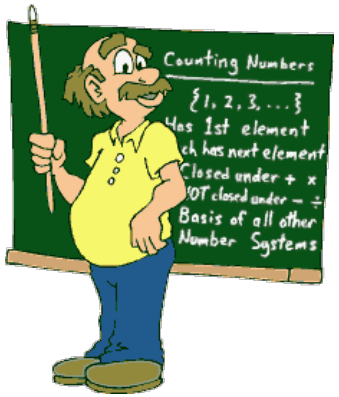
AMS

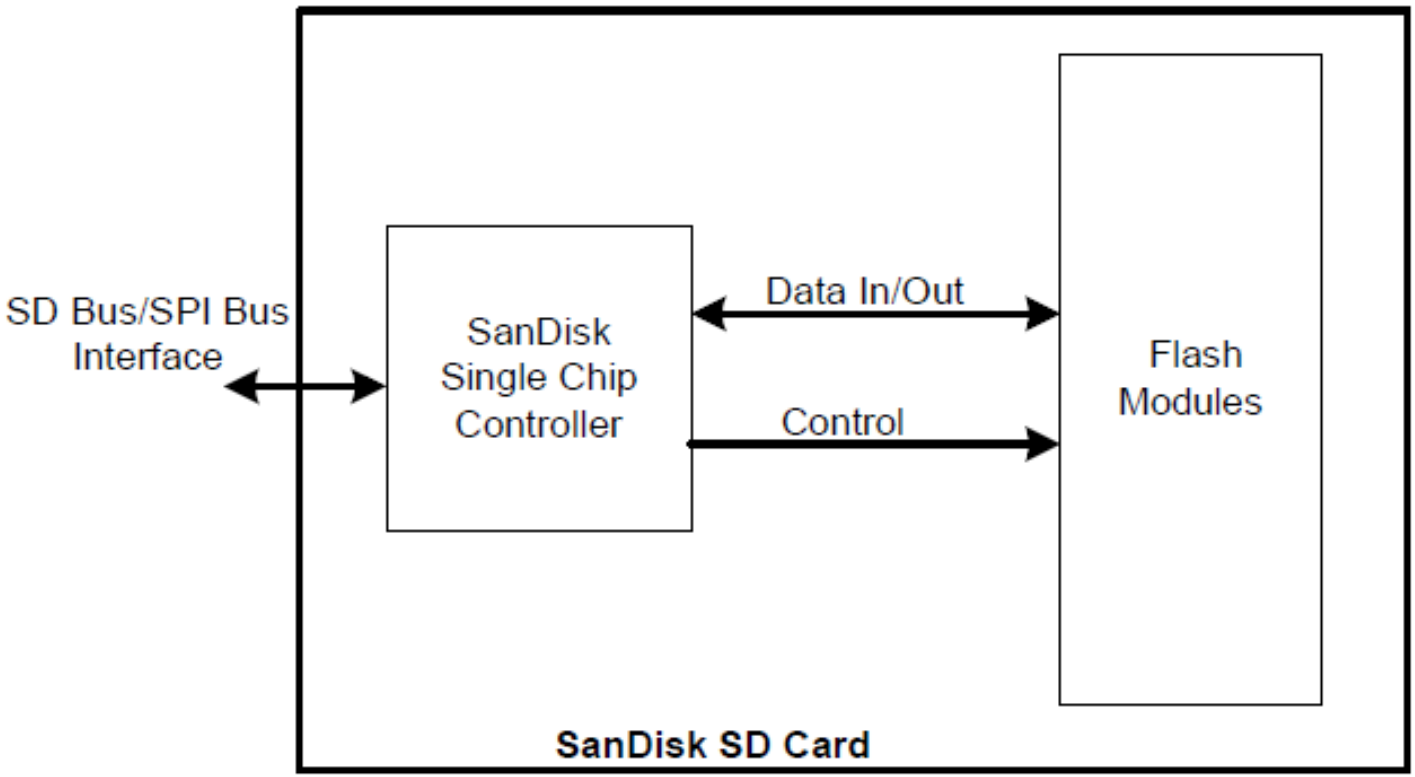
Applied Microcontroller Systems

Lesson 7: Interfacing SD cards

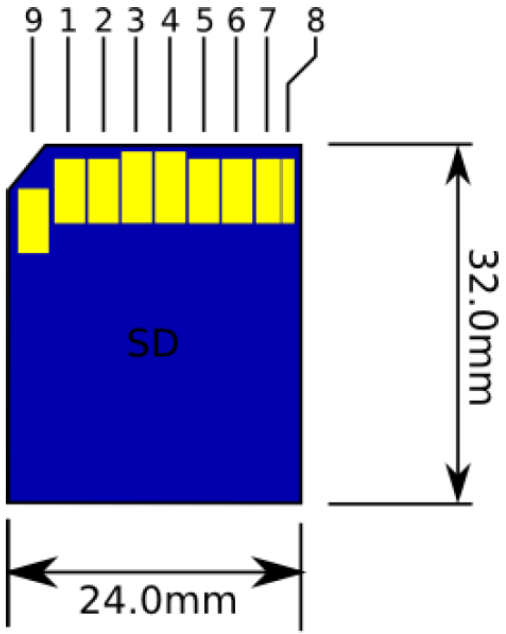


Version: 21-3-2022, Henning Hargaard

SD Card : Block Diagram

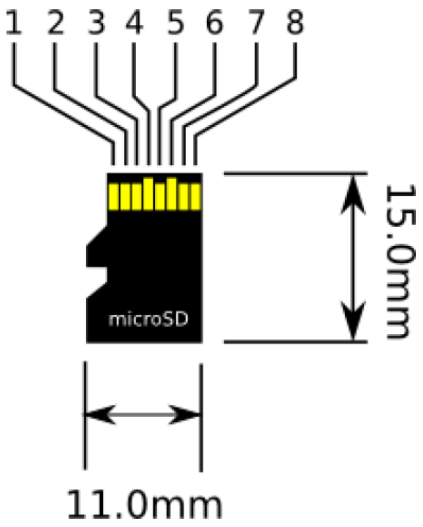


Standard SD card



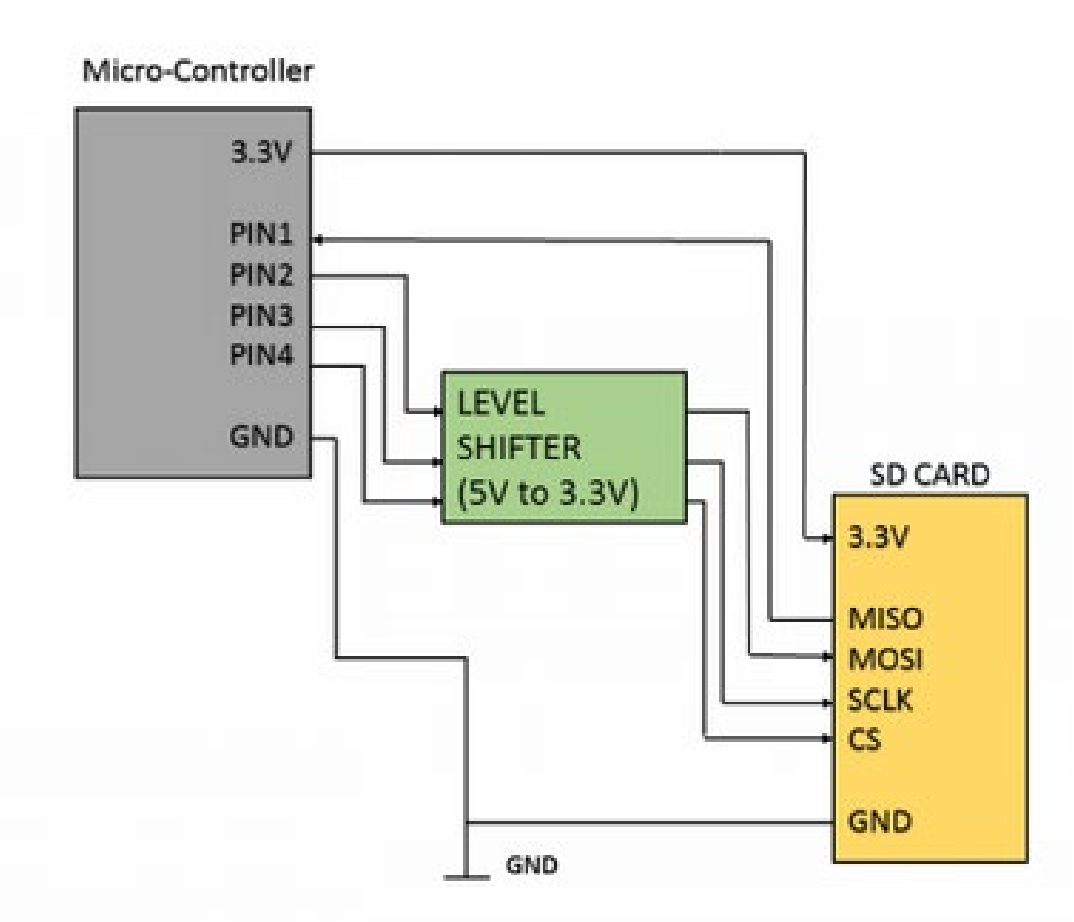
SD Card						
Pin No.	SD Mode			SPI Mode		
	Name	Type	Description	Name	Type	Description
1	CD/DAT	I/O/PP	Card Detect/Data Line [Bit 3]	CS	I	Chip Select (active low)
2	CMD	PP	Command/Response	DI/MOSI	I	Data In/Master Out Slave In
3	Gnd1/Vss1	S	Ground	GND/VSS	S	Ground
4	Vdd	S	Power (2.7V to 3.6V DC)	VDD	S	Power (2.7V to 3.6V DC)
5	CLK	I	Clock	SCLK	I	Clock
6	Gnd2/Vss2	S	Ground	Gnd2/Vss2	S	Ground
7	DAT0	I/O/PP	Data Line [Bit 0]	DO/MISO	O/PP	Data Out/Master In Slave Out
8	DAT1	I/O/PP	Data Line [Bit 1]	RSV		Reserved
9	DAT2	I/O/PP	Data Line [Bit 2]	RSV		Reserved

Micro SD card



microSD Card						
Pin No.	SD Mode			SPI Mode		
	Name	Type	Description	Name	Type	Description
1	DAT2	I/O/PP	Data Line [Bit 2]	RSV		Reserved
2	CD/DAT3	I/O/PP	Card Detect / Data Line [Bit 3]	CS	I	Chip Select
3	CMD	PP	Command/Response	DI/MOSI	I	Data In/Master Out Slave In
4	Vdd	S	Power	Vdd	S	Power
5	CLK	I	Clock	SCLK	I	Clock
6	Gnd/Vss	S	Ground	Gnd/Vss	S	Ground
7	DAT0	I/O/PP	Data Line [Bit 0]	DO/MISO	O/PP	Data Out/Master In Slave Out
8	DAT1	I/O/PP	Data Line [Bit 1]	RSV		Reserved

SD cards runs 3,3 volts

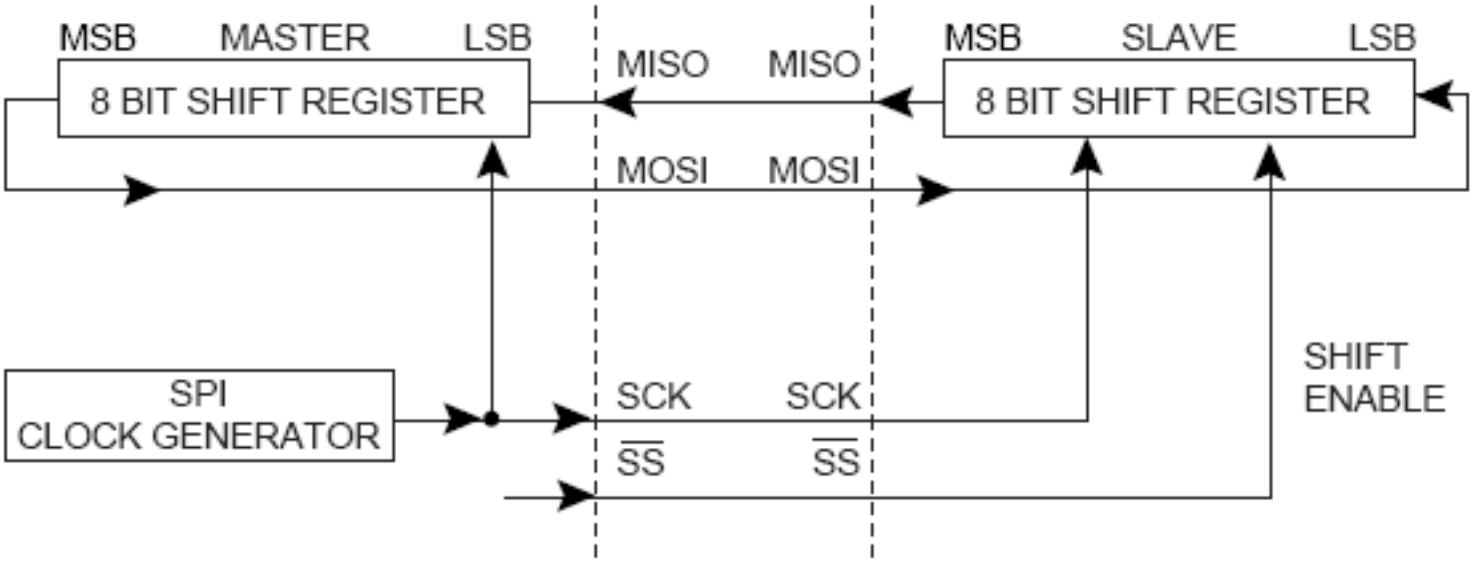


SPI clock frequency (worst case): 100 – 400 kHz

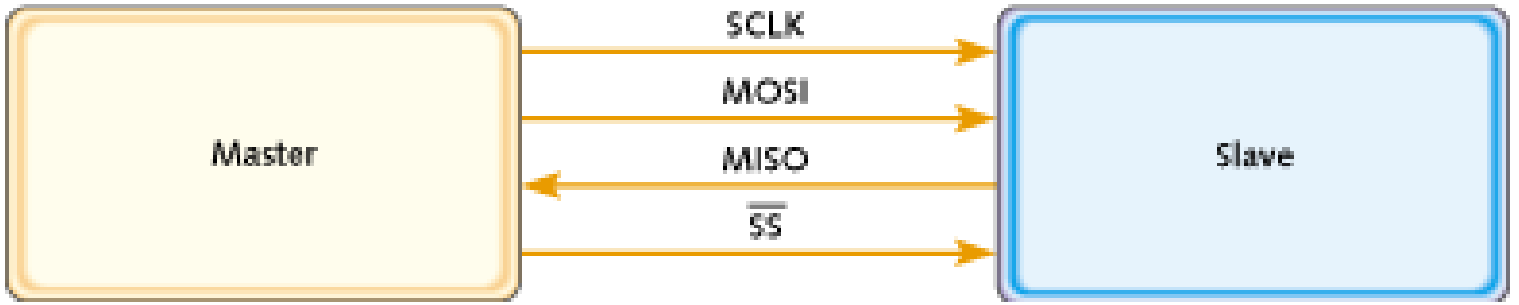
Mega2560 display interface card

D30(PC7)	DB7	-
D41(PG0)	RESET	-
D40(PG1)	CS	-
D39(PG2)	WR	-
D38(PD7)	RS	-
D50(PB3)	SD_OUT	-
D51(PB2)	SD_IN	-
D52(PB1)	SD_CLK	-
D53(PB0)	SD_CS	-
D6	D_CLK	-
D5	D_CS	-
D4	D_IN	-
D3	D_OUT	-
D2	D_IRQ	

SPI: Shift Register based

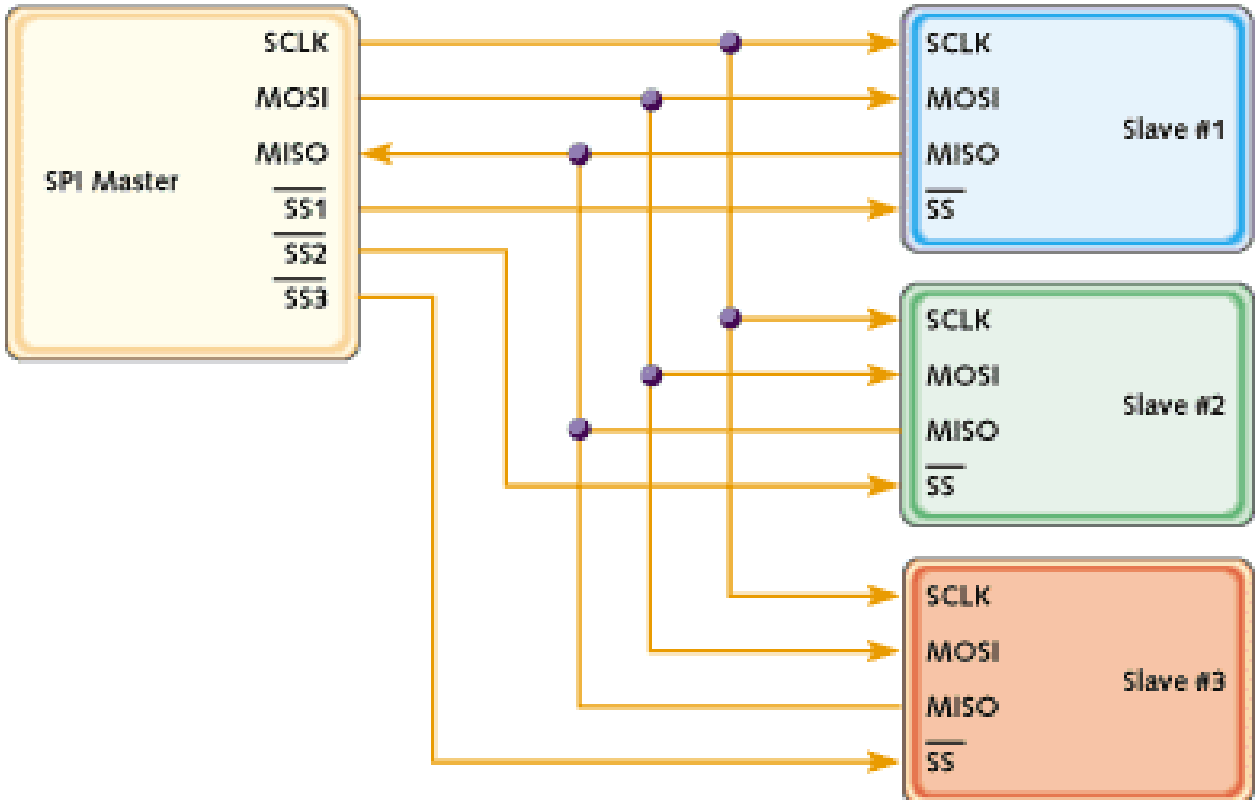


SPI: Master and slave



SPI-mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Slaves in parallel configuration

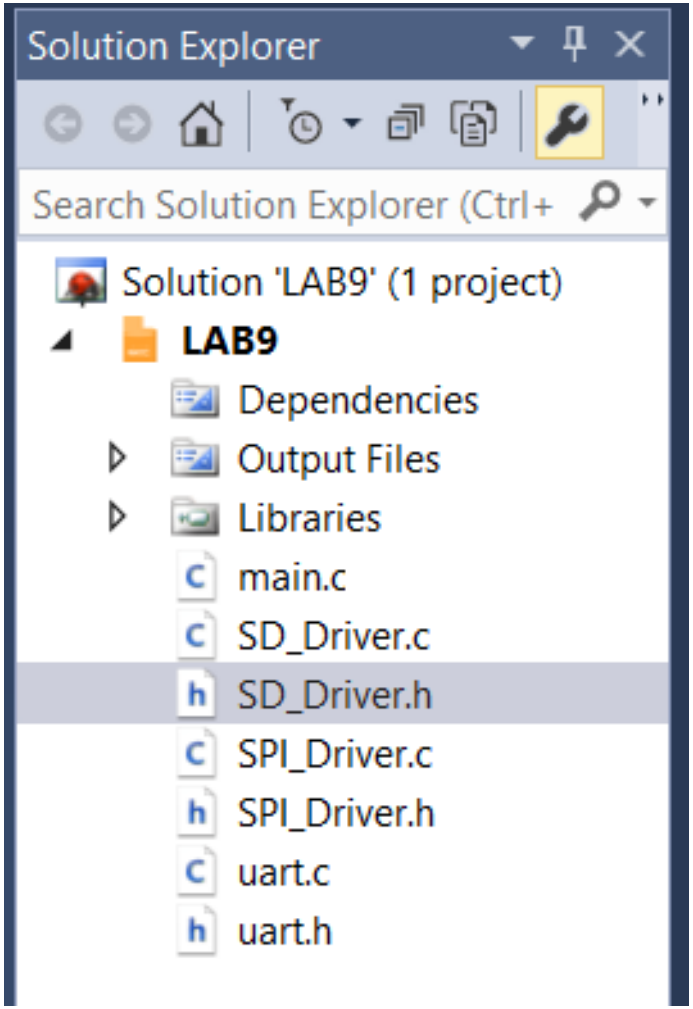


Mega2560: SPI interface

- Mega2560 has HARDWARE for SPI interface.

- Registers:
  SPI Control Register – SPCR
  SPI Status Register – SPSR
  SPI Data Register – SPDR

LAB9 files



SPI_driver.h

```
#define SPI_PORT PORTB
#define SPI_DDR  DDRB
#define SS_BIT   0
#define SCK_BIT  1
#define MOSI_BIT 2
#define MISO_BIT 3

void SPI_init();
void SPI_transmit(unsigned char);
unsigned char SPI_receive();
void SPI_Chip_Select();
void SPI_Chip_Deselect();
```

SPI_Driver.c

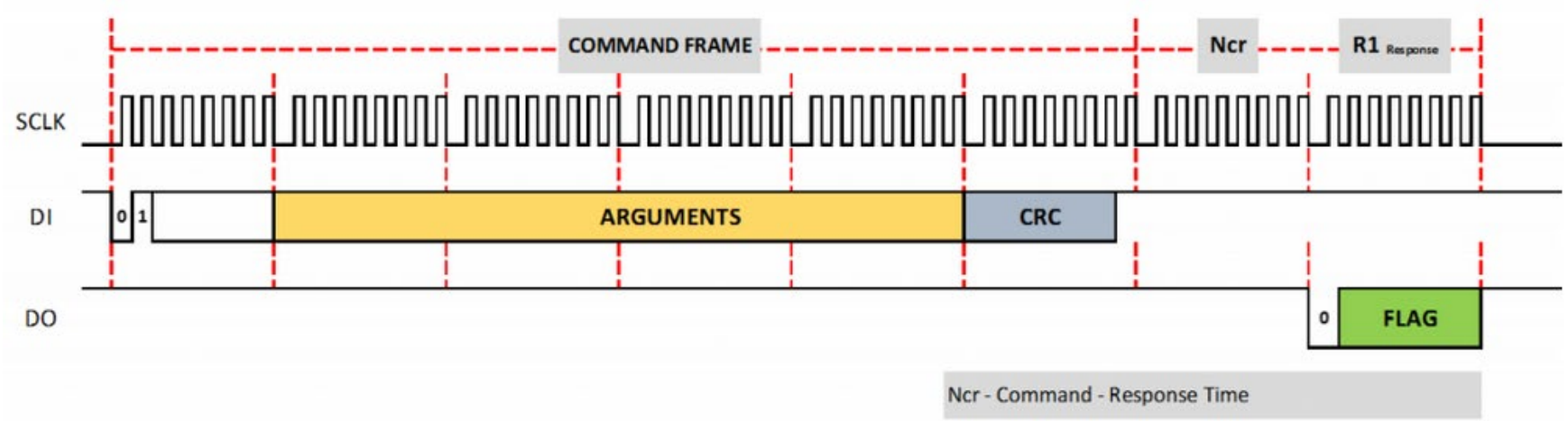
```
void SPI_init(void)
{
    SPI_DDR |= 1 << SS_BIT;
    SPI_DDR |= 1 << MOSI_BIT;
    SPI_DDR &= ~(1 << MISO_BIT);
    SPI_DDR |= 1 << SCK_BIT;
    //Setup SPI: Enable, Master mode, MSB first, SCK phase low, SCK idle low, f = fosc/64 = 16 MHz/64 = 250 kHz
    SPCR = 0b01010010;
    SPSR = 0;
}


void SPI_transmit(unsigned char data)
{
unsigned char dummy;

    // Start transmission
    SPDR = data;
    // Wait for transmission complete
    while(!(SPSR & (1<<SPIF)))
    {}
    // Clear flag
    dummy = SPDR;
}
```

SPI_Driver.c

```
unsigned char SPI_receive()
{
    unsigned char data;
    // Wait for reception complete

    SPDR = 0xff;
    while(!(SPSR & (1<<SPIF)));
    data = SPDR;

    // Return data register
    return data;
}
```

```
// CS active (=low)
void SPI_Chip_Select()
{
    SPI_PORT &= ~(1 << SS_BIT);
}

// CS inactive (=high)
void SPI_Chip_Deselect()
{
    SPI_PORT |= (1 << SS_BIT);
}
```

Command frame



The uC always is the master (controlling SCLK)

CRC only manditory for CMD0 (0x95) and CMD8 (0x87)

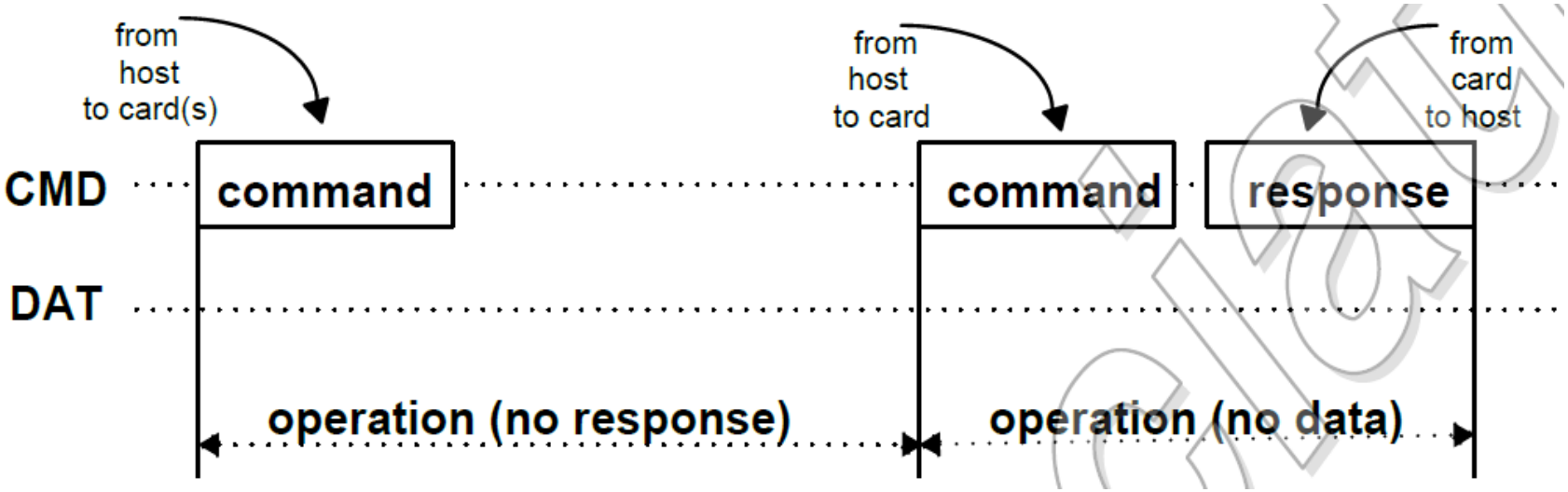
CS has to be low (active) during command / response

DI must be high after the CRC

Command token (= 6 bytes)

	Start Bit	Transmission Bit	Command Bit Pattern	Argument	CRC7	End Bit
Bit position	47	46	[45:40]	[39:8]	[7:1]	0
Width (Bits)	1	1	6	32	7	1
Value	0	1				1

No response / response



Commands (1 of 2)

COMMAND INDEX	ARGUMENT	RESPONSE	DATA	DESCRIPTION
CMD0	None	R1	NO	Software reset
CMD1	None	R1	NO	Initiate initialization process
ACMD41	2	R1	NO	For only SDC. Initiate initialization process
CMD8	3	R7	NO	For only SDC V2. Check voltage range.
CMD9	None	R1	YES	Read CSD register
CMD10	None	R1	YES	Read CID register
CMD12	None	R1b	NO	Stop to read data
CMD16	Block Length(31:0)	R1	NO	Change R/W block size

Commands (2 of 2)

COMMAND INDEX	ARGUMENT	RESPONSE	DATA	DESCRIPTION
CMD17	Address(31:0)	R1	YES	Read block
CMD18	Address(31:0)	R1	YES	Read multiple blocks
CMD23	Number of blocks(15:0)	R1	NO	For only MMC. Define number of blocks to transfer with next multi-block R/W command
ACMD23	Number of blocks(22:0)	R1	NO	For only SDC. Define number of blocks to pre-erase with next multi block write command
CMD24	Address(31:0)	R1	YES	Write a block
CMD25	Address(31:0)	R1	YES	Write multiple blocks
CMD55	None	R1	NO	Leading command of ACMD<n> command
CMD58	None	R3	NO	Read OCR

Response R1 (1 byte)

Bit		
0	In idle state	The card is in idle state and running the initializing process.
1	Erase reset	An erase sequence was cleared before executing because an out of erase sequence command was received.
2	Illegal command	An illegal command code was detected.
3	Communication CRC error	The CRC check of the last command failed.
4	Erase sequence error	An error in the sequence of erase commands occurred.
5	Address error	A misaligned address that did not match the block length was used in the command.
6	Parameter error	The command's argument (e.g. address, block length) was outside the allowed range for this card.
7	MSB	Always Zero

Response R1b = R1 + one or more "busy bytes"

Response R2 (2 bytes)

R2 = R1 + this byte :

Bit		
0	Card is locked	Set when the card is locked by the user. Reset when it is unlocked.
1	Write protect erase skip | lock/unlock command failed	This status bit has two functions overloaded. It is set when the host attempts to erase a write-protected sector or makes a sequence or password errors during card lock/unlock operation.
2	Error	A general or an unknown error occurred during the operation.
3	CC error	Internal card controller error.
4	Card ECC failed	Card internal ECC was applied but failed to correct the data.
5	Write protect violation	The command tried to write a write-protected block.
6	Erase param	An invalid selection for erase, sectors or groups.
7	out of range | csd overwrite	

UART.h

```
/*****************************************
* "uart.h":                              *
* Header file for Mega2560 UART driver.  *
* Using UART 0.                          *
* Henning Hargaard, 10/3 2020            *
*****************************************/
void InitUART(unsigned long BaudRate, unsigned char DataBit);
unsigned char CharReady();
char ReadChar();
void SendChar(char Tegn);
void SendString(char* Streng);
void SendInteger(int Tal);
void SendLong(long Tal);
/*****************************************/
```

SD_Driver.h

```
#include "SPI_Driver.h"

//SD commands, many of these are not used here
#define GO_IDLE_STATE            0
#define SEND_OP_COND             1
#define SEND_IF_COND             8
#define SEND_CSD                 9
#define STOP_TRANSMISSION        12
#define SEND_STATUS              13
#define SET_BLOCK_LEN            16
#define READ_SINGLE_BLOCK        17
#define READ_MULTIPLE_BLOCKS     18
#define WRITE_SINGLE_BLOCK       24
#define WRITE_MULTIPLE_BLOCKS    25
#define ERASE_BLOCK_START_ADDR   32
#define ERASE_BLOCK_END_ADDR     33
#define ERASE_SELECTED_BLOCKS    38
#define SD_SEND_OP_COND          41   //Application specific command
#define APP_CMD                  55
#define READ_OCR                 58
#define CRC_ON_OFF               59

#define ON     1
#define OFF    0
```

SD_Driver.h

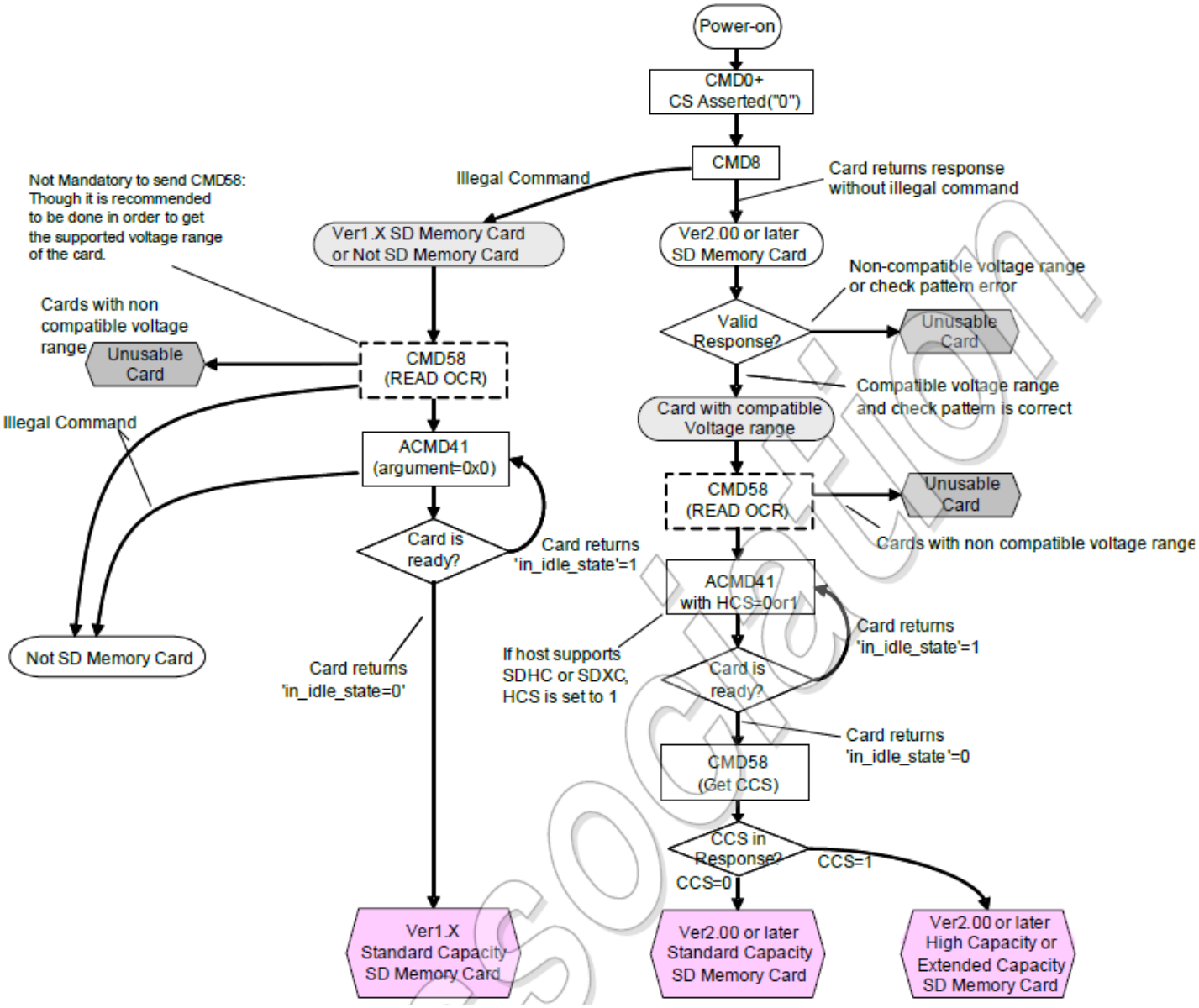
```
volatile unsigned long startBlock, totalBlocks;
volatile unsigned char SDHC_flag, cardType;

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);
```

Initialization

For the SD card to synchronize its clock: Send 8 bytes of dummy data (without CS active).

Only the commands, CMD0, CMD1, ACMD41, CMD58 and CMD59 will be accepted when the card is in its idle state.

SD card initialization



SD_Driver.c : SD_init()

```
//*****************************************************************
//Function  : To initialize the SD/SDHC card in SPI mode
//Arguments : None
//return    : unsigned char; will be 0 if no error,
//            otherwise the response byte will be sent
//*****************************************************************
unsigned char SD_init()
{
    unsigned char i, response, SD_version;
    unsigned int retry = 0;

    SPI_init();

    for(i = 0; i < 10; i++)
    {
        SPI_transmit(0xff); //80 clock pulses before sending the first command (Only needs 76, but we just do 80 to be sure)
    }

    SPI_Chip_Select();
    do
    {
        response = SD_sendCommand(GO_IDLE_STATE, 0); //send 'reset & go idle' command (= CMD0)
        retry++;
        if(retry > 0x20)
            return 1; //time out, card not detected
    } while(response != 0x01); //repeat until SD is in IDLE state

    SPI_Chip_Deselect();
    SPI_transmit (0xff);
    SPI_transmit (0xff);
```

SD_Driver.c : SD_init()

```
retry = 0;

SD_version = 2; //default set to SD compliance with ver2.x;
                //this may change after checking the next command
do
{
    response = SD_sendCommand(SEND_IF_COND, 0x000001AA); //Check power supply status, mandatory for SDHC card (= CMD8)
    retry++;
    if(retry > 0xfe)
    {
        SD_version = 1;
        cardType = 1;
        break;
    } //time out
} while(response != 0x01);

retry = 0;
do
{
    response = SD_sendCommand(APP_CMD, 0); //CMD55, must be sent before sending any ACMD command
    response = SD_sendCommand(SD_SEND_OP_COND, 0x40000000); //ACMD41

    retry++;
    if(retry > 0xfe)
        return 2;  //time out, card initialization failed
} while(response != 0x00);
```

When sending CMD8, we can set the VHS bit through its parameters to tell the SD card, the host's power supply, and let the SD card know the host's power supply range. Here we use the parameter 0X1AA, which tells the SD card that the power supply of the host is between 2.7~3.6V. If the SD card supports CMD8 and supports this voltage range, it will pass the response of CMD8 (R7, for SD card response, please refer to " SD Card 2.0 Protocol.pdf "Section 4.9) returns the original parameter part to the host, *if it does not support CMD8, or does not support this voltage range, it will not respond.*

Send ACMD41 (note: before sending ACMD41, send CMD55 first) to further confirm the operating voltage range of the card, and use the HCS bit to tell the SD card whether the host supports high-capacity cards (SDHC)

SD_Driver.c : SD_init()

```
    retry = 0;
    SDHC_flag = 0;

    if (SD_version == 2)
    {
        do
        {
            response = SD_sendCommand(READ_OCR, 0); // (=CMD58)
            retry++;
            if(retry > 0xfe)
            {
                cardType = 0;
                break;
            } //time out
        } while(response != 0x00);

        if(SDHC_flag == 1)
            cardType = 2;
        else
            cardType = 3;
    }

    SD_sendCommand(CRC_ON_OFF, OFF); //disable CRC; default - CRC disabled in SPI mode
    SD_sendCommand(SET_BLOCK_LEN, 512); //set block size to 512; default size is 512

    return 0; //successful return
}
```

SD_Driver.c : SD_sendCommand()

```
//*****************************************************************
//Function  : To send a command to SD card
//Arguments : unsigned char (8-bit command value)
//            & unsigned long (32-bit command argument)
//return    : unsigned char; response byte
//*****************************************************************
unsigned char SD_sendCommand(unsigned char cmd, unsigned long arg)
{
    unsigned char response, retry = 0, status;

    //SD card accepts byte address while SDHC accepts block address in multiples of 512
    //so, if it's SD card we need to convert block address into corresponding byte address by
    //multiplying it with 512. which is equivalent to shifting it left 9 times.
    //The following 'if' statement does that
    if(SDHC_flag == 0)
    {
        if(cmd == READ_SINGLE_BLOCK      ||
           cmd == READ_MULTIPLE_BLOCKS   ||
           cmd == WRITE_SINGLE_BLOCK     ||
           cmd == WRITE_MULTIPLE_BLOCKS  ||
           cmd == ERASE_BLOCK_START_ADDR ||
           cmd == ERASE_BLOCK_END_ADDR)
           {
               arg = arg << 9;
           }
    }
```

SD_Driver.c : SD_sendCommand()

```
SPI_Chip_Select();
SPI_transmit(cmd | 0b01000000); //send command, the first two bits are always '01'
SPI_transmit(arg >> 24);
SPI_transmit(arg >> 16);
SPI_transmit(arg >> 8);
SPI_transmit(arg);

if(cmd == SEND_IF_COND) //it is compulsory to send correct CRC for CMD8 (CRC=0x87) & CMD0 (CRC=0x95)
    SPI_transmit(0x87); //for remaining commands, CRC is ignored in SPI mode
else
    SPI_transmit(0x95);

while((response = SPI_receive()) == 0xff) //wait for response
{
    if(retry++ > 254)
        break; //time out error
}
```

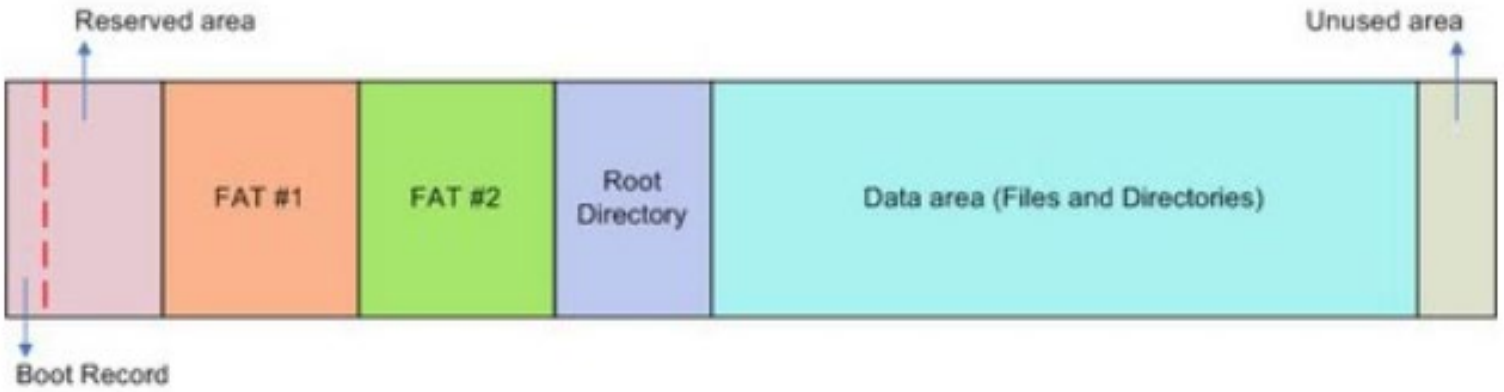
SD_Driver.c : SD_sendCommand()

```
    if(response == 0x00 && cmd == READ_OCR) //checking response of CMD58
    {
        status = SPI_receive() & 0x40; //first byte of the OCR register (bit 31:24)
        if(status == 0x40)
            SDHC_flag = 1; //we need it to verify SDHC card
        else
            SDHC_flag = 0;

        SPI_receive(); //remaining 3 bytes of the OCR register are ignored here
        SPI_receive(); //one can use these bytes to check power supply limits of SD
        SPI_receive();
    }

    // This is added by Henning Hargaard 6/3 2020 (Response = 1b => busy while reading 0)
    if (cmd == ERASE_SELECTED_BLOCKS)
    {
        while (SPI_receive() == 0)
        {}
    }

    SPI_receive(); //extra 8 CLK
    SPI_Chip_Deselect();
    return response; //return state
}
```

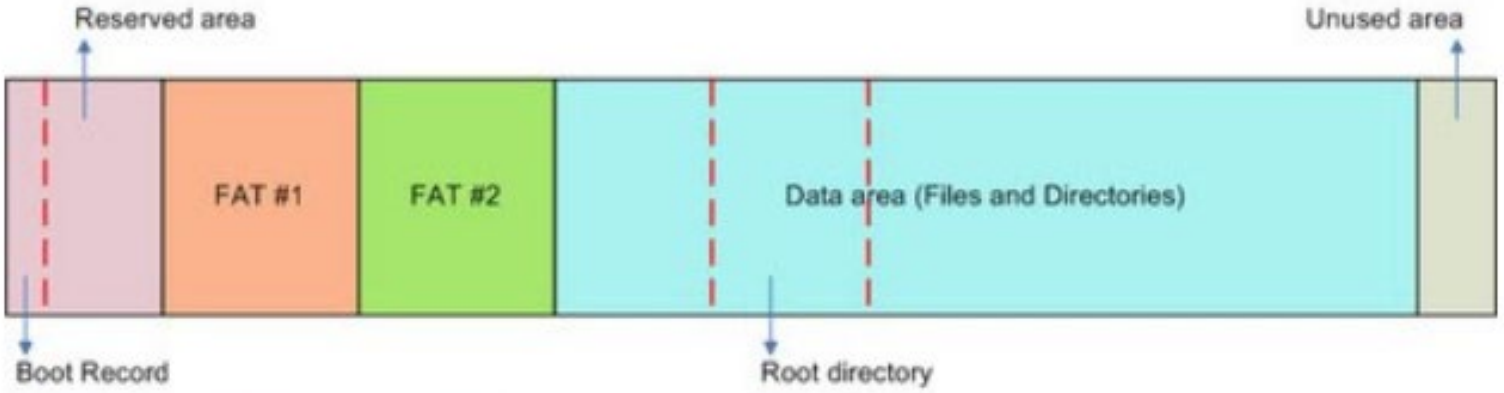
”To be implemented”

```
//*****************************************************************
//Function  : To erase specified no. of blocks of SD card
//Arguments : None
//return    : unsigned char; will be 0 if no error,
//            otherwise the response byte will be sent
//*****************************************************************
unsigned char SD_erase (unsigned long startBlock, unsigned long numberOfBlocks)
{
  // To be implemented
}

//*****************************************************************
//Function  : To read a single block from SD card
//Arguments : None
//return    : unsigned char; will be 0 if no error,
//            otherwise the response byte will be sent
//*****************************************************************
unsigned char SD_readSingleBlock(unsigned long startBlock, unsigned char* ptr)
{
  // To be implemented
}

//*****************************************************************
//Function  : To write to a single block of SD card
//Arguments : None
//return    : unsigned char; will be 0 if no error,
//            otherwise the response byte will be sent
//*****************************************************************
unsigned char SD_writeSingleBlock(unsigned long startBlock, unsigned char* ptr)
{
  // To be implemented
}
```

FAT file system



The structure of FAT16 file system

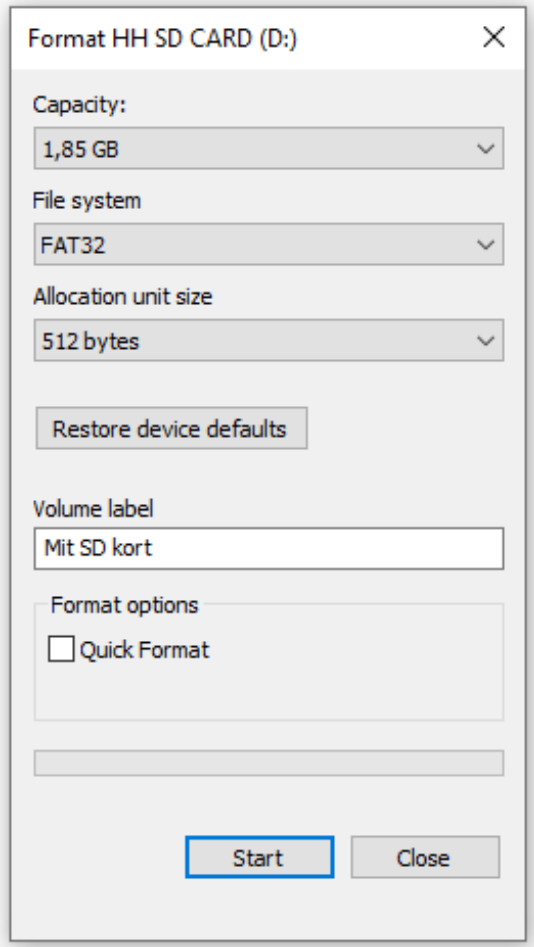


The structure of FAT32 file system

Video: FAT system

https://www.youtube.com/watch?v=V2Gxqv3bJCk

Format SD card

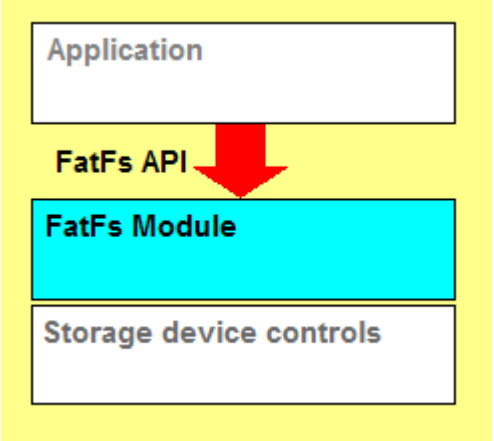


FatFs - Generic FAT Filesystem

FatFs is a generic FAT/exFAT filesystem module for small embedded systems. The FatFs module is written in compliance with ANSI C (C89) and completely separated from the disk I/O layer. Therefore it is independent of the platform. It can be incorporated into small microcontrollers with limited resource, such as 8051, PIC, AVR, ARM, Z80, RX and etc. Also Petit FatFs module for tiny microcontrollers is available

Features

- DOS/Windows compatible FAT/exFAT filesystem.
- Platform independent. Easy to port.
- Very small footprint for program code and work area.
- Various configuration options to support for:
  - Long file name in ANSI/OEM or Unicode.
  - exFAT filesystem, 64-bit LBA and GPT for huge storages.
  - Thread safe for RTOS.
  - Multiple volumes (physical drives and partitions).
  - Variable sector size.
  - Multiple code pages including DBCS.
  - Read-only, optional API, I/O buffer and etc...



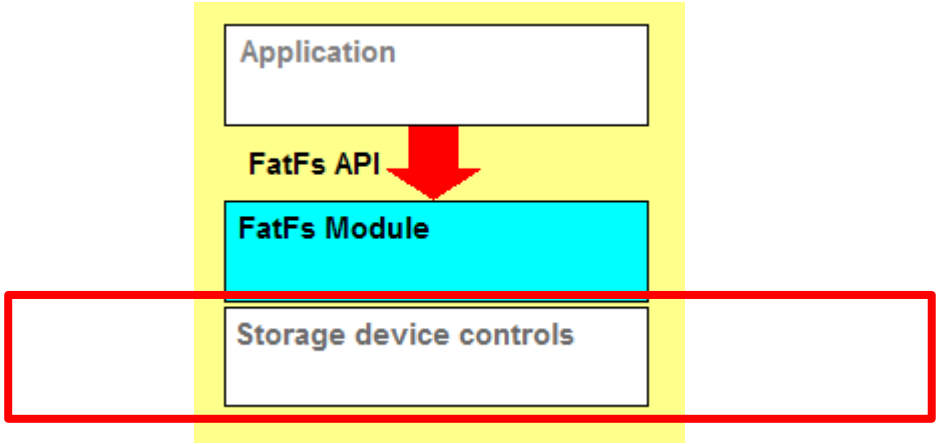
FatFs API

- File Access
  - f_open - Open/Create a file
  - f_close - Close an open file
  - f_read - Read data from the file
  - f_write - Write data to the file
  - f_lseek - Move read/write pointer, Expand size
  - f_truncate - Truncate file size
  - f_sync - Flush cached data
  - f_forward - Forward data to the stream
  - f_expand - Allocate a contiguous block to the file
  - f_gets - Read a string
  - f_putc - Write a character
  - f_puts - Write a string
  - f_printf - Write a formatted string
  - f_tell - Get current read/write pointer
  - f_eof - Test for end-of-file
  - f_size - Get size
  - f_error - Test for an error
- Directory Access
  - f_opendir - Open a directory
  - f_closedir - Close an open directory
  - f_readdir - Read a directory item
  - f_findfirst - Open a directory and read the first item matched
  - f_findnext - Read a next item matched
- File and Directory Management
  - f_stat - Check existance of a file or sub-directory
  - f_unlink - Remove a file or sub-directory
  - f_rename - Rename/Move a file or sub-directory
  - f_chmod - Change attribute of a file or sub-directory
  - f_utime - Change timestamp of a file or sub-directory
  - f_mkdir - Create a sub-directory
  - f_chdir - Change current directory
  - f_chdrive - Change current drive
  - f_getcwd - Retrieve the current directory and drive
- Volume Management and System Configuration
  - f_mount - Register/Unregister the work area of the volume
  - f_mkfs - Create an FAT volume on the logical drive
  - f_fdisk - Create partitions on the physical drive
  - f_getfree - Get free space on the volume
  - f_getlabel - Get volume label
  - f_setlabel - Set volume label
  - f_setcp - Set active code page

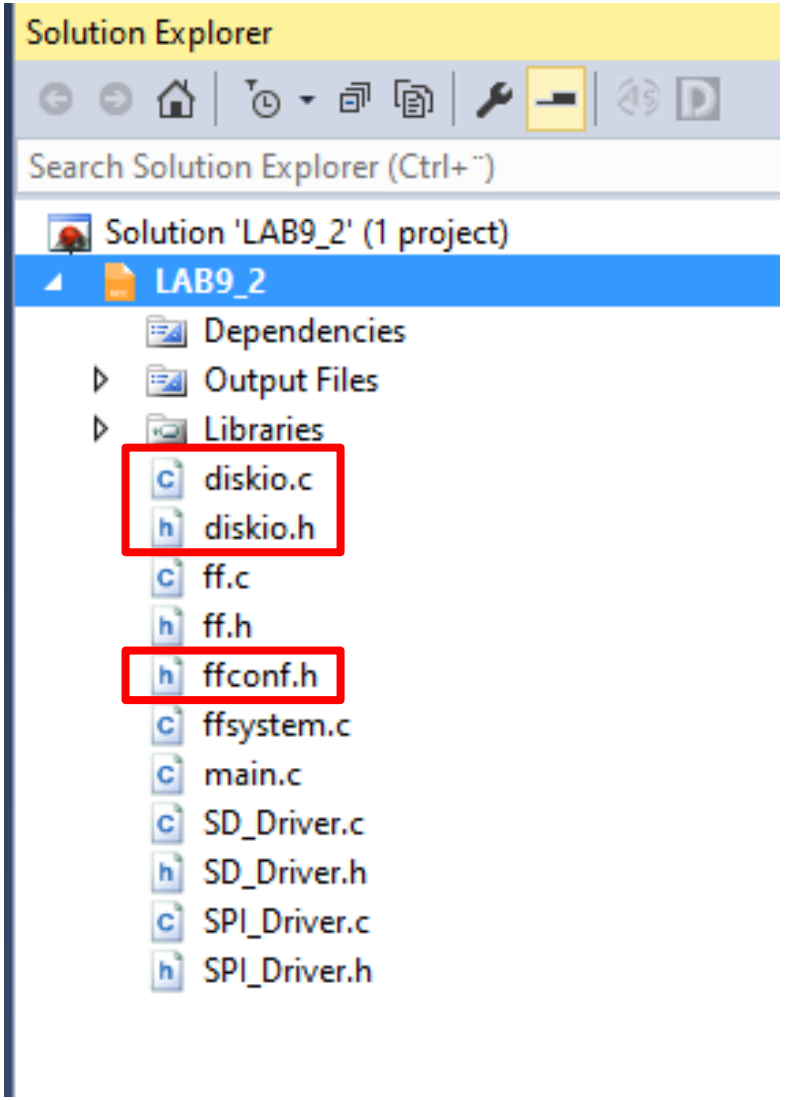
FatFs: Media Access Interface

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



FasFs: Atmel Studio project



FasFs: Test program

```
/*-----------------------------------------------------------------------*/
/* 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)
    {}
}
```

Disk editor

https://www.disk-editor.org/index.html

End of lesson 7