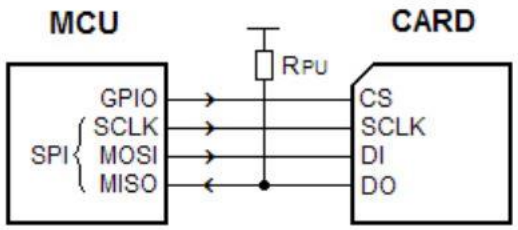
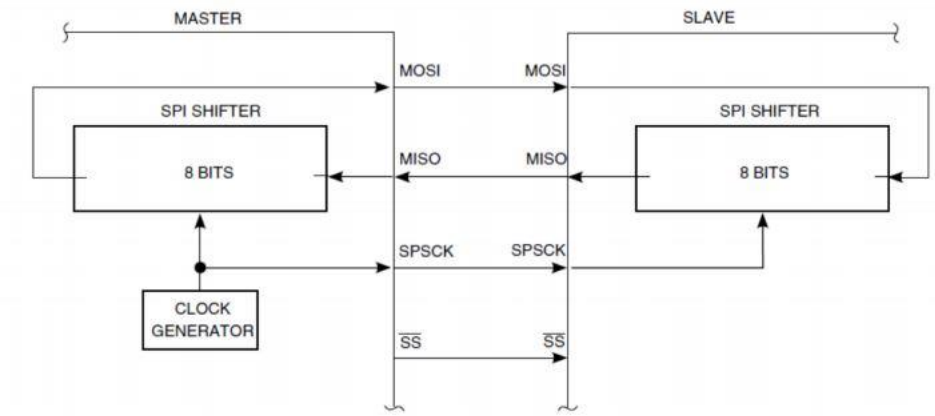
Embedded Systems(EE30004)
Final TestQ ES

Submitted By:

Pratyush Jaiswal
18EE30021

Interface a microSD card to Atmega328. Read a single block, complement and write back. Show the step by step algorithm and code in assembly language. Please explain the codes by proper commenting. Simulate in proteus environment for the read command while showing each of the signals on the scope. Upload your reports by 10AM and demo through your google drive-link also by 10AM

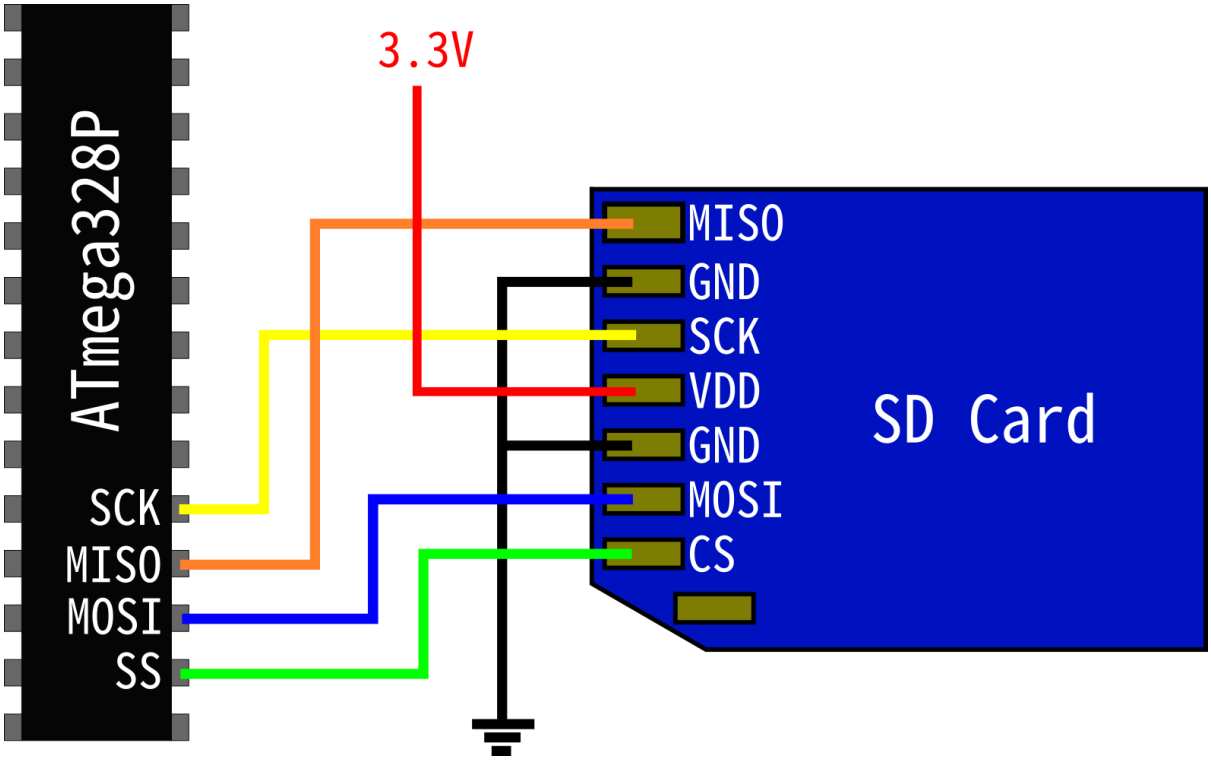




Solution

Connections

A simplified schematic is shown below for the connections between an ATmega328P and an SD card. Note that the SD card requires 3.3V power, do not connect to 5V! Also, I am using PINB2 as the Chip Select.



Initializing SPI

In order to setup SPI, we need to setup a few things, namely the clock polarity, phase, and rate. Unfortunately, bus timing diagrams are not available in the simplified (free) versions of the SD Physical Layer specification. However, in this SanDisk SD Card product manual, a timing diagram is provided for one of their cards, shown below.

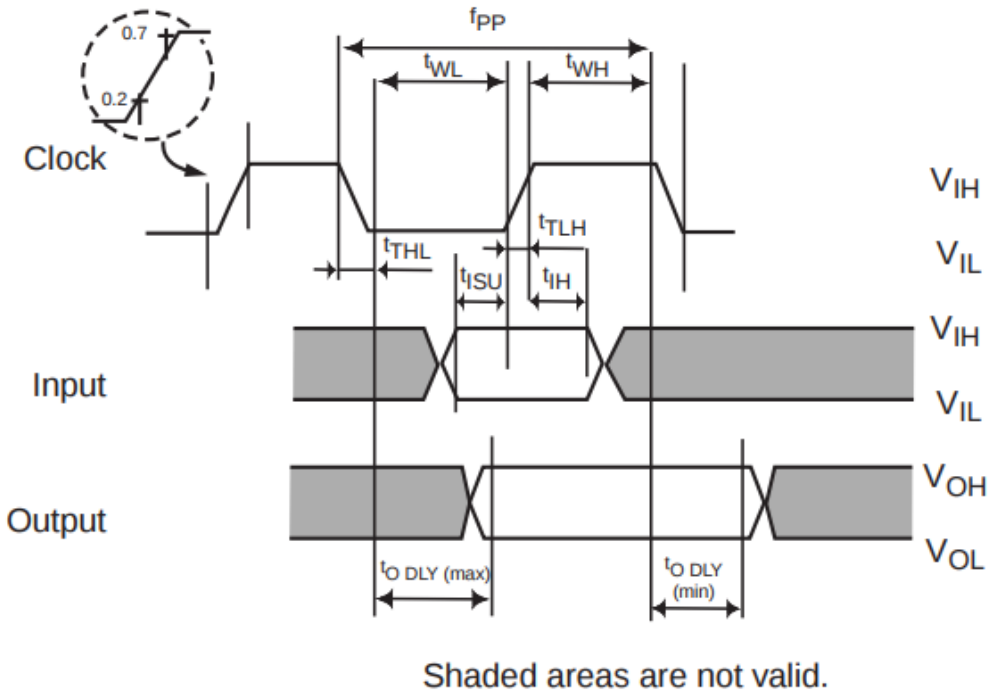


Figure 3-7. Timing Diagram Data Input/Output Referenced to Clock

As can be seen, the clock idles low and the output is sampled on the leading edge, which corresponds to CPOL = 0 and CPHA = 0. We can assume that this mode of operation should work for other SD Cards.

CPOL and CPHA are default set to 00 in the SPCR (SPI Control Register), so we do not need to do anything with them in SPI initialization. Although modern SD Cards can operate at very high speeds, here we will set SCK to the lowest value for debugging purposes, and increase it later when we have our code working.

To abstract the pins used in our code, it will be helpful to give them functional names. Additionally, all SPI commands to the SD card require our 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. Such simple commands do not necessarily warrant a function, but some simple preprocessor macros will be very helpful.

Here is our SPI initialization code, as well as our send/receive function, and some useful pin definitions and macros.

```
// pin definitions
#define DDR_SPI          DDRB
#define PORT_SPI         PORTB
#define CS               PINB2
#define MOSI             PINB3
#define MISO             PINB4
#define SCK              PINB5
```

```
// macros
#define CS_ENABLE()      PORT_SPI &= ~(1 << CS)
#define CS_DISABLE()     PORT_SPI |= (1 << CS)

void SPI_init()
{
    // set CS, MOSI and SCK to output
    DDR_SPI |= (1 << CS) | (1 << MOSI) | (1 << SCK);

    // enable pull up resistor in MISO
    DDR_SPI |= (1 << MISO);

    // enable SPI, set as master, and clock to fosc/128
    SPCR = (1 << SPE) | (1 << MSTR) | (1 << SPR1) | (1 << SPR0);
}

uint8_t SPI_transfer(uint8_t data)
{
    // load data into register
    SPDR = data;

    // Wait for transmission complete
    while(!(SPSR & (1 << SPIF)));

    // return SPDR
    return SPDR;
}
```

Power Up Sequence

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

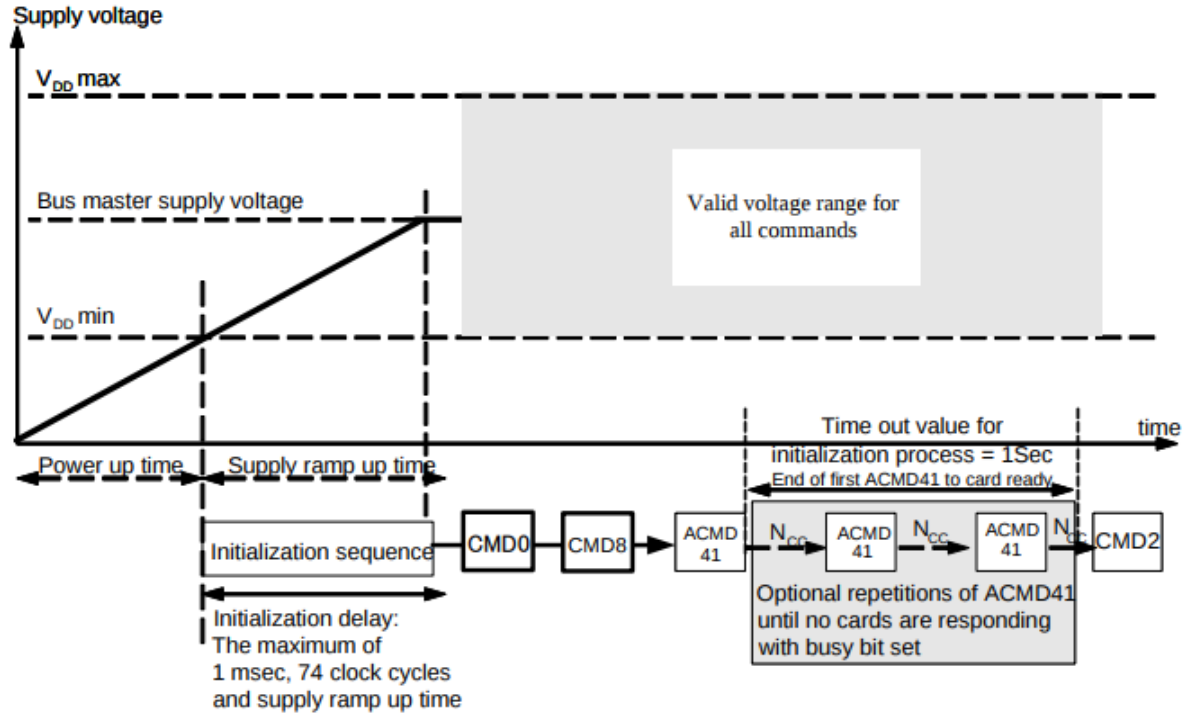


Figure 6-1: Power-up Diagram

Note that we need to provide at least 1 msec delay and 74 clock cycles before sending commands to an 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.

```
#include<util/delay.h>

void SD_powerUpSeq()
{
    // make sure card is deselected
    CS_DISABLE();

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

    // send 80 clock cycles to synchronize
    for(uint8_t i = 0; i < 10; i++)
        SPI_transfer(0xFF);

    // deselect SD card
    CS_DISABLE();
    SPI_transfer(0xFF);
}
```

Sending Commands

Below is the command format for SD card commands. All commands are 6 bytes long and contain a command index, arguments, and CRC. The command index field is used to tell the SD card which command we are sending. For example, if CMD0 is required, then the 6 bits in command index should be set to 000000b. The argument field is used in some commands and is ignored by the SD card in others. Whenever no arguments are needed, we will fill this field with all zeros. Finally, the Cyclic Redundancy Check (CRC) is used to ensure the contents of the command have been correctly received by the SD card. As we will see, in SPI mode, only a few commands actually require a correct CRC. In the cases where it is not needed, we will set it to all zeros.

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

Here we will write a function to send commands to an SD card. We will pass an 8-bit command index, a 32-bit argument, and an 8-bit CRC.

```
void SD_command(uint8_t cmd, uint32_t arg, uint8_t crc)
{
    // transmit command to sd card
```

```
    SPI_transfer(cmd|0x40);

    // transmit argument
    SPI_transfer((uint8_t)(arg >> 24));
    SPI_transfer((uint8_t)(arg >> 16));
    SPI_transfer((uint8_t)(arg >> 8));
    SPI_transfer((uint8_t)(arg));

    // transmit crc
    SPI_transfer(crc|0x01);
}
```

We begin by transmitting the command index. Note however in the command format table, that the command index is only 6 bits long. The 2 most significant bits of the command always set to 01b. If we always pass command indices less than 128 (which we will see is the case), then bit 48 of our 'cmd' argument will always be zero. However, in order to set bit 47 to 1, we need to bitwise OR our input 'cmd' with 0x40.

```
    // transmit command to sd card
    SPI_transfer(cmd|0x40);
```

Next, we transmit our 4 byte argument, one byte at a time, by shifting it down 8 bits at a time.

```
    // transmit argument
    SPI_transfer((uint8_t)(arg >> 24));
    SPI_transfer((uint8_t)(arg >> 16));
    SPI_transfer((uint8_t)(arg >> 8));
    SPI_transfer((uint8_t)(arg));
```

Finally, we transmit the CRC. Note that the CRC is only 7 bits long, and the final bit of any command is always set to 1. To make our lives easier, we will bitwise OR the 'crc' argument with 0x01 so the final bit is always 1 and we don't have to worry about it when supplying arguments to the function.

```
    // transmit crc
    SPI_transfer(crc|0x01);
```

Initialization Flow

The SPI mode initialization flow for an SD card is shown in the diagram below

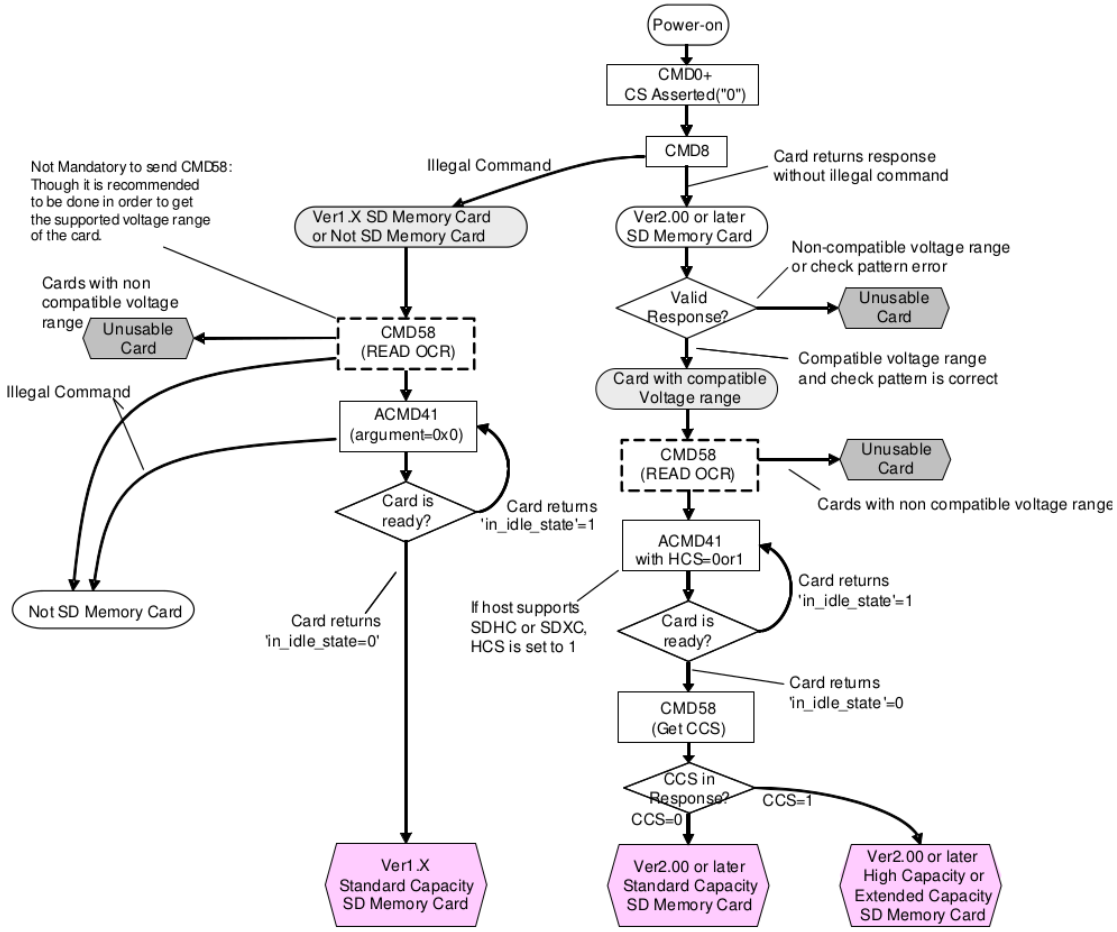


Figure 7-2 : SPI Mode Initialization Flow

The first step in the process 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 a software reset for the SD card. Its argument is 'stuff bits', which means it will be ignored by the SD card, and the response is of type R1 (more on that later).

SPI mode is a secondary mode of communication for SD cards - they power up in an "SD Bus protocol mode." Cards only switch to SPI when the Chip Select line is driven low and CMD0 is sent. In SPI mode CRCs are ignored by default, but in SD Bus mode, which we are coming from at startup, they are required. So for the first command we must have a correct checksum.

We specify CMD0 simply by setting 'command index' to 0 in our command, and since the argument is just stuff, we will set it to 0x00000000. Now we need the CRC that corresponds to these bits. Luckily, the physical spec provides us this 7 bit value on page 43 - 10010100b. To pass this to our function, we need an 8-bit value. We can put a 0 on the end (0x94) or a 1 (0x95) but it does not matter as our function will always the final bit will always be set to 1 before we transmit.

```
#define CMD0        0
#define CMD0_ARG    0x00000000
#define CMD0_CRC    0x94

// send CMD0
SD_command(CMD0, CMD0_ARG, CMD0_CRC);
```

Note that CMD0 returns a response of format R1. The R1 format is shown below from the physical spec.

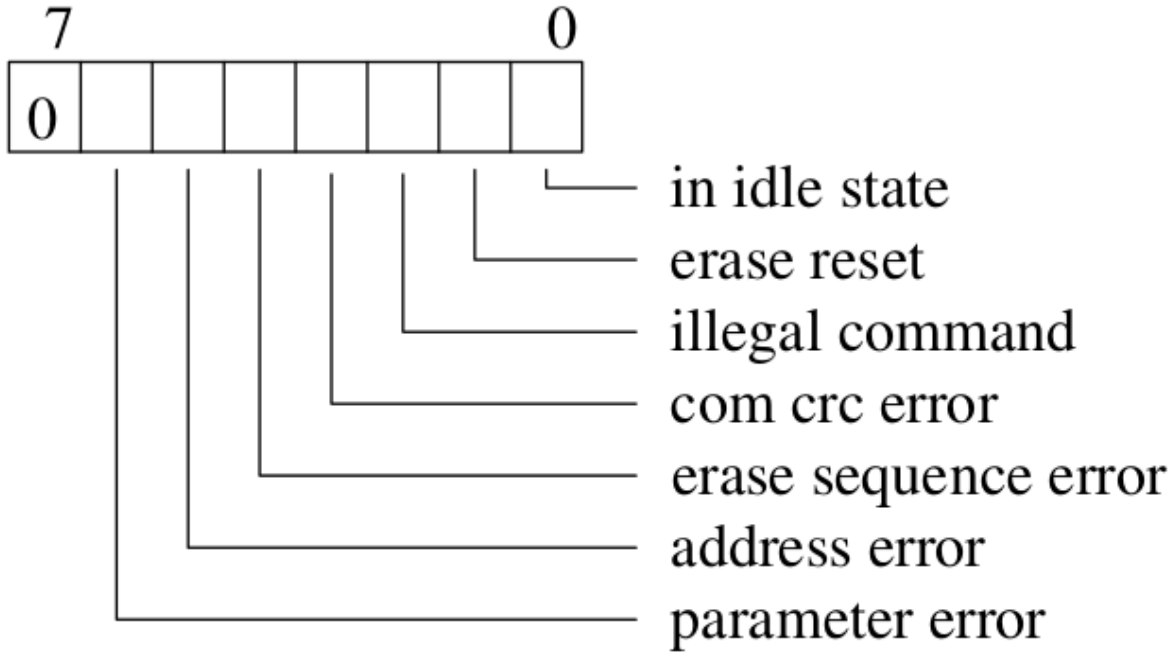


Figure 7-9 : R1 Response Format

Note that R1 is a single byte, where the first bit is always 0, and every other bit represents an error condition.

Knowing this, we can write a function to retrieve a single byte response after sending CMD0. Here it is below

```
uint8_t SD_readRes1()
{
    uint8_t i = 0, res1;
```

```
    // keep polling until actual data received
    while((res1 = SPI_transfer(0xFF)) == 0xFF)
    {
        i++;

        // if no data received for 8 bytes, break
        if(i > 8) break;
    }

    return res1;
}
```

Since the MISO line is high by default, if the SD card does not respond, we will simply read 0xFF. The SD card may require some time after our command is sent to process it, so we will keep polling until we receive data. However, the card should respond within 8 cycles. If it has not responded by then we break and return the response (which will be 0xFF).

Putting all of this together, the full sequence for sending CMD0, aka GO_IDLE_STATE, is shown below

```
uint8_t SD_goIdleState()
{
    // assert chip select
    SPI_transfer(0xFF);
    CS_ENABLE();
    SPI_transfer(0xFF);

    // send CMD0
    SD_command(CMD0, CMD0_ARG, CMD0_CRC);

    // read response
    uint8_t res1 = SD_readRes1();

    // deassert chip select
    SPI_transfer(0xFF);
    CS_DISABLE();
    SPI_transfer(0xFF);

    return res1;
}
```

Here, we provide 8 clocks before and after pulling CS low, and another 8 before sending the command. This is recommended to ensure the card recognizes the change in CS. In addition, we provide and extra byte before and after transitioning CS high. These extra bytes are not always necessary but many people have had issues when multiple cards are on the bus.

Putting It All Together

Now it is time to put all of this into a program. Below is a main function using the functions defined previously in this report. Here we simply go through the power up sequence and command the card to idle.

```
int main(void)
{
    // initialize SPI
    SPI_init();

    // start power up sequence
    SD_powerUpSeq();

    // command card to idle
    SD_goIdleState();

    while(1);
}
```

Block Length

Read and write operations are performed on SD cards in blocks of set lengths. Block length can be set in Standard Capacity SD cards using CMD16 (SET_BLOCKLEN). Here, we will only consider SDHC and SDXC cards, where the block length is always set to 512 bytes.

Single Block Read

The command for a single block read is CMD17 - READ_SINGLE_BLOCK. Its format is shown in the table below.

CMD INDEX	SPI Mode	Argument	Resp	Abbreviation	Command Description
CMD17	Yes	[31:0] data address[10]	R1	READ_SINGLE_BLOCK	Reads a block of the size selected by the SET_BLOCKLEN command.[3]

Note that we need to provide the address we want to read from in the argument. SDHC and SDXC are block addressed, meaning an address of 0 will read back bytes 0-511, address 1 will read back 512-1023, etc. With 32 bits to specify a 512 byte block, the maximum size a card can support is (2^32)*512 = 2048 megabytes or 2 terabytes.
The flow of a single block read is shown in the diagram below:

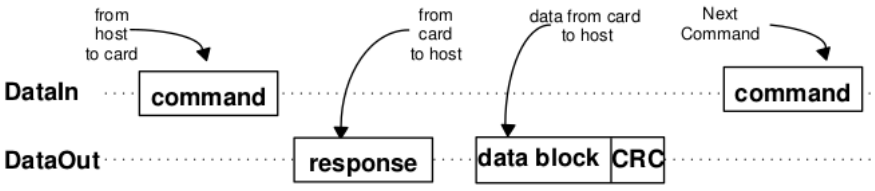


Figure 7-3 : Single Block Read Operation

When CMD17 is sent, the card will respond with R1, followed by a data block suffixed with a CRC. The card may take some time between sending its R1 response and its

first data byte, so how do we know when the data is actually starting? Single data blocks are always preceded by the bits 0b11111110 (0xFE), which is referred to as a start token.

For Single Block Read, Single Block Write and Multiple Block Read:

- First byte: Start Block

7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0

- Bytes 2-513 (depends on the data block length): User data
- Last two bytes: 16 bit CRC.

Immediately after sending the start token the card will send out a data block length of data (which is always 512 bytes for SDHC and SDXC cards). This is then followed by a 16-bit CRC.

Writing a Read Block Function

Here is a function that allows us to read a single block from an SD card.

```
#define CMD17                   17
#define CMD17_CRC               0x00
#define SD_MAX_READ_ATTEMPTS    1563

/*****************************************************************************
*****
 Read single 512 byte block
 token = 0xFE - Successful read
 token = 0x0X - Data error
 token = 0xFF - Timeout
*****************************************************************************
****/
uint8_t SD_readSingleBlock(uint32_t addr, uint8_t *buf, uint8_t *token)
{
    uint8_t res1, read;
    uint16_t readAttempts;

    // set token to none
    *token = 0xFF;

    // assert chip select
    SPI_transfer(0xFF);
    CS_ENABLE();
    SPI_transfer(0xFF);

    // send CMD17
    SD_command(CMD17, addr, CMD17_CRC);
```

```
    // read R1
    res1 = SD_readRes1();

    // if response received from card
    if(res1 != 0xFF)
    {
        // wait for a response token (timeout = 100ms)
        readAttempts = 0;
        while(++readAttempts != SD_MAX_READ_ATTEMPTS)
            if((read = SPI_transfer(0xFF)) != 0xFF) break;

        // if response token is 0xFE
        if(read == 0xFE)
        {
            // read 512 byte block
            for(uint16_t i = 0; i < 512; i++) *buf++ = SPI_transfer(0xFF);

            // read 16-bit CRC
            SPI_transfer(0xFF);
            SPI_transfer(0xFF);
        }

        // set token to card response
        *token = read;
    }

    // deassert chip select
    SPI_transfer(0xFF);
    CS_DISABLE();
    SPI_transfer(0xFF);

    return res1;
}
```

Note here that our function takes a 32-bit address as an argument and requires a buffer of at least 512 bytes to store the data in. We also pass a pointer to an 8 bit value that will be used to store the data token.

After we have read R1, if it is not empty (i.e. the card actually responded to the command), we poll the card until we receive a token or we timeout.

```
    // if response received from card
    if(res1 != 0xFF)
    {
        // wait for a response token (timeout = 100ms)
        readAttempts = 0;
        while(++readAttempts != SD_MAX_READ_ATTEMPTS)
            if((read = SPI_transfer(0xFF)) != 0xFF) break;
```

Per section 4.6.2.1 of the physical spec:

> *The host should use **100ms timeout** (minimum) for single and multiple read operations*

In this example, I am using a 16MHz oscillator and have the SPI clock set to divide by 128. Thus, to get the number of bytes we need to send over SPI to reach 100ms, we do (0.1s * 16000000 MHz)/(128 * 8 bytes) = 1563.
If we have received a start token, we simply read 512 bytes into our buffer, followed by the 16-bit CRC (which we will not use for now).

```
        // if response token is 0xFE
        if(read == 0xFE)
        {
            // read 512 byte block
            for(uint16_t i = 0; i < 512; i++) *buf++ = SPI_transfer(0xFF);

            // read 16-bit CRC
            SPI_transfer(0xFF);
            SPI_transfer(0xFF);
        }
```

If we do not receive a start token, we will not try to read a block from the card. If our token is 0xFF, then we never received anything from the card (and thus timed out). Otherwise, we (should) have received an error token (more on that later).

Testing it out

Using the intialization code from the previous, lets try reading the very first block from the card.

```
int main(void)
{
    // array to hold responses
    uint8_t res[5], sdBuf[512], token;

    // initialize UART
    UART_init();

    // initialize SPI
    SPI_init();

    // initialize sd card
    if(SD_init() != SD_SUCCESS)
    {
        UART_pputs("Error initializaing SD CARD\r\n"); while(1);
    }
    else
    {
        UART_pputs("SD Card initialized\r\n");
    }

    // read sector 0
    res[0] = SD_readSingleBlock(0x00000000, sdBuf, &token);

    // print response
```

```
    if(SD_R1_NO_ERROR(res[0]) && (token == 0xFE))
    {
        for(uint16_t i = 0; i < 512; i++) UART_puthex8(sdBuf[i]);
        UART_puts("\r\n");
    }
    else
    {
        UART_pputs("Error reading sector\r\n");
    }

    while(1) ;
}
```

Writing Blocks

Writing, just like reading, is done in blocks of 512 bytes. The command we will be looking at here is CMD24 - WRITE_BLOCK. The format for this command is shown below.

CMD INDEX	SPI Mode	Argument	Resp	Abbreviation	Command Description
CMD24	Yes	[31:0] data address[10]	R1	WRITE_BLOCK	Writes a block of the size selected by the SET_BLOCKLEN command.[4]

The flow for a single block write is shown below.

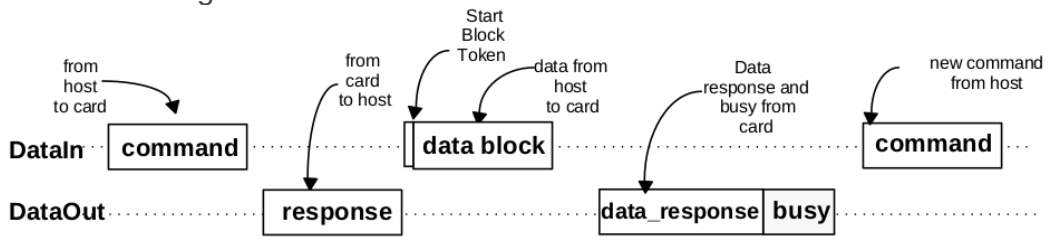
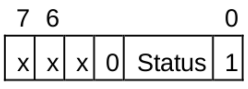


Figure 7-6: Single Block Write Operation

Notice that we send the write block command, wait for a response (R1), then send a start block token (0xFE) followed by 512 bytes of data to be written. After this we will wait for new type of token from the card: a data response token. The format for this is shown below.



The meaning of the status bits is defined as follows:
‘010’ - Data accepted.
‘101’ - Data rejected due to a CRC error.
’110’ - Data Rejected due to a Write Error

If the card accepts the data, we will get the token 0bxxx00101. The card will then send busy tokens (0x00) until it has finished writing the data.

Putting all of this into a function looks something like this:

```
#define CMD24                   24
#define CMD24_ARG               0x00
#define SD_MAX_WRITE_ATTEMPTS   3907

/*************************************************************************
*****
 Write single 512 byte block
 token = 0x00 - busy timeout
 token = 0x05 - data accepted
 token = 0xFF - response timeout
*************************************************************************
****/
uint8_t SD_writeSingleBlock(uint32_t addr, uint8_t *buf, uint8_t *token)
{
    uint8_t readAttempts, read;

    // set token to none
    *token = 0xFF;

    // assert chip select
    SPI_transfer(0xFF);
    CS_ENABLE();
    SPI_transfer(0xFF);

    // send CMD24
    SD_command(CMD24, addr, CMD24_CRC);

    // read response
    res[0] = SD_readRes1();

    // if no error
    if(res[0] == SD_READY)
    {
        // send start token
        SPI_transfer(SD_START_TOKEN);

        // write buffer to card
        for(uint16_t i = 0; i < SD_BLOCK_LEN; i++) SPI_transfer(buf[i]);

        // wait for a response (timeout = 250ms)
        readAttempts = 0;
        while(++readAttempts != SD_MAX_WRITE_ATTEMPTS)
            if((read = SPI_transfer(0xFF)) != 0xFF) { *token = 0xFF; break;
}

        // if data accepted
        if((read & 0x1F) == 0x05)
        {
            // set token to data accepted
            *token = 0x05;

            // wait for write to finish (timeout = 250ms)
            readAttempts = 0;
```

```
            while(SPI_transfer(0xFF) == 0x00)
                if(++readAttempts == SD_MAX_WRITE_ATTEMPTS) { *token = 0x00;
break }
        }
    }

    // deassert chip select
    SPI_transfer(0xFF);
    CS_DISABLE();
    SPI_transfer(0xFF);

    return res[0];
}
```

Note that we send the write command to the card and make sure there are no errors in the R1 response.

```
    // send CMD24
    SD_command(CMD24, addr, CMD24_CRC);

    // read response
    res[0] = SD_readRes1();

    // if no error
    if(res[0] == SD_READY)
    {
        /***/
    }
```

If R1 is error free, we send the start token and then start transmitting the data in the buffer.

```
        // send start token
        SPI_transfer(SD_START_TOKEN);

        // write buffer to card
        for(uint16_t i = 0; i < SD_BLOCK_LEN; i++) SPI_transfer(buf[i]);
```

When we have finished writing the buffer, we wait for the card to send a data reponse token. Remember data accepted tokens are 0bxxx00101, so we mask the upper three bits of the first non 0xFF response we get and see if it is equal to 0b00000101.

```
        // wait for a response (timeout = 250ms)
        readAttempts = 0;
        while(++readAttempts != SD_MAX_WRITE_ATTEMPTS)
            if((read = SPI_transfer(0xFF)) != 0xFF) { *token = 0xFF; break;
}

        // if data accepted
        if((read & 0x1F) == 0x05)
        {
            /***/
        }
```

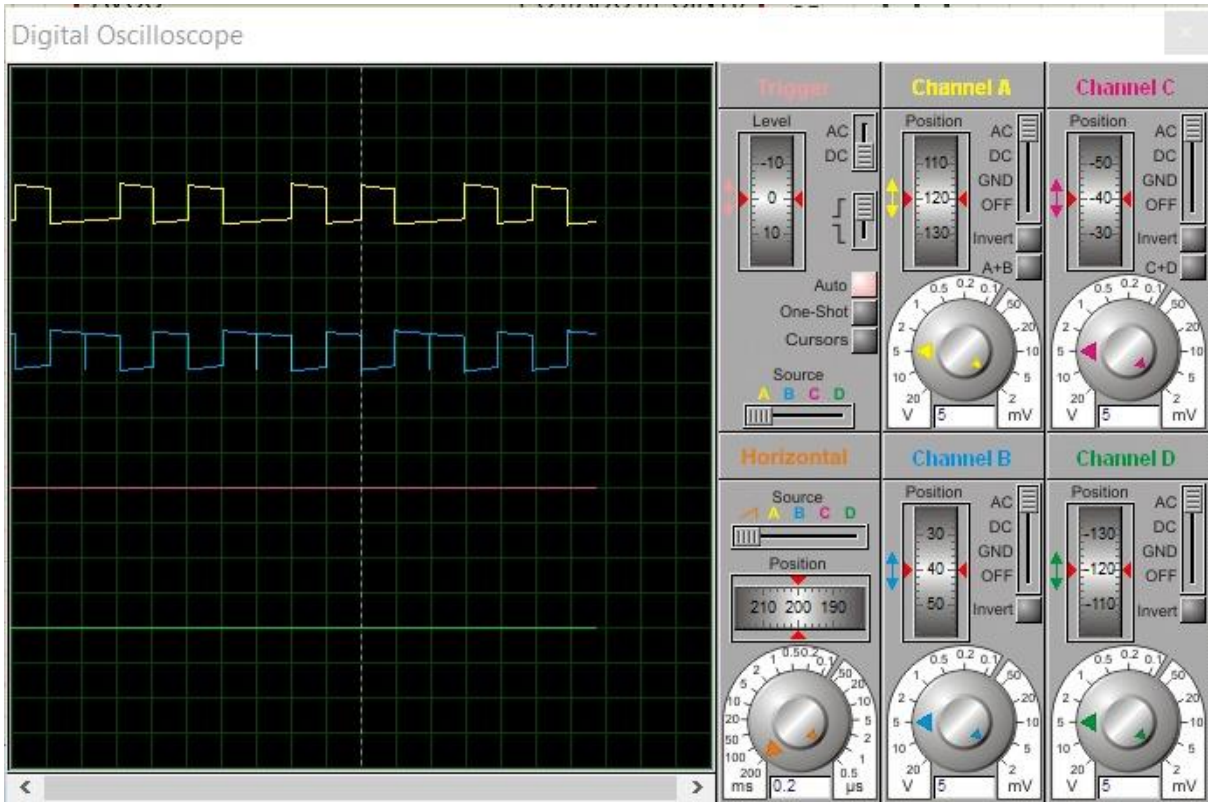
Finally we wait for the card to finish writing the data.

```
            // wait for write to finish (timeout = 250ms)
            readAttempts = 0;
            while(SPI_transfer(0xFF) == 0x00)
                if(++readAttempts == SD_MAX_WRITE_ATTEMPTS) { *token = 0x00;
break }
```

Let's try this out, simply writing a set block of data to a sector on the card. Note: it should go without saying, make sure we backup our card at this point if there is any sensitive data on it - we may break the filesystem or corrupt data by writing to a random sector. Here I will fill the buffer with the value 0x55 and write to address 0x00000100.

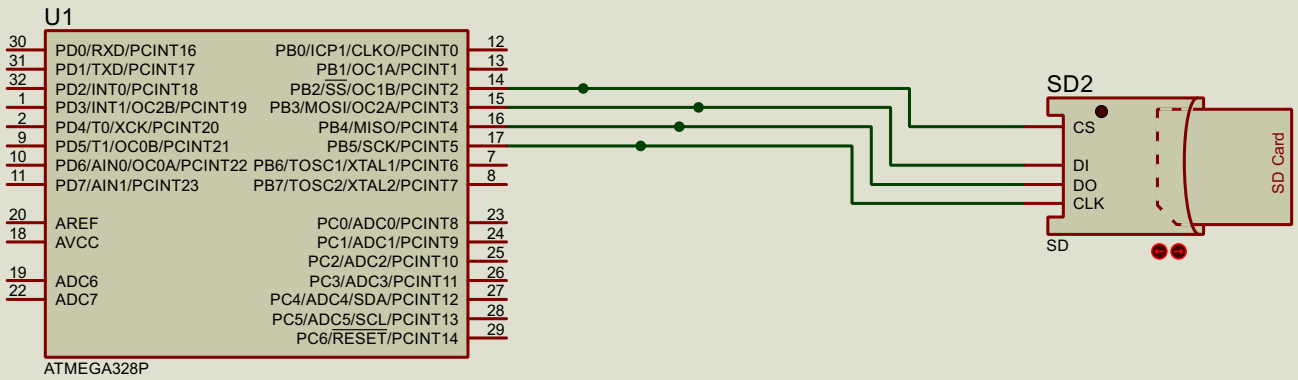
```
    // fill buffer with 0x55
    for(uint16_t i = 0; i < 512; i++) buf[i] = 0x55;

    // write 0x55 to address 0x100 (256)
    res = SD_writeSingleBlock(0x00000100, buf, &token);
```

Proteus Output



Demo Link

https://drive.google.com/file/d/10yVuXuR3ULGax6He5hWw5H9eCKW1YTya/view?usp=sharing


U1
30
31
32
1
2
9
10
11
20
18
19
22
12
13
14
15
16
17
7
8
23
24
25
26
27
28
29
PD0/RXD/PCINT16
PD1/TXD/PCINT17
PD2/INT0/PCINT18
PD3/INT1/OC2B/PCINT19
PD4/T0/XCK/PCINT20
PD5/T1/OC0B/PCINT21
PD6/AIN0/OC0A/PCINT22
PD7/AIN1/PCINT23
AREF
AVCC
ADC6
ADC7
PB0/ICP1/CLKO/PCINT0
PB1/OC1A/PCINT1
PB2/SS/OC1B/PCINT2
PB3/MOSI/OC2A/PCINT3
PB4/MISO/PCINT4
PB5/SCK/PCINT5
PB6/TOSC1/XTAL1/PCINT6
PB7/TOSC2/XTAL2/PCINT7
PC0/ADC0/PCINT8
PC1/ADC1/PCINT9
PC2/ADC2/PCINT10
PC3/ADC3/PCINT11
PC4/ADC4/SDA/PCINT12
PC5/ADC5/SCL/PCINT13
PC6/RESET/PCINT14
ATMEGA328P
SD2
CS
DI
DO
CLK
SD
SD Card