



TRANSLATE

Select Language 

Powered by  Google Translate

CATEGORIES

- 7 Segment Display (1)
- ADC Library (1)
- Apps (1)
- Beginner (2)
- Bit Code Modulation (1)
- Bitwise Operators (1)
- Bluetooth (1)
- Button Reading (2)
- CAD Eagle (4)
- Chiptunes (1)
- Clock - Digital (1)
- Code Utils EFM8 (2)
- Colorspace - RGB vs HSL (1)
- DAC Library (2)
- Dev Board (1)
- Display Library (4)
- EEPROM Library - AVR (1)
- Electronics Storage (1)
- FAT (1)
- Flashlight (1)
- Home Appliance - Washing Machine (1)

WEDNESDAY, JULY 13, 2022

SD card tutorial - Interfacing an SD card with a microcontroller over SPI (part 1 of 2)

This is a two-part series tutorial about the SD protocol used by SD flash memory cards.

Personally I learn better using practical examples instead of abstract data, and for this reason I have constructed this tutorial as a step by step with practical code examples written in C language, that can be followed by anyone with basic programming skills and knowledge on how to use a microcontroller. Although I am using an ATmega328P in this tutorial, the concepts extend to any microcontroller.

If you need an SD card library, you can find one here: <https://www.programming-electronics-diy.xyz/2022/07/sd-memory-card-library-for-avr.html>. This tutorial will help in understanding the low-level interactions between a microcontroller and a memory card but it will not be enough to use an SD card in a practical way. Apart from an SD card driver, that is the code described in this articles, you will also need a file system driver such as FAT16 or FAT32 in order to read or write files.

Contents

1. General Description
 - 1.1 Bus Mode and Clock Speed
 - 1.2 Read/Write Mode Selection
2. SD Card Hardware Interface
 - 2.1 microSD Card Schematic SPI Interface
3. SPI Setup
4. Card Initialization
 - 4.1 Power Up Sequence
 - 4.2 Sending Commands
 - 4.3 Initialization Flow
 - 4.4 CMD0

SEARCH ON BLOG

Support the blog

Any donation is appreciated

Donate



Get new posts by email:

Enter your email

Subscribe

Powered by  follow.it

FEATURED POST

Plotting data from a logic analyzer - CSV and LibreOffice Calc

Sometimes it is useful to visualize data from a logic analyzer in a graphical way but unfortunately not many applications can do that so I c...

[Humidity and Temperature \(1\)](#)
[I2C - TWI \(1\)](#)
[I2C - TWI Library \(1\)](#)
[Infrared \(1\)](#)
[IoT - ESP8266 Library \(2\)](#)
[LCD Driver IC \(1\)](#)
[LCD Library \(2\)](#)
[LCD Library - EFM8 \(1\)](#)
[LCD Library Alpha Numeric Module \(1\)](#)
[LED Controller \(3\)](#)
[Logic Analyzer \(1\)](#)
[Mains Protection \(1\)](#)
[Memory Card \(3\)](#)
[Microcontroller AVR \(26\)](#)
[Microcontroller EFM8 \(4\)](#)
[Microcontroller EFM8BB1 \(6\)](#)
[Microcontroller EFM8BB3 \(3\)](#)
[PCB Homemade \(3\)](#)
[Piezo Buzzer Library \(1\)](#)
[Plot CSV \(1\)](#)
[POV Display \(2\)](#)
[Power Supply - Capacitive \(1\)](#)
[Programming AVR \(4\)](#)
[PWM - Software \(1\)](#)
[Remote Control \(1\)](#)
[RGB Library \(1\)](#)
[Salvage Electronics \(1\)](#)
[SD Card \(3\)](#)
[Sensor - PIR \(2\)](#)
[Sensor Library \(2\)](#)

[4.5 Response R1](#)

[4.6 CMD8](#)

[4.7 Response R7](#)

[4.9 CMD58](#)

[4.10 Response R3](#)

[4.11 ACMD41 & CMD55](#)

[5. Reading/Writing Data Blocks](#)

[5.1 Block Length](#)

[5.2 CMD17 – Reading a Single Block](#)

[5.2.1 Read Errors](#)

[5.3 CMD24 – Writing a Single Block](#)

[6. Links](#)



[1. GENERAL DESCRIPTION](#)

The Secure Digital (SD) Card was developed by the [SD Association \(SDA\)](#) as an improvement over MMCs. The SD Card specifications were originally defined by MEI (Matsushita Electric Company), Toshiba Corporation and SanDisk Corporation. Currently, the specifications are controlled by the Secure Digital Association (SDA).

In addition to the mass storage specific flash memory chip, the SD Card includes an on-card intelligent controller which manages interface protocols, security algorithms for copyright protection, data storage and retrieval, as



[POPULAR POSTS](#)



[Library for A4988 stepper motor driver using timer interrupt](#)

This library is designed for AVR ATmega328 microcontroller, but with few adjustments can work with any AVR microcontroller with at least 3-4...



[Library for interfacing alphanumeric LCD modules with AVR microcontrollers](#)

This library provides an interface between the microcontroller and LCD module. Note that this code is not for I2C modules. Main features: ...



[How an outdoor motion PIR sensor switch works with schematic](#)

In this article I will be explaining how a pyroelectric (PIR) sensor works and show the reverse engineered schematic simulated in LTspice...



[Playing music and tones using a piezo buzzer - library for AVR microcontrollers](#)

[using timer interrupt | ATmega328](#)

- [Stepper Motor Drivers](#) (1)
- [Stepper Motor Library](#) (1)
- [Temperature Sensor](#) (1)
- [Timing Library](#) (1)
- [UART Library](#) (2)
- [USB to UART](#) (1)
- [UV Exposure Box](#) (1)
- [VFD Display](#) (2)
- [Watchdog Library](#) (1)
- [Wireless Energy](#) (1)
- [Zener and LED Tester](#) (1)

CONTACT

Name

Email *

Message *

Send

well as Error Correction Code (ECC) algorithms, defect handling and diagnostics, power management and clock control.

The data is transferred between the memory card and the host controller as data blocks in units of 512 bytes. The SD Card communication is based on an advanced nine-pin interface (Clock, Command, 4xData and 3xPower lines) designed to operate in a low voltage range. Both MMC/SD standards have their own proprietary protocols but they also support SPI which can be selected during card initialization. Since microcontrollers have SPI integrated hardware, this is the most used interface for memory cards.

SPI mode is a secondary mode of communication for SD cards – they power up in an "SD Bus protocol mode." To switch the card to SPI, the Chip Select line is driven low and CMD0 is sent.

The recommended file systems for memory cards are FAT12/16/32 or exFAT. Maximum volume sizes for each one are: 256MB, 4GB, 16TB* for FAT32 and 128PB for exFAT.

1.1 BUS MODE AND CLOCK SPEED

The MultiMediaCard and SD Card support multiple busses. Both cards support the 1-bit SPI bus that includes bus pins DATin, DATout, CLK, and CS.

The SD Card also supports a 4-bit and a 1-bit SD bi-directional bus mode. SD bus pins are CLK, CMD, and DATin 1-bit mode and CLK, CMD, and DAT[0:3] in 4-bit mode.

The MultiMediaCard also supports the 1-bit bi-directional MMC bus mode that has CLK, CMD, and DAT bus pins.

The CMD and DAT pins are bi-directional on the SD 1-bit, SD 4-bit, and MMC 1-bit.

The maximum burst rate achievable with the SD Card and MultiMediaCard depends on the clock speed and bus mode. The burst rate is the data transfer rate between the card's buffer and host.

Table 2. MultiMediaCard and SD Card Clock Speed and Burst Rate

Product	Maximum Clock Speed and Burst Rate	
MultiMediaCard	Clock Speed	Burst Rate
SPI Bus mode	20 MHz	2.5 MB/s
MMC 1-bit mode	20 MHz	2.5 MB/s
SD Card		
SPI Bus mode	25 MHz	3.125 MB/s
SD 1-bit mode	25 MHz	3.125 MB/s
SD 4-bit mode	25 MHz	12.5 MB/s

Table 1: from "SanDisk Secure Digital Card v1.9"

This is a library for playing monophonic music using PWM and a piezoelectric buzzer. Monophonic means it can play only one note at a time...



[AVR EEPROM Library | ATmega328P](#)

Sometimes some data needs to be

saved after the microcontroller is powered off. Say you have a rotary encoder to set audio volume. When the ...

BLOG ARCHIVE

- [2023](#) (2)
- ▼ [2022](#) (9)
 - [August](#) (3)
 - ▼ [July](#) (4)

[Library for MCP4725 DAC for AVR microcontrollers](#)

[SD card tutorial - Interfacing an SD card with a m...](#)

[SD card tutorial - Interfacing an SD card with a m...](#)

[SD Memory Card Library for AVR Microcontrollers - ...](#)

- [May](#) (1)
- [February](#) (1)

- [2021](#) (16)
- [2020](#) (8)
- [2019](#) (3)
- [2018](#) (16)
- [2017](#) (8)
- [2016](#) (2)

The write and read throughput rates of the SD Card and MultiMediaCard are slower than the burst rate because each card includes the busy time to write data from the card's buffers to its internal Flash RAM, and busy time to read data from the internal Flash RAM to the card's buffer. Since most designs use this write and read busy time to complete other processes, choosing a 1- or 4-bit bus mode can have a 4x speed effect on the time spent servicing the SD Card.

1.2 READ/WRITE MODE SELECTION

Another major MultiMediaCard and SD Card design consideration is the use of Singleblock or Multiblock command modes. Singleblock mode reads and writes data one block at a time; Multiblock mode reads and writes multiple blocks until a stop command is received.

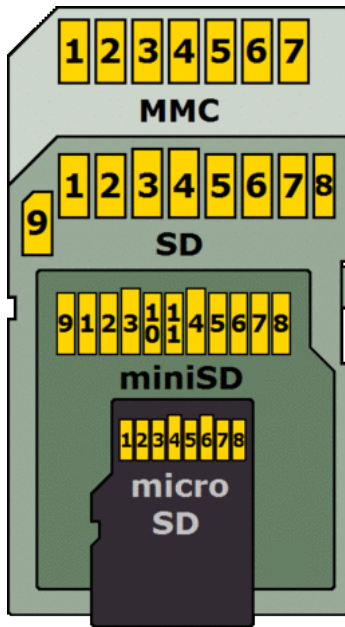
Multiblock mode takes advantage of the multiple internal block buffers present in all MultiMediaCards or SD Cards. In Multiblock mode, when one block buffer gets full during write, the card gives the host access to the other empty block buffers to fill while programming the first block. The card does not enter a busy state until all block buffers are full.

In Singleblock mode, the card enters a busy state by forcing the DAT line low when the first block buffer is full and remains busy until the write process is complete. During the busy state, the host cannot send any additional data to the card because the card forces the DAT line low.

If speed is critical in a design, Multiblock mode is the faster and recommended mode. The more blocks that can be written in Multiblock mode the better the performance of the design. Therefore when planning the design, ensure that enough system RAM is designed in to support the multiblock capability. The performance gain will always outweigh the cost of the extra RAM. However, if speed is not critical – for example, a data-logger design that records only 512 bytes of data every minute – Singleblock mode is more than adequate.

Reading and writing to an SD Card and MultiMediaCard is generally done in 512 byte blocks, however, erasing often occurs in much larger blocks. The NAND architecture used by SanDisk and other card vendors currently has Erase Block sizes of (32) or (64) 512 byte blocks, depending on card capacity. In order to re-write a single 512 byte block, all other blocks belonging to the same Erase Block will be simultaneously erased and need to be rewritten.

2. SD CARD HARDWARE INTERFACE

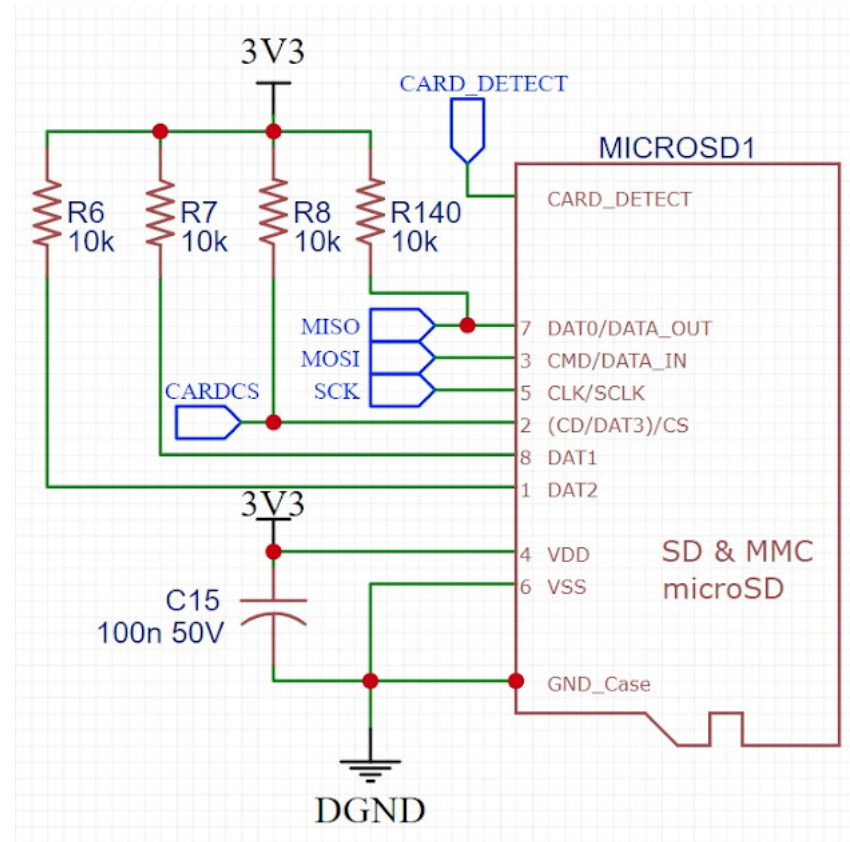


	SDC		miniSD		- SD - SPI
PIN	SD	SPI	SD	SPI	Description
1	DAT3	CS	DAT3	CS	- SD Serial Data 3 - SPI Card Select (active Low)
2	CMD	DI	CMD	DI	- Command, response - SPI Serial Data In (MOSI)
3	VSS		VSS		Ground
4	VDD		VDD		Power. Usually 2.7 to 3.6V and minimum 100mA. Not 5V tolerant!
5	CLK		CLK		Serial Clock
6	VSS		VSS		Ground
7	DAT0	DO	DAT0	DO	- SD Serial Data 0 - SPI Serial Data Out (MISO)
8	DAT1	NC	DAT1	NC	- SD Serial Data 1 - Unused
9	DAT2	NC	DAT2	NC	- SD Serial Data 2 - Unused
10			NC		Not Connected

	microSD		- SD - SPI
PIN	SD	SPI	Description
1	DAT2	NC	- SD Serial Data 2 - Unused
2	DAT3	CS	- SD Serial Data 3 - SPI Card Select (active Low)
3	CMD	DI	- Command, response - SPI Serial Data In (MOSI)
4	VDD		Power. Usually 2.7 to 3.6V and minimum 100mA. Not 5V tolerant!
5	CLK		Serial Clock
6	VSS		Ground
7	DAT0	DO	- SD Serial Data 0 - SPI Serial Data Out (MISO)
8	DAT1	NC	- SD Serial Data 1 - Unused

	SDC	miniSD	- SD - SPI
11		NC	Not Connected

2.1 MICROSD CARD SCHEMATIC SPI INTERFACE



The microcontroller and the memory card must use the same voltage. If the microcontroller is powered from 5V and the card is on 3.3V there needs to be a logic voltage level shifter such as 74AHC125D.

According to the SD card specification, unused Data lines on the card must be terminated with pull-up resistors. Floating pins can lead to excessive power consumption.

A pull-up on the DO line is needed because, until initialized into SPI mode, the DO pin is open drain.

A pull-up on the CS (Card Select) is recommended because it could take many milliseconds until the microcontroller enables the CS pin High and during that time the SPI pins could have random values and possibly corrupting the card.

Some sockets have a 9th pin that optionally can be pulled high using a microcontroller pin set as an input high. This is a mechanical switch. When the card is inserted the pin goes Low allowing the microcontroller to detect card insertion/removal.

3. SPI SETUP

To be able to access the SD card we first need to set up the SPI on the microcontroller. The SPI clock (CLK) idles low and the output is sampled on the leading edge, which corresponds to CPOL = 0 and CPHA = 0. Since this is the default mode on our micro, we don't have to change the mode in SPCR (SPI Control Register). Although modern SD Cards can operate at very high speeds it is recommended to set SCK to a frequency between 100kHz and 400kHz during the initialization process and then change it to maximum speed.

It is a good practice to abstract the pins used by using defines. This way it will be easier to change the pins later on.

```
// SPI0 I/O pins (MOSI and SCK)
#define SPI0_PORT          PORTB // SPI PORT
#define SPI0_DDR           DDRB  // SPI DDR
#define SPI0_PINS          PINB  // State of each pin on SPI port

#define SPI0_MOSI_PIN      PB3   // SDA pin
#define SPI0_SCK_PIN       PB5   // SCK pin

// SPI Chip Select pin
#define CARD_CS_DDR        DDRB
#define CARD_CS_PORT       PORTB
#define CARD_CS_PIN        PB2
```

The CS (Card Select or Chip Select) is the SPI SS0 pin and is set to PB2 but can be changed to any other port and pin. Regardless of what CS pin you choose to use it is important that the SS SPI pin be set as an output high or low otherwise the SPI won't work properly.

All SPI commands to the SD card require the microcontroller to first assert the Chip Select line, which means pulling it low. When we are done, we will need to return the Chip Select to a high state. Some pre-processor macros suits this purpose very well.

```
#define SPI_CARD_CS_DISABLE()    CARD_CS_PORT |= (1 << CARD_CS_PIN)
#define SPI_CARD_CS_ENABLE()    CARD_CS_PORT &= ~(1 << CARD_CS_PIN)
```

Now that we defined our pins let's create a function for SPI initialization. Making the function static will decrease the code size.


```

/*
    _____
    SPI – Initialization
    _____ */

static void SPI_Init(void){
    // Set MOSI and SCK as output
    SPI0_DDR |= (1 << SPI0_MOSI_PIN) | (1 << SPI0_SCK_PIN);
    CARD_CS_DDR |= (1 << CARD_CS_PIN); // CS pin

    // Enable SPI, Master mode
    SPCR0 = (1 << SPE0) | (1 << MSTR0);

    // Start with a lower speed for older cards (100-400kHz)
    SPSR0 = (1 << 0);
    SPCR0 |= (1 << SPR01); // set clock rate fosc/32
}

```

This function sets CS, MOSI and SCK pins as output, the SPI is enabled in Master mode and the SPI clock prescaler is set to 32 for a lower frequency.

Next function is used to receive one byte from the SPI. Because the memory card needs a clock pulse to send the bytes, we need to put some dummy data on the transmit line.

```

/*
    _____
    SPI – Receive and returns one byte only
    _____ */

static uint8_t SPI_ReceiveByte(void){
    // Write dummy byte out to generate clock, then read data from MISO
    SPDR0 = 0xFF;

    // Wait until the Busy bit is cleared
    while(!(SPSR0 & (1 << SPIF0)));

    return SPDR0;
}

```

To send a byte is even simpler:

```

/*
    _____
    SPI – Send a byte
    _____ */

static void SPI_SendByte(uint8_t byte){
    SPDR0 = byte;

    // Wait until the Busy bit is cleared
    while(!(SPSR0 & (1 << SPIF0)));
}

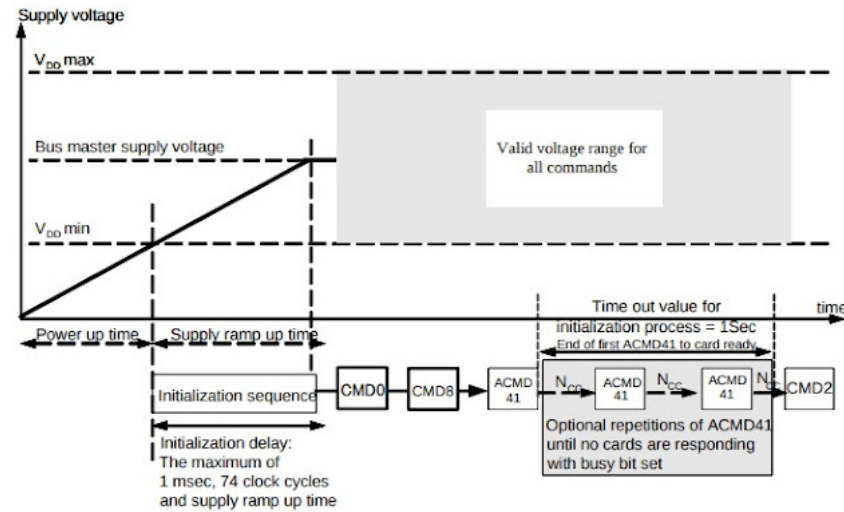
```


Now that the SPI is ready we can proceed with card initialization.

4. CARD INITIALIZATION

4.1 POWER UP SEQUENCE

The power up sequence from the physical spec is shown below.



Looking at the above diagram we see that it needs at least 1 msec delay and 74 clock cycles before sending commands to the SD Card. Since we get 8 clock cycles with each byte, we can send 10 bytes for a total of 80 clock cycles. In the notes of this section, the spec also specifies that CS must be held high during this period.

Before making a function to initialize the SD card we need to define some return codes to see if the function succeeded and if not what was the cause.

```
/* Card initialization return codes (SD_RETURN_CODES) */  
typedef enum {  
    SD_OK,  
    SD_IDLE_STATE_TIMEOUT,  
    SD_GENERAL_ERROR,  
    SD_CHECK_PATTERN_MISMATCH,  
    SD_NONCOMPATIBLE_VOLTAGE_RANGE,  
    SD_POWER_UP_BIT_NOT_SET,  
    SD_NOT_SD_CARD  
} SD_RETURN_CODES;
```

```

SD_RETURN_CODES sd_init(void){
    uint8_t SD_Response[5]; // array to hold response
    uint8_t cmdAttempts = 0;

    // SPI Setup
    SPI_Init();

    // Give SD card time to power up
    _delay_ms(1);

    // Send 80 clock cycles to synchronize
    // The card must be deselected during this time
    sd_deassert_cs();
    for(uint8_t i = 0; i < 10; i++) SPI_SendByte(0xFF);

    return SD_OK;
}

```

We will be adding more code to this function as we go along. You might have noticed a new function and actually we will need to like this: assert() and deassert().

```

/*
    Assert Chip Select Line by sending a byte
    before and after changing the CS line
*/
static void sd_assert_cs(void){
    SPI_SendByte(0xFF);
    SPI_CARD_CS_ENABLE();
    SPI_SendByte(0xFF);
}

/*
    De-Assert Chip Select Line by sending a byte
    before and after changing the CS line
*/
static void sd_deassert_cs(void){
    SPI_SendByte(0xFF);
    SPI_CARD_CS_DISABLE();
    SPI_SendByte(0xFF);
}

```

Both functions provide 8 clocks before and after changing the state of CS pin. This is recommended to ensure the card recognizes the change in CS. These extra bytes are not always necessary but many people have had issues when multiple cards are on the bus.

4.2 SENDING COMMANDS

Bit position	47	46	[45:40]	[39:8]	[7:1]	0
Width (bits)	1	1	6	32	7	1
Value	'0'	'1'	x	x	x	'1'
Description	start bit	transmission bit	command index	argument	CRC7	end bit

Table 7-1 : Command Format

SD card commands are 6 bytes long (48 bits) and contain a command index, arguments, and CRC. The **command index** field is used to tell the SD card which command you are sending. For example, if CMD1 is required, then the 6 bits in command index should be set to 000001b. The **argument field** is used in some commands and is ignored by other. Whenever no arguments are needed, we will fill this field with zeros. The Cyclic Redundancy Check (**CRC**) is used to ensure that the data has been correctly received by the SD card. In SPI mode the CRC is optional with few exceptions but in those cases, the spec manual provides us with pre-calculated values. In the cases where the CRC is not needed, we will set it to zero.

To send commands to the SD card we need a new function that takes an 8-bit command index, a 32-bit argument, and an 8-bit CRC.

```
/*  
    Send a command to the card  
  
    cmd                8-bit command index  
    arg                a 32-bit argument  
    crc                an 8-bit CRC  
*/  
static void sd_command(uint8_t cmd, uint32_t arg, uint8_t crc){  
    // Transmit command  
    SPI_SendByte(cmd | 0x40);  
  
    // Transmit argument  
    SPI_SendByte((uint8_t)(arg >> 24));  
    SPI_SendByte((uint8_t)(arg >> 16));  
    SPI_SendByte((uint8_t)(arg >> 8));  
    SPI_SendByte((uint8_t)(arg));  
  
    // Transmit CRC  
    SPI_SendByte(crc | 0x01);  
}
```

Note in the command format table, that the command index is only 6 bits long. The 2 most significant bits of the command is always set to 01b. In order to set bit 47 to 1, we need to bitwise OR our input 'cmd' with 0x40.

The CRC is only 7 bits long, and the end bit of any command is always 1 so we bitwise OR the 'crc' argument with 0x01.

4.3 INITIALIZATION FLOW

The SPI mode initialization flow for an SD card is shown in the diagram below. It might look complicated but is not really.

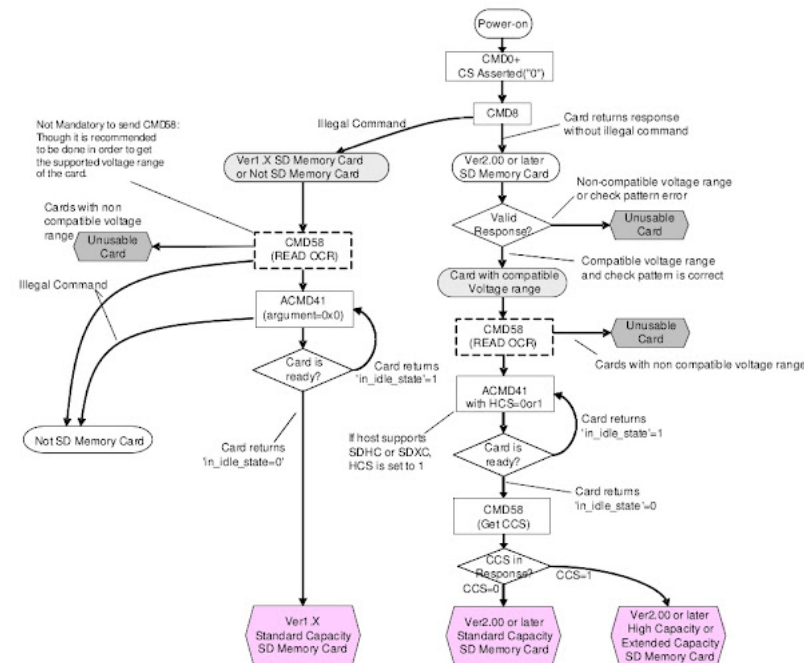


Figure 7-2 : SPI Mode Initialization Flow

From Power-on stage there are two main branches that we need to implement not only to initialize the SD card in SPI mode but also to detect what type of card this is. The end results could be:

- Unusable Card: do to non-compatible voltage range, check patter error
- Not SD Memory Card: could be an MMC
- Version 1.x cards: Standard Capacity SD Memory Card (SDSC)
- Version 2 or later: Standard Capacity SD Memory Card (SDSC)
- Version 2 or later: High Capacity (SDHC) or Extended Capacity (SDXC) Memory Card

4.4 CMD0

The first step is to send CMD0. Below is the description of CMD0 from the physical spec.

CMD INDEX	SPI Mode	Argument	Resp	Abbreviation	Command Description
CMD0	Yes	[31:0] stuff bits	R1	GO_IDLE_STATE	Resets the SD Memory Card

This command is used to reset the memory card. The argument is 'stuff bits', which means it will be ignored by the SD card so we can send whatever, and the response is of type R1 (more on that later).

We specify CMD0 simply by setting 'command index' to 0 in our command, and since the argument can be whatever we will set it to 0x00000000. The CRC that corresponds to these bits are provided by the physical spec on page 43 as a 7-bit which is 10010100b. To pass this to our function, we need an 8-bit value. We can put a 0 (0x94) or a 1 at the end but it does not matter as our function will always set the final bit to 1.

```
/* Command List */
// CMD0 - GO_IDLE_STATE
#define CMD0                0
#define CMD0_ARG            0x00000000
#define CMD0_CRC            0x94
```

Before sending CMD0 we need to be able to read the response called R1.

4.5 RESPONSE R1

The R1 format is shown below from the physical spec.

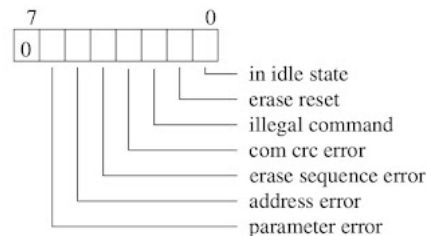


Figure 7-9 : R1 Response Format

R1 is a single byte, where the MSB is always 0, and every other bit represents an error condition. Here is the function for reading the R1:

```
/*  
    Wait for a response from the card other than 0xFF which is
```

the normal state of the MISO line. Timeout occurs after 16 bytes.

return response code

```
*/
static uint8_t sd_read_response1(void){
    uint8_t i = 0, response = 0xFF;

    // Keep polling until actual data received
    while((response = SPI_ReceiveByte()) == 0xFF){
        i++;

        // If no data received for 16 bytes, break
        if(i > 16) break;
    }

    return response;
}
```

The function waits for the MISO line to change it's state. The idle state of the MISO line is high which means 0xFF. If after reading 16 bytes, the state doesn't change then it exits with 0xFF response. The timeout is needed because it takes some time for the card to respond. It should take the SD card 8 clock cycles to respond but it doesn't hurt to use 16 just for good measure.

Now the card initialization function looks like this:

```
SD_RETURN_CODES sd_init(void){
    uint8_t SD_Response[5]; // array to hold response
    uint8_t cmdAttempts = 0;

    // SPI Setup
    SPI_Init();

    // Give SD card time to power up
    _delay_ms(1);

    // Send 80 clock cycles to synchronize
    // The card must be deselected during this time
    sd_deassert_cs();
    for(uint8_t i = 0; i < 10; i++) SPI_SendByte(0xFF);

    // Send CMD0 (GO_IDLE_STATE) - R1 response
    // Resets the SD Memory Card
    // Argument is 0x00 for the reset command, pre-calculated checksum
    while(SD_Response[0] != 0x01){
        // Assert chip select
```

```

        sd_assert_cs();

        // Send CMD0
        sd_command(CMD0, CMD0_ARG, CMD0_CRC);

        // Read R1 response
        SD_Response[0] = sd_read_response1();

        // Send some dummy clocks after GO_IDLE_STATE
        // Deassert chip select
        sd_deassert_cs();

        cmdAttempts++;

        if(cmdAttempts > 10){
            SPI_CARD_CS_DISABLE();
            return SD_IDLE_STATE_TIMEOUT;
        }

    }

    return SD_OK;
}

```

Resetting the card could take some time therefore the CMD0 is transmitted up to 10 times or until the response is 0x01 which means idle state (bit 0 in R1 set to 1). After the timeout the chip is deselected and the function returns `SD_IDLE_STATE_TIMEOUT`.

4.6 CMD8

Next command in the initialization flow is CMD8, SEND_IF_COND (send interface condition), R7 response.

One use of this command is to check whether the card is first generation or version 2.00 (or later). If the card is of first generation, it will respond with R1 with bit 2 set (illegal command). This necessitates going down the left path in the initialization diagram. If the card does not indicate an illegal command we will go through the Version 2.00 path. I couldn't find a version 1.x card to test but some code will be provided regardless.

Another use is to send the host supply voltage information and asks the accessed card whether it can operate in supplied voltage range. Reserved bits shall be set to '0'.

Bit position	47	46	[45:40]	[39:20]	[19:16]	[15:8]	[7:1]	0
Width (bits)	1	1	6	20	4	8	7	1
Value	'0'	'1'	'001000'	'00000h'	x	x	x	'1'
Description	Start bit	Transmission bit	Command index	Reserved bits	Voltage supplied (VHS)	Check pattern	CRC7	End bit

Voltage Supplied	Value Definition
0000b	Not Defined
0001b	2.7-3.6V
0010b	Reserved for Low Voltage Range
0100b	Reserved
1000b	Reserved
Others	Not Defined

Table 4-18 : Format of CMD8

In the above table the command argument (bits 39:8) is expanded and shows what should be. **Reserved bits** must be 0. **Voltage supplied** (VHS) will be set to 0001b since the 3.3V supply voltage is the most common. The **check pattern** is an arbitrary string of bits that can be set to any value, which the SD card will return in the response to ensure the command was processed correctly. Per page 40 of the version 2.00 physical spec, the recommended pattern is '10101010b'. CMD8 is the only command other than CMD0 that requires a correct CRC. Using the recommended check pattern and 3.3V the correct CRC is 1000011b. Command 8 is defined in hexadecimal values as:

```
// CMD8 - SEND_IF_COND (Send Interface Condition)
#define CMD8 8
#define CMD8_ARG 0x0000001AA
#define CMD8_CRC 0x86 // (1000011 << 1)
```

4.7 RESPONSE R7

Command 8 introduced another type of response named R7 and the format can be seen below.

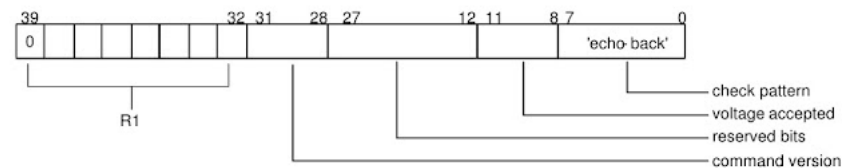


Figure 7-12 : R7 Response Format

R7 is 5 bytes long, and the first byte is identical to R1. The "echo-back" is the check pattern we sent in the command and now it is returned back. First generation cards will only return R1 with the illegal bit command set. To read R7 let's make another function.

```
/*
Read Response 3 or 7 which is 5 bytes long
```

```

        res                                     Pointer to a 5 bytes array
    _____ */
static void sd_read_response3_7(uint8_t *res){
    res[0] = sd_read_response1(); // read response 1 in R3 or R7

    if(res[0] > 1) return; // if error reading R1, return

    res[1] = SPI_ReceiveByte();
    res[2] = SPI_ReceiveByte();
    res[3] = SPI_ReceiveByte();
    res[4] = SPI_ReceiveByte();
}

```

Skipping a bit ahead I have named the function to indicate it can read Response 3 and R7 as well since they are the same. Now we can see why *SD_Response[5]* in the initialization function is 5 bytes long. If the card is of first generation or an error occurs it will return only R1 so if res[0] does not equal 0x01 or 0x00 then we return.

Now CMD8 can be added to our initialization function.

```

sd_assert_cs();
sd_command(CMD8, CMD8_ARG, CMD8_CRC);
sd_read_response3_7(SD_Response);
sd_deassert_cs();

// Enable maximum SPI speed
SPCR0 &= ~(1 << SPR01); // set clock rate fosc/2

// Select initialization sequence path
if(SD_Response[0] == 0x01){
    // The card is Version 2.00 (or later) or SD memory card
    // Check voltage range
    if(SD_Response[3] != 0x01){
        return SD_NONCOMPATIBLE_VOLTAGE_RANGE;
    }

    // Check echo pattern
    if(SD_Response[4] != 0xAA){
        return SD_CHECK_PATTERN_MISMATCH;
    }
}
else if(SD_Response[0] == 0x05){
    // Response code 0x05 = Idle State + Illegal Command indicates
    // the card is of first generation. SD or MMC card.
}
else{

```

```

    }
    return SD_GENERAL_ERROR;
}

```

The initialization sequence path is selected based on the R7 response. Also the SPI speed is set to maximum by setting the prescaler to 2 instead of 32 as in the SPI init function. Ideally you would query the card for maximum speed capability by sending the appropriate command but that takes extra code and space on the MCU. If the card is version 2 or greater, the supported voltage range and echo pattern returned by the card are checked if they equal with the ones transmitted by us.

4.9 CMD58

Moving to the next step, CMD58 – reads OCR (operation conditions register). The response to this command is R3. Here the CRC does not matter unless the CRC option is activated using the appropriate command.

CMD INDEX	SPI Mode	Argument	Resp	Abbreviation	Command Description
CMD58	Yes	[31:0] stuff bits	R3	READ_OCR	Reads the OCR register of a card. CCS bit is assigned to OCR[30].

```

// CMD58 – read OCR (Operation Conditions Register)
#define CMD58 58
#define CMD58_ARG 0x00000000
#define CMD58_CRC 0x00

```

4.10 RESPONSE R3

The response format R3 is shown below.



Figure 7-11 : R3 Response Format

Just like with response 7, first byte is the same as R1 and the following 4 bytes are the contents of OCR, which is defined as below.

OCR bit position	OCR Fields Definition
0-3	reserved
4	reserved
5	reserved
6	reserved
7	Reserved for Low Voltage Range
8	reserved
9	reserved
10	reserved
11	reserved
12	reserved
13	reserved
14	reserved
15	2.7-2.8
16	2.8-2.9
17	2.9-3.0
18	3.0-3.1
19	3.1-3.2
20	3.2-3.3
21	3.3-3.4
22	3.4-3.5
23	3.5-3.6
24 ³	Switching to 1.8V Accepted (S18A)
25-28	reserved
29	UHS-II Card Status
30	Card Capacity Status (CCS) ¹
31	Card power up status bit (busy) ²

VDD Voltage Window

- 1) This bit is valid only when the card power up status bit is set.
- 2) This bit is set to LOW if the card has not finished the power up routine.
- 3) Only UHS-I card supports this bit.

Table 5-1 : OCR Register Definition

Bits 15-24 indicates the voltage ranges supported by this card. Since all cards should support 3.3V, there is no need to check it here. This fields are important only if you need to use the 1.8V supply voltage in which case the bit 24 should be checked but that would be valid only if the card is of UHS type and that can be determined by checking the bit 29.

Bit 31 is the card power up status, which is low if the card has not finished the power up routine. And bit 30, the card capacity status, which is only valid if the power up status is set (bit 31). Bit 30 indicates that the card is a high capacity SD card (SDHC) or extended capacity SD card (SCXC) and not a standard capacity SD card (SDSC).

At this stage this command can only be used to check the supported voltage ranges since the bit 31 will only be set after the next command (ACMD41) and after this command CMD58 will be used again like it is shown in the initialization chart. The response is not used here but it must be read in order for the clock to be generated and the card to clear the response flags.

We already have a function that can read response R3 so the next code is added to the init function:

```
// CMD58 — read OCR (Operation Conditions Register) - R3 response
// Reads the OCR register of a card
sd_assert_cs();
sd_command(CMD58, CMD58_ARG, CMD58_CRC);
```

```
sd_read_response3_7(SD_Response);
sd_deassert_cs();
```

4.11 ACMD41 & CMD55

ACMD41 – SD_SEND_OP_COND (send operating condition) is used to start the card's initialization process and is the next command in the initialization flow diagram. The 'A' before the command name signifies that it is an application specific command. Before this type of commands we need to inform the card controller that this is an application specific command rather than a standard command by first sending CMD55 – APP_CMD.

CMD INDEX	SPI Mode	Argument	Resp	Abbreviation	Command Description
CMD55	Yes	[31:0] stuff bits	R1	APP_CMD	Defines to the card that the next command is an application specific command rather than a standard command

CMD INDEX	SPI Mode	Argument	Resp	Abbreviation	Command Description
ACMD41	Yes	[31]Reserved bit [30]HCS [29:0]Reserved bits	R1	SD_SEND_OP_COND	Sends host capacity support information and activates the card's initialization process. Reserved bits shall be set to '0'

ACMD41 has the same format as CMD41. If CMD55 is not sent first, then sending a command with 41 in the first byte will be interpreted as CMD41, which is a reserved command and will result in an error response.

All we can set in ACMD41 argument is bit 30 (HCS) which indicates that we support high capacity cards and set reserved bits to '0'.

```
// CMD55 – APP_CMD
#define CMD55                55
#define CMD55_ARG            0x00000000
#define CMD55_CRC            0x00

// ACMD41 – SD_SEND_OP_COND (Send Operating Condition)
// ACMD stands for Application Specific Command and before issued,
// the CMD55 must be sent first
#define ACMD41                41
#define ACMD41_ARG            0x40000000
#define ACMD41_CRC            0x00
```

In the power-up diagram it is shown that the initialization process can take up to 1 second so we need another function that sends CMD55 followed by ACMD41 until the card returns 'in_idle_state' = 0 or the timeout condition occurs. The loop is set to run up to 100 times and each loop has a 10ms delay resulting in a 1000ms timeout.

```

static uint8_t SD_CardType;

/*
    Send ACMD41 – Sends host capacity support information
    and activates the card's initialization process.
    Reserved bits shall be set to '0'.
*/
static SD_RETURN_CODES sd_command_ACMD41(void){
    uint8_t response;
    uint8_t i = 100;

    // Initialization process can take up to 1 second so we add a 10ms delay
    // and a maximum of 100 iterations

    do{
        // CMD55 – APP_CMD – R1 response
        sd_assert_cs();
        sd_command(CMD55, CMD55_ARG, CMD55_CRC);
        sd_read_response1();
        sd_deassert_cs();

        // ACMD41 – SD_SEND_OP_COND (Send Operating Condition) – R1 response
        sd_assert_cs();

        if(SD_CardType == SD_V1_SDSC)
            sd_command(ACMD41, 0, ACMD41_CRC);
        else
            sd_command(ACMD41, ACMD41_ARG, ACMD41_CRC);

        response = sd_read_response1();
        sd_deassert_cs();

        i--;
        _delay_ms(10);
    }while((response != 0) && (i > 0));

    // Timeout
    if(i == 0) return SD_IDLE_STATE_TIMEOUT;

    return response;
}

```

The variable `SD_CardType` holds the card type and the values can be:

```
// Card Type
#define SD_V1_SDSC          1
#define SD_V2_SDSC          2
#define SD_V2_SDHC_SDXC     3
```

If the card is SDSC version 1 then bit 30 (HCS) must be 0 and `SD_CardType` is used for that. In the initialization function we use this new function like this:

```
if(sd_command_ACMD41() > 0) return SD_IDLE_STATE_TIMEOUT;
```

That means if the function doesn't return 0 it waited for 1 second and the card was still in idle mode or an error occurred. The response R1 could be checked to see which bit is set by using bit-wise AND like this:

```
// R1 Response
#define PARAM_ERROR(X)      X & 0b01000000
#define ADDR_ERROR(X)       X & 0b00100000
#define ERASE_SEQ_ERROR(X)  X & 0b00010000
#define CRC_ERROR(X)        X & 0b00001000
#define ILLEGAL_CMD(X)      X & 0b00000100
#define ERASE_RESET(X)      X & 0b00000010
#define IN_IDLE(X)          X & 0b00000001
```

The last thing to do is send CMD58 again and check power up status bit (OCR bit 31) and Card Capacity Status (bit 30) that indicates if this is an SDSC or SDHC, SDXC card.

```
// CMD58 - read OCR (Operation Conditions Register) - R3 response
sd_assert_cs();
sd_command(CMD58, CMD58_ARG, CMD58_CRC);
sd_read_response3_7(SD_Response);
sd_deassert_cs();

// Check if the card is ready
// Read bit OCR 31 in R3
if(!(SD_Response[1] & 0x80)){
    return SD_POWER_UP_BIT_NOT_SET;
}

// Read CCS bit OCR 30 in R3
if(SD_Response[1] & 0x40){
    // SDXC or SDHC card
    SD_CardType = SD_V2_SDHC_SDXC;
}else{
    // SDSC
    SD_CardType = SD_V2_SDSC;
}
```


Now the complete initialization function looks like this:

```
SD_RETURN_CODES sd_init(void){
    uint8_t SD_Response[5]; // array to hold response
    uint8_t cmdAttempts = 0;

    // SPI Setup
    SPI_Init();

    // Give SD card time to power up
    _delay_ms(1);

    // Send 80 clock cycles to synchronize
    // The card must be deselected during this time
    sd_deassert_cs();
    for(uint8_t i = 0; i < 10; i++) SPI_SendByte(0xFF);

    // Send CMD0 (GO_IDLE_STATE) - R1 response
    // Resets the SD Memory Card
    // Argument is 0x00 for the reset command, precalculated checksum
    while(SD_Response[0] != 0x01){
        // Assert chip select
        sd_assert_cs();

        // Send CMD0
        sd_command(CMD0, CMD0_ARG, CMD0_CRC);

        // Read R1 response
        SD_Response[0] = sd_read_response1();

        // Deassert chip select
        sd_deassert_cs();

        cmdAttempts++;

        if(cmdAttempts > 10){
            SPI_CARD_CS_DISABLE();
            return SD_IDLE_STATE_TIMEOUT;
        }
    }

    // Send CMD8 - SEND_IF_COND (Send Interface Condition) - R7 response (or F
    // Sends SD Memory Card interface condition that includes host supply volt
    // accessed card whether card can operate in supplied voltage range.
```

```

// Check whether the card is first generation or Version 2.00 (or later).
// If the card is of first generation, it will respond with R1 with bit 2
// otherwise the response will be 5 bytes long R7 type
// Voltage Supplied (VHS) argument is set to 3.3V (0b0001)
// Check Pattern argument is the recommended pattern '0b10101010'
// CRC is 0b1000011 and is precalculated
sd_assert_cs();
sd_command(CMD8, CMD8_ARG, CMD8_CRC);
sd_read_response3_7(SD_Response);
sd_deassert_cs();

// Enable maximum SPI speed
SPCR0 &= ~(1 << SPR01); // set clock rate fosc/2

// Select initialization sequence path
if(SD_Response[0] == 0x01){
    // The card is Version 2.00 (or later) or SD memory card
    // Check voltage range
    if(SD_Response[3] != 0x01){
        return SD_NONCOMPATIBLE_VOLTAGE_RANGE;
    }

    // Check echo pattern
    if(SD_Response[4] != 0xAA){
        return SD_CHECK_PATTERN_MISMATCH;
    }

    // CMD58 - read OCR (Operation Conditions Register) - R3 response
    sd_assert_cs();
    sd_command(CMD58, CMD58_ARG, CMD58_CRC);
    sd_read_response3_7(SD_Response);
    sd_deassert_cs();

    // ACMD41 - starts the initialization process - R1 response
    // Continue to send ACMD41 (always preceded by CMD55) until the ca
    // with 'in_idle_state', which is R1 = 0x00.
    if(sd_command_ACMD41() > 0) return SD_IDLE_STATE_TIMEOUT;

    // CMD58 - read OCR (Operation Conditions Register) - R3 response
    sd_assert_cs();
    sd_command(CMD58, CMD58_ARG, CMD58_CRC);
    sd_read_response3_7(SD_Response);
    sd_deassert_cs();

    // Check if the card is ready

```

```

        // Read bit OCR 31 in R3
        if(!(SD_Response[1] & 0x80)){
            return SD_POWER_UP_BIT_NOT_SET;
        }

        // Read CCS bit OCR 30 in R3
        if(SD_Response[1] & 0x40){
            // SDXC or SDHC card
            SD_CardType = SD_V2_SDHC_SDXC;
        }else{
            // SDSC
            SD_CardType = SD_V2_SDSC;
        }

    }else if(SD_Response[0] == 0x05){
        // Response code 0x05 = Idle State + Illegal Command indicates
        // the card is of first generation. SD or MMC card.
        SD_CardType = SD_V1_SDSC;

        // ACMD41
        SD_Response[0] = sd_command_ACMD41();
        if(ILLEGAL_CMD(SD_Response[0])) return SD_NOT_SD_CARD;
        if(SD_Response[0]) return SD_IDLE_STATE_TIMEOUT;

    }else{
        return SD_GENERAL_ERROR;
    }

    // Read and write operations are performed on SD cards in blocks of set length
    // Block length can be set in Standard Capacity SD cards using CMD16 (SET_BLOCKLEN)
    // For SDHC and SDXC cards the block length is always set to 512 bytes.

    return SD_OK;
}

```

In the [next part](#) we will see how to read and write the SD card. Hopefully I will see you there.

You can also download this tutorial in a PDF format at the link below. If you find it useful, consider a small donation at the link provided.

[SD card tutorial - Interfacing an SD card with a microcontroller over SPI, v1.0, 2022.pdf](#)

Donations link: <https://paypal.me/alientransducer>

Share

Posted by [Liviú Istrate](#) at [July 13, 2022](#)

Labels: [Memory Card](#), [SD Card](#)

No comments:

Post a Comment



Enter Comment

[Newer Post](#)

[Home](#)

[Older Post](#)

Liviú Istrate © 2017. Theme images by merrymoonmary. Powered by Blogger.

Do Not Sell or Share My Personal Information