Laboratory Exercise 9

Interface Protocols

Many embedded systems are implemented using a combination of a processor, memory, and off-the-shelf peripheral components. To communicate with a processor, peripheral components support a predefined set of communication interfaces. Serial interfaces are usually used to reduce the number of pins needed to exchange data. One of the popular communication standards is the Serial Peripheral Interface (SPI).

The SPI comprises four wires, as shown in Figure 1. They are: clock (CLK), Master-Out Slave-In (MOSI), Master-In Slave-Out (MISO) and chip select (CS). The clock signal is generated by the master to synchronize the exchange of data. The MOSI line is used by the master to send commands and data to the slave, while the MISO line is used by the slave to respond to commands and send data back to the master. The fourth line, called chip select, enables or disables the slave device. When multiple slave devices are present, a separate chip select line is used for each slave.

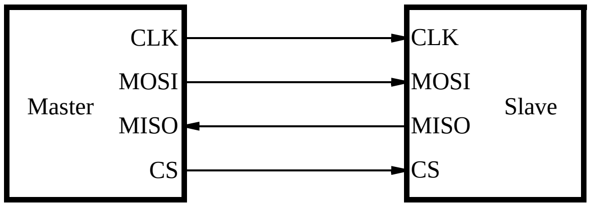

Figure 1: Diagram of SPI connecting a master and a slave.

An example of a device that supports SPI is a Secure Data (SD) card. SD cards are portable non-volatile memory devices with a capacity to store a large amount of data and provide encryption for security purposes. They are widely used to store pictures in digital cameras.

This exercise focuses on designing an embedded system on an Altera DE-series board that can access an SD card using an SPI interface.

Background

A block diagram of the SD card is shown in Figure 2. It consists of a 9-pin interface, a card controller, a memory interface and a memory core. The 9-pin interface allows the exchange of data between a connected system and the card controller. The controller can read/write data from/to the memory core using the memory core interface. In addition, several internal registers store the state of the card.

The controller responds to two types of user requests: control and data. Control requests set up the operation of the controller and allow access to the SD card registers. Data requests are used to either read data from or write data to the memory core.

To interface with an SD card, the Altera DE-series board features an SD card slot, on which pins labeled as CLK, CMD, DAT0, and CD/DAT3 are connected to the FPGA. These four pins are used to exchange data between the FPGA and the SD card using one of two modes: the SD mode or the SPI mode. By default, the SD card operates in the SD mode. However, in this exercise we will set the SD card into the SPI mode and communicate with it using the SPI protocol. Table 1 shows each pin on the SD card, what function it is used for in the SPI mode, and the pin on the FPGA to which it is connected on Altera DE-series boards.

In this exercise you will create a hardware interface to the SD card using these four pins and write software to set the SD card into SPI mode to exchange data with it using SPI.

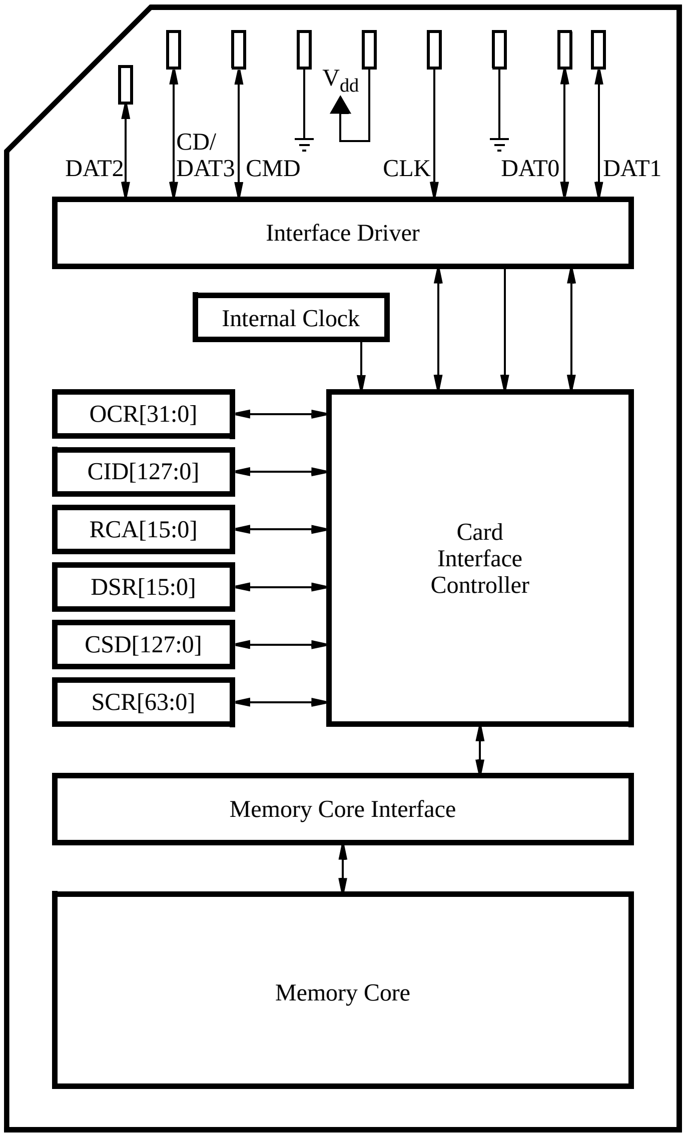

Figure 2: A block diagram of an SD card.

Table 1: SD Card Pin Function and Location Assignments for Altera DE-series boards.

SD Card Pin	**SPI Pin Function**	**FPGA Pin Location**					**Node Name**
		DE0	**DE1**	**DE2**	**DE2-70**	**DE-115**	
CMD	MOSI	PIN_Y22	PIN_Y20	PIN_Y21	PIN_W28	PIN_AD14	SD_CMD
DAT0	MISO	PIN_AA22	PIN_W20	PIN_AD24	PIN_W29	PIN_AE14	SD_DAT
CD/DAT3	CS	PIN_W21	PIN_U20	PIN_AC23	PIN_Y30	PIN_AC14	SD_DAT3
CLK	CLK	PIN_Y21	PIN_V20	PIN_AD25	PIN_T26	PIN_AE13	SD_CLK

Part I

In this part, you are to design an SPI interface to the SD card, specified in Figure 3, and implement a system with this interface component using the Qsys tool.

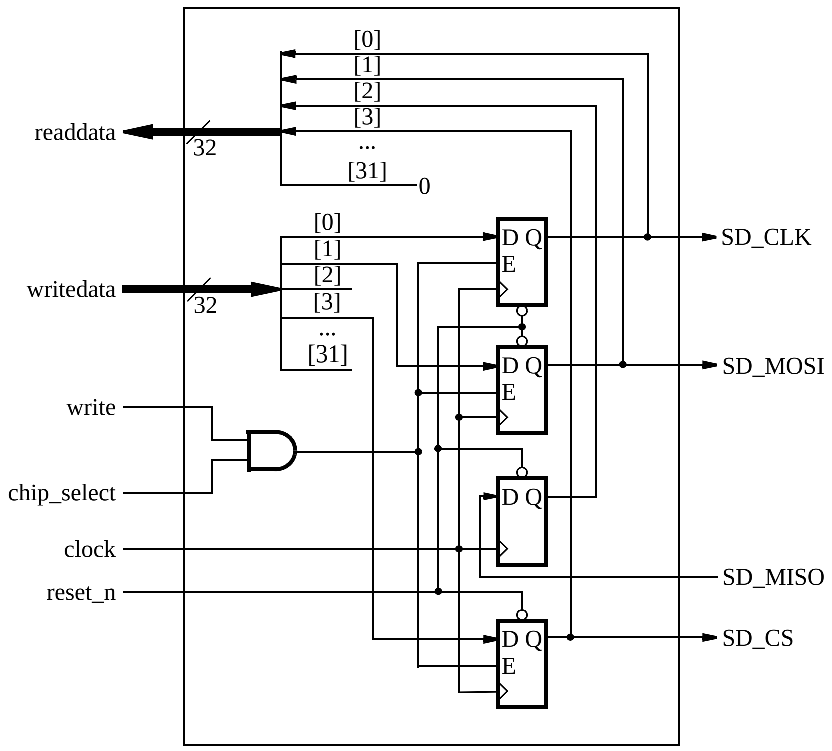

Figure 3: A circuit diagram of a simple SPI interface component.

The SPI interface component consists primarily of four flip-flops. Each flip-flop is responsible for transmitting a signal via the SD_CLK and SD_MOSI lines and receiving responses from the SD card on the SD_MISO line. The fourth SPI line, SD_CS, is used to properly initialize the SD card and set it to function in the SPI mode.

The left-hand side of Figure 3 shows several signals that serve as an Avalon memory-mapped slave interface. They comprise readdata lines to read the state of the flip-flops, writedata lines to send data through the SPI, as well as the write and chip_select lines to ensure that only data intended for this component is accepted from the Avalon Interconnect.

Implement the component shown in Figure 3 in a module called my_spi_interface. Then, include it in a Qsys system comprising:

1. Nios II processor, standard version
2. SRAM Memory Controller (available as a University Program IP Core). It should have a base address of 0x00000000.
3. JTAG UART core (you can find it under Interface Protocols/Serial)
4. Your SPI interface component, with a base address set to 0x40000000. If you don't know how to create a hardware module as a Qsys component, read the Tutorial called *Making Qsys Components* available on the Altera University Program website.

Connect all inputs and outputs to the circuit as needed, ensuring that the clock input is connected to the 50-MHz clock on the DE-series board. Compile and download your design onto the Altera DE-series board. Test your system by reading and writing data to your SPI interface component. Since there is no SD card connected on the right-hand side of Figure 3, test the component by simply writing data into the four flip-flops and reading it back.

Part II

In this part, you will use your SPI interface component to communicate with an SD card. To do this, you will set the SD card into the SPI mode and then send commands to it.

Communication with the SD card is performed by sending commands to it and receiving responses from it. A valid SD card command consists of 48 bits as shown in Figure 4. The leftmost two bits are the start bits which we set to (01). They are followed by a 6-bit command number and a 32-bit argument where additional information may be provided. Next, there are 7 bits containing a Cyclic Redundancy Check (CRC) code, followed by a single stop bit (1).

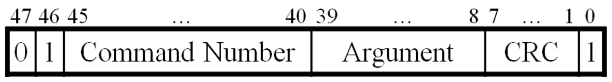

Figure 4: Format of a 48-bit command for an SD card.

The CRC code is used by the SD card to verify the integrity of a command it receives. By default, the SD card ignores the CRC bits for most commands (except CMD8) unless a user requests that CRC bits be checked upon receiving every message. In this exercise, we assume that errors during transmission of commands and data between your circuit and the SD card will not occur, and thus omit the discussion of CRC codes. Readers interested in using the CRC codes may consult the *SD Specification - Physical Layer version 2.00* that can be found on the Web.

Notation CMDX is used to represent the command with a command number X. For example, CMD8 means the command with $(001000)_2$ as its command number.

To communicate with the SD card you will write a program that runs on the Nios II processor. The program will control the values of all signals in the SPI interface component, including the SD_CLK signal, by reading and writing data to the component.

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

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

To ensure the proper operation of the SD card, the SD_CLK signal should have a frequency in the range of 100 to 400 kHz. A simple way to do this is to wait an appropriate amount of time between changing the value of the SD_CLK signal. You can experiment with various methods to generate the desired signal or use the code provided in Figure 5.

Note that the keyword **volatile** is used for the pointer to the SPI interface. This keyword is required to ensure that each processor read operation directly accesses the SPI hardware component to obtain its current value. The C compiler used in the Altera Monitor Program implements this operation by using a **ldwio** Nios II instruction. If the volatile keyword is not included, the C compiler may choose to read the SPI interface hardware only once, and then store its value in a Nios II general-purpose register. Then, subsequent accesses to this address would not provide updated values, and just use the value stored in the register.

```
volatile int *spi_interface = (int *) 0x40000000;

void create_clock_pulse(void) {
    int index;
    int contents;

    for (index = 0; index ≤ 15; index++){
        contents = *spi_interface;
    }
    *spi_interface = contents | 0x01;
    for (index = 0; index ≤ 15; index++){
        contents = *spi_interface;
    }
    *spi_interface = contents & 0x0E;
}
```

Figure 5: C-language code to generate an SD_CLK pulse.

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

01 000000 00000000 00000000 00000000 00000000 1001010 1

This is the reset command, which puts the SD card into the SPI mode if executed when the CS line is low. The SD card will respond to the reset command by sending a basic 8-bit response on the MISO line. The structure of this response is shown in Figure 6. The first bit is always a 0, while the other bits specify any errors that may have occured when processing the last message. If the command you sent was successfully received, then you will receive the message $(00000001)_2$.

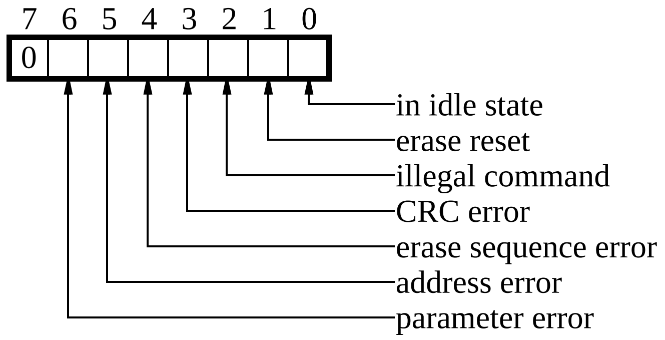

Figure 6: Format of a basic 8-bit response to every command in SPI mode.

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

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

1. 01 001000 00000000 00000000 00000001 10101010 0000111 1 (CMD8) - this command is only available in the latest cards, compatible with SD card Specifications version 2.00. It is explained in more detail later in the exercise. For most older cards this command should fail and cause the SD card to respond with a message that this command is illegal.

2. 01 111010 00000000 00000000 00000000 00000000 0111010 1 (CMD58) - request the contents of the operating conditions register for the connected card.

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

Write a C-language program that allows a user to input a 48-bit command on the keyboard and send it to the SD card via SPI. Your program should observe the MISO line for at least 16 clock cycles after each command is sent. If during this time the SD card starts sending a response, your program should continue to toggle the SD_CLK line until all bits in the response have been received, and then for another 8 cycles. It should then print out the response in the terminal window. Remember to access the memory-mapped SPI interface using a volatile pointer in C code to ensure that the SPI interface is read properly, as discussed for the example code in Figure 5.

Part III

In this part, you will implement a complete SD card initialization routine to allow you to exchange data with the SD card. To initialize the card correctly, you will have to send several commands in a sequence and process the responses received. Once you complete this procedure, you will be able to access the data stored on the SD card.

The SD card protocol supports many commands. In this exercise we will use only a few commands, which are listed in Table 2. Reserved bits in the argument field will be ingored by the SD card.

Table 2: Detailed Command Description.

Command Number	**Name**	**Argument**	**Width of Response**	**Command Description**
`CMD0`	`GO_ IDLE_ STATE`	`[31:0]reserved bits`	`8`	`Resets the SD card.`
`CMD8`	`SEND_IF_ COND`	`[31:12]reserved bits` `[11:8]supply voltage` `[7:0]check pattern`	`40`	`Sends SD Card interface a condition and asks the card whether it supports the voltage specified in the argument.`
`CMD17`	`READ_ SINGLE_ BLOCK`	`[31:0]data address`	`8`	`Reads a block of memory selected by the data address.`
`CMD55`	`APP_CMD`	`[31:0]reserved bits`	`8`	`Informs the card that the next command is an application specific command rather than a standard command.`
`CMD58`	`READ_OCR`	`[31:0]reserved bits`	`40`	`Reads the OCR register of the card. CCS bit is assigned to OCR[30].`
`ACMD41`	`SD_SEND_ OP_COND`	`[31]resetved bit [30]HCS` `[29:0]reserved bits`	`8`	`Sends host capacity support (HCS) information and activates the card's initialization process.`

The steps necessary to complete the SD card initialization are shown in the flowchart in Figure 7. The flowchart consists of boxes in which a command number is specified. Starting at the top of the flowchart, each command has to be sent to the SD card, and a response has to be received. In some cases, a response from the card needs to be processed to decide the next course of action, as indicated by the diamond-shaped decision boxes.

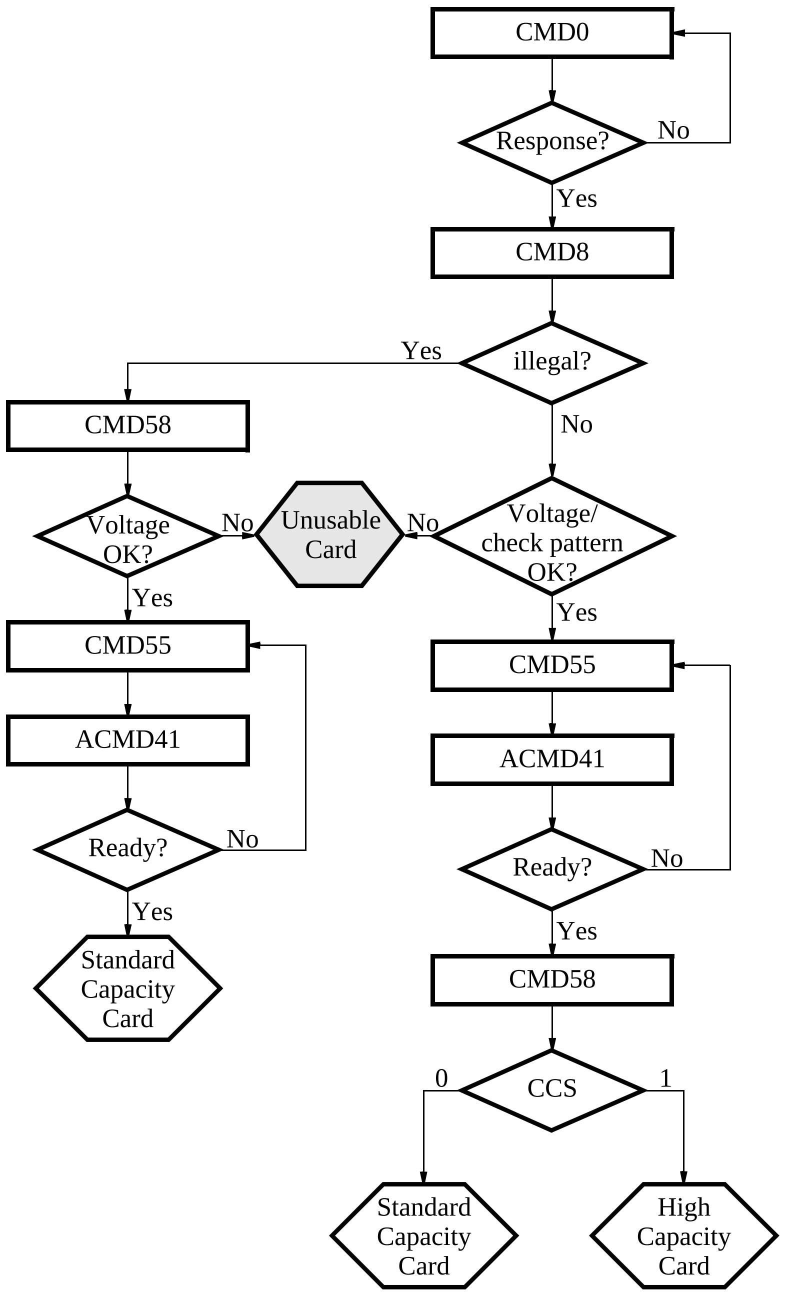

Figure 7: Flowchart representing the SD card initialization routine in SPI mode.

To begin initialization, your program should first set the SD card into the SPI mode as in Part II. That is, it should set the MOSI and CS lines to logic 1 and toggle the SD_CLK for at least 74 cycles. Next, your program should follow the initialization procedure outlined in Figure 7, while keeping the CS line low at all times.

The first command to send is the reset command, CMD0, which together with the CS set to 0 places the SD card into SPI mode. The SD card should respond to this command with the 8-bit message $(00000001)_2$ indicating no errors. However, should the SD card fail to respond, or respond with an error, another reset command should be sent until the card is reset properly.

The second command, CMD8, determines if the SD card is compliant with version 2.00 of the SD card specifications. To perform this step successfully, send the 48-bit command CMD8:

01 001000 00000000 00000000 00000001 10101010 0000111 1

Its argument field specifies the valid supply voltage range that the DE-series board supports. The last group of 8 bits of the argument are called a check pattern, and have been set to an alternating pattern of 0s and 1s. Following the command argument, the CRC code is specified. The CRC code for this command must be correct or the command will be ignored.

In reply to CMD8, the SD card will issue a 40-bit response in the format shown in Figure 8. The response indicates how to proceed with the initialization. In particular, if bit 34 is set to 1 then the SD card is an older model and should be initialized following the left branch of the flowchart in Figure 7. If the command is valid and no other errors occured, then verify if the check pattern in the response is 10101010 and that the voltage field is set to 0001. If either of these conditions fails, the SD card will not function correctly and you should stop the initialization procedure. Otherwise, proceed with the initialization steps of version 2.00 compliant SD card, following the right-hand side of the flowchart.

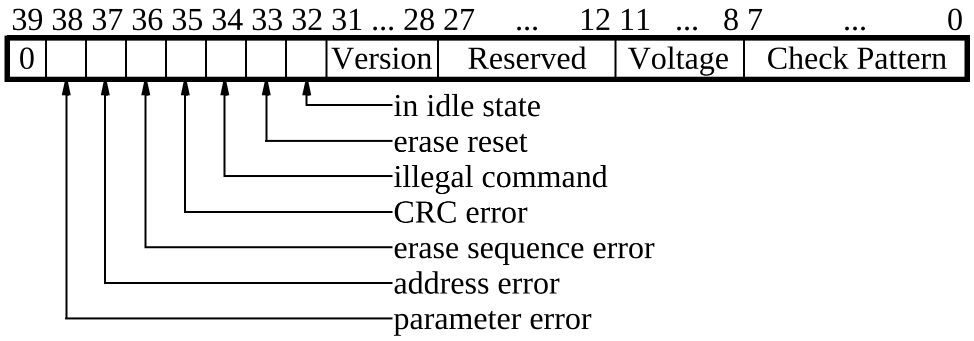

Figure 8: Format of the response to CMD8.

In the case of an older SD card, the next step of the initialization procedure is to send CMD58 to obtain the contents of the Operating Conditions Register (OCR). The OCR indicates the voltage levels the SD card can work with. To work correctly with the DE-series board, the SD card must be able to operate with supply voltage of 3.3V, which is indicated by bit 21 of the response. If this bit is not set to 1, then the card will not work properly with the DE-series board. Otherwise, wait for the card to be ready to exchange data.

To determine if the card is ready for exchange of data, your program should send two commands, CMD55 followed by ACMD41, to the SD card. The argument in each of these commands must be 0. After each command is sent, the SD card will respond. Of particular interest is the response to command ACMD41. The response is a basic 8-bit response as in Figure 6. It is expected that the bit ”in idle state” will remain as 1 for a while. However, the bit will be set to 0 when the card is ready. If it is not, send the pair of commands again and again until the response received is 00000000_2.

Write a program that initializes the SD card as described above. Your program should send the sequence of commands (all with the argument field set to 0, except for CMD8) to initialize the SD card and display the response to each command in the terminal window. When needed, your program should scan the response and proceed with the initialization only if the correct response has been received.

To simplify your task, implement only the initialization for the SD card that you use for this exercise. Consult the manufacturer’s datasheet to determine if your SD card is compliant with version 2.00 of the SD card specifications.

Part IV

In this part, you will use a read command to access data on the SD card.

CMD17 can be used to read a 512-byte segment of data from the card. A standard-capacity SD card takes as argument the byte-wise address of the data to be accessed on the card and returns a block of 512 bytes that begins at the specified address. The command is only valid if the argument is a multiple of 512. Any other address passed as an argument will result in response with the "address error" bit set.

If a correct address is provided as an argument in CMD17 and the card is able to access the specified memory segment, a response $(00000000)_2$ giving an indication of no errors will be sent by the SD card. Following this response, the card will immediately begin sending the requested data on the MISO line. The data sent by the card should be ignored until a byte comprising $(FE)_{16}$ is received. This byte signifies the beginning of a data packet arranged as shown in Figure 9.

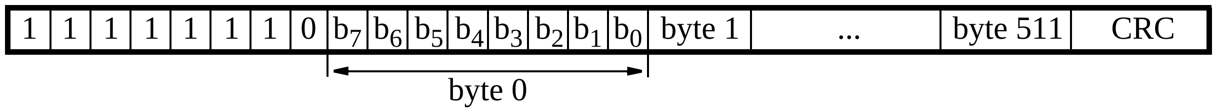

Figure 9: Format of a data packet.

The data packet consists of a leading $(FE)_{16}$ byte, 512 bytes of data, and a 16-bit CRC code. The data is arranged from the least-significant to the most-significant byte. Each byte of the data starts with the most-significant and ends with the least-significant bit.

Write a C-language program that reads any user-specified segment of the SD card and displays it in the terminal window. Your program should ask the user for the segment number (0 to (card size / 512)), send CMD17 with the appropriate address, and read the specified 512-byte segment of data from the SD card. It should then display the data in the terminal window 16 bytes per line in hexdecimal notation .

Addendum

This exercise shows how to access an SD card using the SPI protocol. In most applications, the purpose of using the SD card is to provide access to data.

Usually, files on the SD card are stored with the help of a file system. This allows different systems to access the same set of files. A popular choice is a FAT (File Allocation Table) file system, because it is relatively easy to implement. Although the details of the file system are beyond the scope of this exercise, completing all of the parts of this exercise provides a sufficient basis to read the contents of an SD card. If the card contains a file system, the next step should be to implement routines to read the appropriate sections of the SD card memory to decode the locations and sizes of files stored on it.

Troubleshooting

A secure data card is a sensitive device that may easily malfunction if proper initialization and operating procedures are not followed. To ensure successful completion of this exercise, we recommend that the instructions in each part be followed very carefully. In particular, be aware that:

1. The initialization procedure works best when it follows 74 (or more) SD_CLK cycles during which both the CS and the MOSI lines are set high. Only afterwards should the CS line be set to 0. Failure to do so causes the first few instructions sent to the card to fail.

2. After sending each instruction, exactly 8 SD_CLK cycles with MOSI line set to 1 must be sent to the SD card. A second command should not follow until this condition is met. Otherwise, the SD card may malfunction. It is best to wait for a response after each instruction before sending another command.

3. It is important to wait for the $(FE)_{16}$ byte as the start of the data packet. You may not assume that there is only a leading 0 bit followed by 512 bits of data, as the MISO line is not guaranteed to be set to 1 prior to the start of the data packet.

Preparation

The recommended preparation for this laboratory exercise includes:

1. HDL code, Quartus II, and Qsys project for Part I
2. C code for Parts II through IV