned at the back of the display board:




The SPI connections for the SD card as seen from the Mega2560 board (in green):




This table shows the connections between the Meaga2560 and the SD card:

| **Mega2560 pin** | **SPI use** | **SD card signal** |
|---|---|---|
| PB3 | MISO | SD_OUT |
| PB2 | MOSI | SD_IN |
| PB1 | SCLK | SD_CLK |
| PB0 | SS | SD_CS |

The CD card runs at 3,3 volts, and its signal levels are 3,3 volts.
Conversion between 5 volt (Mega2560 supply voltage) and 3,3 volt are done at the ITDB02 shield.

**Part 1: "Basic driver SD"**
Implement and test a C driver for basic, direct access to the SD card (using no file system).
The driver must at least have functions for initializing, block reading, block writing and blocks erasing.

As appendix to this part of the exercise, a partly implemented driver (plus a suitable test program) is available at AMS Brightspace.

Write and test the unimplemented functions (in the file "SD_Driver.c").
However first study – and understand - the already implemented functions.

The driver's public functions are:

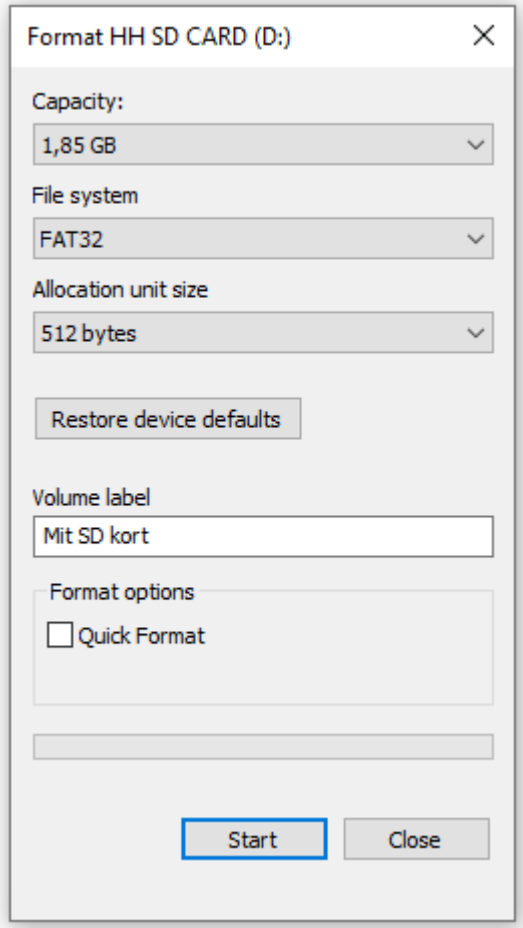
```
unsigned char SD_init(void);
unsigned char SD_sendCommand(unsigned char cmd, unsigned long arg);
unsigned char SD_readSingleBlock(unsigned long startBlock, unsigned char* ptr);
unsigned char SD_writeSingleBlock(unsigned long startBlock, unsigned char* ptr);
unsigned char SD_erase (unsigned long startBlock, unsigned long totalBlocks);
```

PS: As another appendix you will find the file "LAB9_1.hex".
This is the compiled version of the full solution.
It might be useful for testing your hardware setup.

**Part 2: "FAT32"**
There are numerous implementations of FAT file systems for microcontrollers.
One popular example is "FatFs" since it has a good structure and most modules written in generic C code. The developer can glue FatFs to virtually any low level "Media Access Interface" – in our case, we will use our SD driver from part 1 of this exercise.

It is assumed that you have some basic knowledge of the FAT32 file system.
Find some tutors at the internet and watch this video:
https://www.youtube.com/watch?v=V2Gxqv3bJCk

For the exercise, you need to have your SD card FAT32 formatted.
Insert the card in your PC, right-click and select "format".
The allocation unit size must be 512 bytes and do a full format (not quick format):

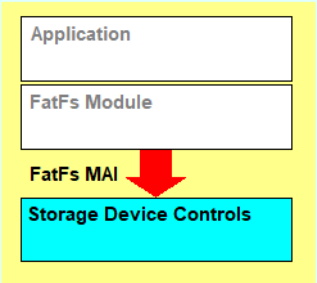



Now, study FatFs using this link: http://elm-chan.org/fsw/ff/00index_e.html.

Notice: The functions in the "Media Access Interface" part are practically the only functions the developer has to implement:

## Media Access Interface

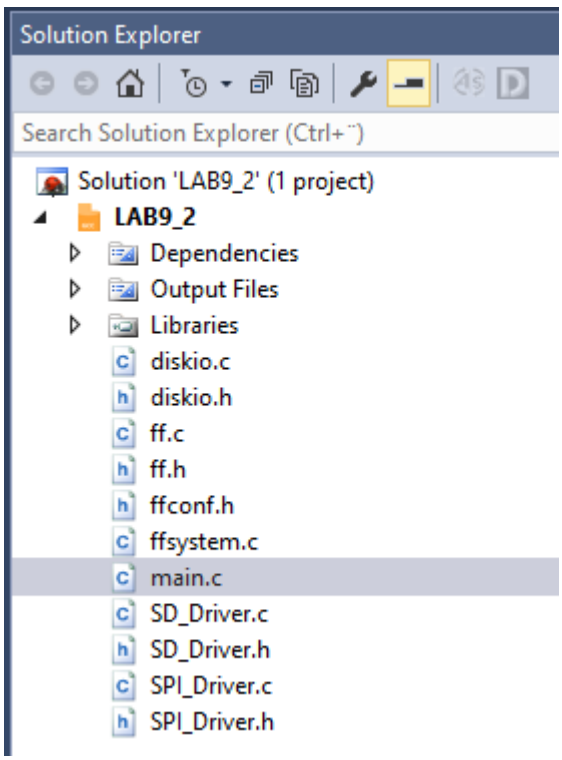
Since FatFs module is the **filesystem layer** independent of platforms and storage media, it is completely separated from the physical devices, such as memory card, harddisk and any type of storage device. The storage device control module is **not any part of FatFs module** and it needs to be provided by implementer. FatFs controls the storage devices via a simple media access interface shown below. Also sample implementations for some platforms are available in the downloads. A function checker for storage device control module is available here.




- Storage Device Controls
  - disk_status - Get device status
  - disk_initialize - Initialize device
  - disk_read - Read data
  - disk_write - Write data
  - disk_ioctl - Control device dependent functions
- Real Time Clock
  - get_fattime - Get current time

PS: At AMS Brightspace, the "FatFs" files are available in the zipped file "ff14a.zip".

Now create a new Atmel Studio project (call it LAB9_2) and merge the files from "ff14a.zip" and your SD driver + SPI driver from part 1 of this exercise.




The file "ffunicode.c" can be removed, since it is only relevant if we use long filenames (we don't).

Use this or similar as your main() code ("main.c" can be downloaded from AMS Brightspace):

```
/*-----------------------------------------------------------------------*/
/* Foolproof FatFs sample project for AVR              (C)ChaN, 2014     */
/*-----------------------------------------------------------------------*/
#include <avr/io.h> /* Device specific declarations */
#include "ff.h"     /* Declarations of FatFs API */

FATFS FatFs;        /* FatFs work area needed for each volume */
FIL Fil;            /* File object needed for each open file */

int main ()
{
    UINT bw;
    FRESULT fr;

    f_mount(&FatFs, "", 0);     /* Give a work area to the default drive */

    fr = f_open(&Fil, "newfile.txt", FA_WRITE | FA_CREATE_ALWAYS);  /* Create a file */
    if (fr == FR_OK) {
        f_write(&Fil, "Det virker!\r\n", 13, &bw);  /* Write data to the file */
        fr = f_close(&Fil);                          /* Close the file */
        if (fr == FR_OK && bw == 11) {
            /* Indicate all right */
            /* in some way */
        }
    }
    while (1)
    {}
}
```

Before the project will compile and execute correctly, do the following modifications:

1. Change configuration settings in the file "ffconf.c":
```
#define FF_CODE_PAGE 932      //Optional change
#define FF_FS_NORTC 1
#define FF_NORTC_MON 3        //Optional change
#define FF_NORTC_MDAY 15      //Optional change
#define FF_NORTC_YEAR 2021    //Optional change
```

2. The file "diskio.c" implements the "Media Access Interface".
   Some confusion can raise from these defines:
```
/* Definitions of physical drive number for each drive */
#define DEV_RAM     0   /* Example: Map Ramdisk to physical drive 0 */
#define DEV_MMC     1   /* Example: Map MMC/SD card to physical drive 1 */
#define DEV_USB     2   /* Example: Map USB MSD to physical drive 2 */
```
   We only want to interface our SD card (= MMC), so it might be usefull to remove the "case" in the functions of "diskio.c" (and not use the defines listed above).

3. The function disk_status() shall simply return the variable "stat".

4. The function disk_initialize() shall initialize the SD card.
   Return RES_OK if initializing was ok, otherwise return RES_ERROR.The function disk_read() shall read "count" number of 512 byte blocks from the SD card to the byte array pointed to by "buff".
   Return RES_OK if read was ok, otherwise return RES_ERROR.

5. The function disk_write() shall write "count" number of 512 byte blocks to the SD card from the byte array pointed to by "buff".
   Return RES_OK if read was ok, otherwise return RES_ERROR.

6. The function disk_diskiocl() is not used in our implementation.
   Let it simply return the variable "res".

Test the program and make some relevant modifications to the main() function.