Lab 7: SD Card

NOTE: This lab is in the early stages, you may run into problems while working through it. Just a heads up! If you have questions or need any help, you can email me at abe.nelson@tufts.edu. I am happy to help however I can.

Prelab

While there is no formal prelab for this option, there is a lot of content to understand for the SD card to work! Please spend some time reading this document and the resources linked below.

Resources

Here are the resources I used while researching and creating this lab. We will be using the SPI interface to communicate with the SD card. The good news is that SPI is a very common interface, so it is a great protocol to know in general, not just for interfacing with SD cards!

- SD SPI information
  - How to Use MMC/SDC
    Goes into detail about the initialization sequence, commands
  - Lecture 12: SPI and SD Cards
    Covers the steps necessary to implement SPI communication on a microcontroller
- Example Code
  - VHDL SPI Interface (GitHub)
  - My MicroPython code is included at the end of this document

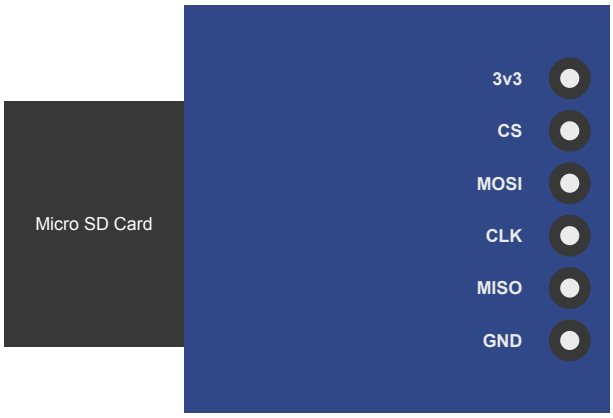
Testing/Debugging

I would highly recommend getting familiar with the Digital Discovery for this assignment. It will make debugging much easier, because you will be able to see exactly what your circuit is doing, and what the SD card is responding with

SPI

While SD cards have a "native" operating mode, they also support an SPI mode. This is the mode will will be using to communicate with the SD card, as it is more simple. However, there are still some parts that are a bit tricky.

Pinout



SPI Protocol

SPI is a simple *synchronous* protocol. It consists of 4 wires:

- A clock signal (CLK/SCK)
- A data in signal (MOSI) – FPGA ⟶ SD

- A data out signal (MISO) – FPGA ← SD
- A chip select signal (CS)

We will be operation our side of things in SPI "Master" mode, meaning we are responsible for supplying the CS and CLK signals. The SD card will receive the CS and CLK signals along with data through the MOSI line, and will respond with data on the MISO line.

Sparkfun has a great resource on SPI if you are interested: https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi. The diagrams in particular are helpful for understanding the behavior we are trying to implement.

SD Commands

To communicate with the SD card, we send it commands. SD Commands consist of 48 bits (6 byte) divided up as follows:

Bits	47	46	45 40	39 8	7 0
Value	`0`	`1`	Command ID	Argument	CRC

The first two bits are always `01`. The next 6 bits are the command ID. This website has a list of all the commands and their IDs. The next 32 bits are the argument for the command. This argument varies between commands, but for example when reading a block of data from the SD card, this argument would be the address of the block.

The last 7 bits are the CRC. This is a type of error checking code. Fortunately, once we put our SD card into SPI mode, we can ignore the CRC. For the first few commands we will just "hard code" the CRC values. These values will be provided below.

SD Card Response

Most commands will return a 1 byte response (called the R1 response). This consists of 7 flags that can be set by the SD card.

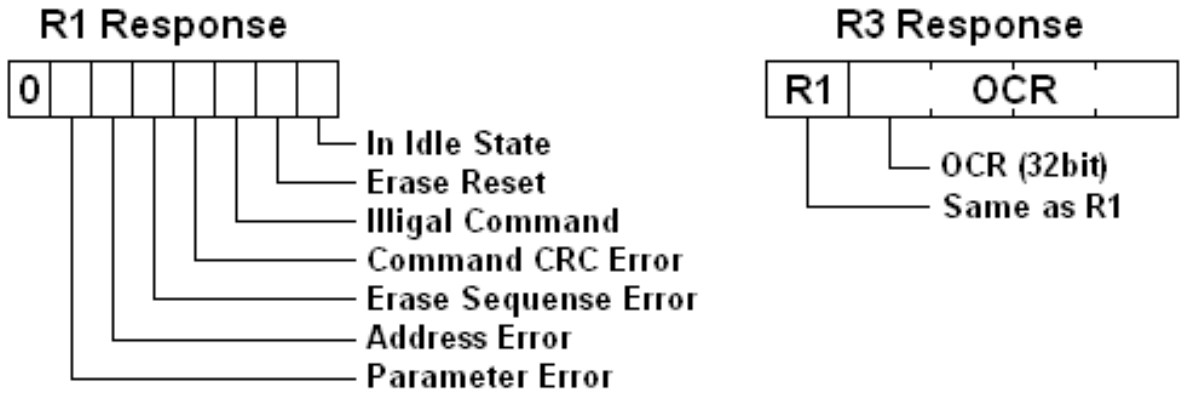


Figure 2: Most flags represent errors, so hopefully we don't see them! The least significant bit is the "idle" bit, which should be high during initialization, and low once we are ready to read from the card.

Initialization Sequence

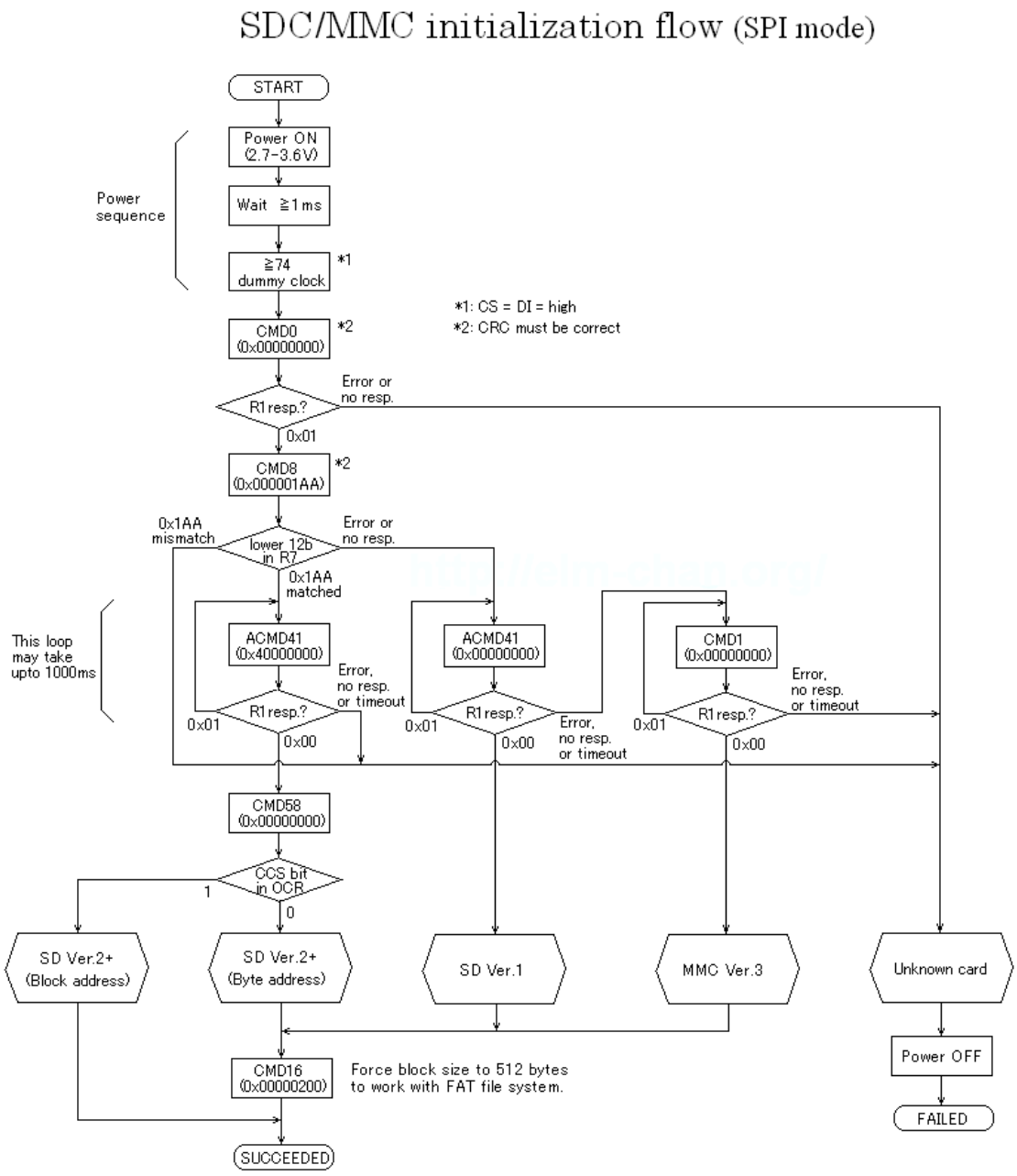


Figure 3: Initialization Flow from http://elm-chan.org/docs/mmc/mmc_e.html

The initialization sequence for the SD card consists of a few commands. These commands are also detailed here : How to Use MMC/SDC.

Figure 3 shows a detailed flow for the initialization sequence based on all the possible SD card types. Because we know our SD card type, we won't have to worry about all the cases.

The initialization sequence for our specific card is below.

Power Sequence

After the SD card is supplied power (3.3V), it will take a few ms to power up. After that, we must send it at least 74 clock cycles to allow it to internally initialize. At the same time, the CS and MOSI lines should be high. This can be done with SPI by sending at least 10 bytes of `0xFF`. Sending `0xFF` is the same as keeping MOSI high.

Throughout this initialization stage we should be using a clock frequency of 100-400 kHz. If we were to use our 48 MHz clock, we would need to divide it by 128 to get it into the correct range.

Then, set CS low. It will remain low for the rest of the communication.

Soft Reset

Name	Command Data (binary)	Command Data (hex)
`CMD0`	01 000000 00000000000000000000000000000000 10010101	`0x40 00000000 95`

CMD0 performs a "soft reset" on the SD card, putting it into idle state. This command also serves to put the SD card into SPI mode. This is done by setting the CS line low while sending the command.

If everything goes to plan, the SD card should respond with an R1 response: `00000001`. The last bit means that the SD card is in idle state, and has been successfully put into SPI mode. Take a look at **Figure 6** for how this SPI communication should look in hardware.

Check Voltage Range / Version

Name	Command Data (binary)	Command Data (hex)
`CMD8`	01 001000 00000000000000000000000110101010 10001111	`0x48 000001AA 87`

CMD8 is used to check the voltage range on V2 SD cards. Our SD card is a V2 card, so we will receive a valid response from our card. The response is 5 bytes and consists of the following fields:

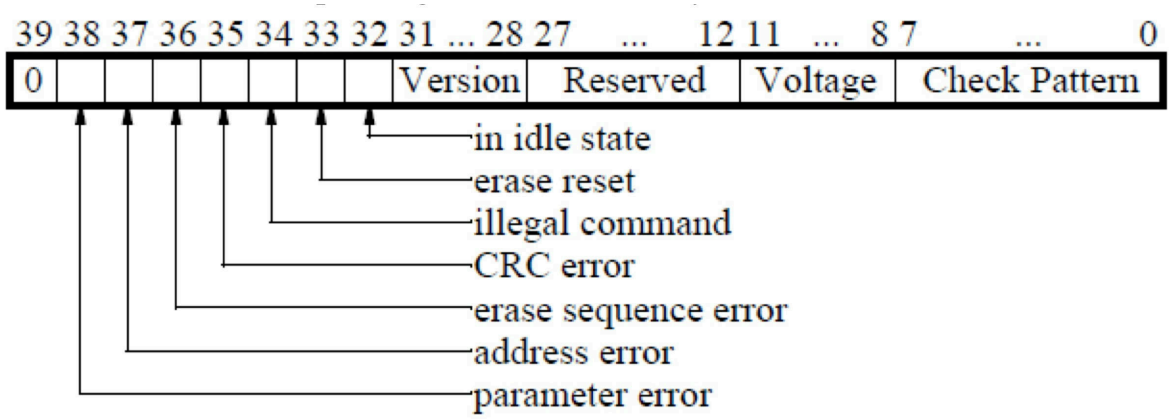


Figure 4: The first byte is the same as the R1 response, while the remaining bytes provide additional information. From https://www.dejazzer.com/ee379/lecture_notes/lec12_sd_card.pdf

Take a look at **Figure 7** for how this SPI communication should look in hardware.

Initialization loop

Next we need to tell the SD card to go from idle state to "ready" state. Depending on the version, there is a few commands that does this. For our cards, we will use `ACMD41`. `ACMD` stands for Application Command, and is actually made up of the two commands `CMD55` and `CMD41`. `CMD55` is used to tell the SD card that the next command is an "application specific" command. `CMD41` is the command that actually tells the SD card to go from idle to ready state.

Name	Command Data (binary)	Command Data (hex)
`CMD55`	01 110111 00000000000000000000000000000000 00000000	`0x77 00000000 00`

After sending CMD55, we will receive the R1 response. We will then send CMD41.

Name	Command Data (binary)	Command Data (hex)
`CMD41`	01 101001 01000000000000000000000000000000 00000000	`0x69 40000000 00`

> **NOTE:** You might have noticed that the argument for CMD41 is `0x40000000`. This is because we want to set the "High Capacity" bit high in the argument.
>
> A 32 bit address is only enough to address $2^{32} = 4$GB of data. But our card is 16 GB! The solution is that the SD card will use this High Capacity mode, which addresses the card in **blocks** of 512 bytes, instead of individual bytes. For example, the address 0 would refer to the first 512 blocks available on the card.

We will receive the R1 response for CMD41. If the SD card is still in idle state, the response will be `00000001`. In this case, we will send `ACMD41` again.

If the card is in ready state, the response will be `00000000`, and we are ready to move on to reading data!

From this point on, we can increase the clock speed, allowing us to read data faster. SDC cards like ours can operate at up to 25 MHz. This means dividing our 48 MHz clock by 2 will give us a 24 MHz clock, which is within the range.

Reading from our card

To read from our card, we use CMD17. As mentioned before, we read from this card in 512 byte chunks. Because the argument is `0` below, we are reading form the first 512 byte block on the card.

Name	Command Data (binary)	Command Data (hex)
`CMD17`	01 010111 00000000000000000000000000000000 00000000	`0x57 00000000 00`

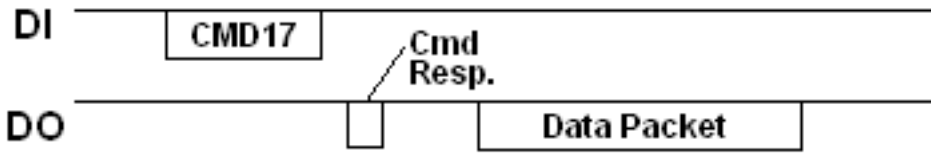


Figure 5: The CMD17 first responds with a R1 response, then sends its 512 bytes of data. From http://elm-chan.org/docs/mmc/mmc_e.html

Additional Resources/Notes

Digital Discovery

I relied heavily on the Digital Discovery throughout this lab, and I reccomend you do the same. Because the SD card gives you no visual feedback, it is near impossible to figure out what is working and what isn't without a tool like the Digital Discovery.

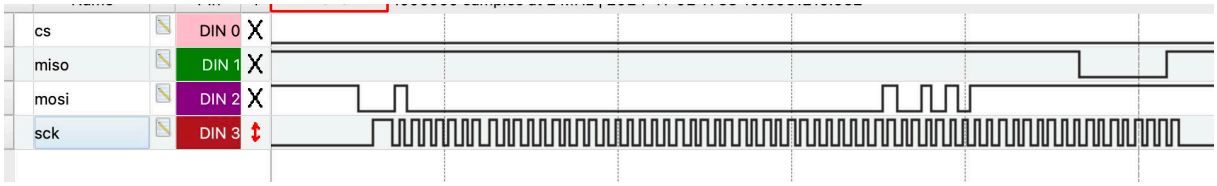


Figure 6: A Waveforms screenshot of the CMD0 command.

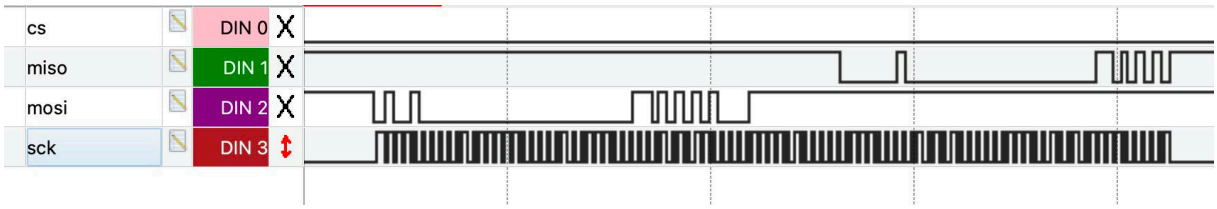


Figure 7: A Waveforms screenshot of the CMD8 command.

MicroPython implementation

If you are familiar with Python, it may help to look through this micropython code that has been tested with our SanDisk SD cards. It is a good reference for how to send the necessary commands to the SD card.

```
# necessary micropython imports
import machine
from machine import Pin
import time

# define the pins to be used for the SPI interface
sck = Pin(14)
mosi = Pin(13)
miso = Pin(12)
cs = Pin(5, Pin.OUT, value = 1)

# define the read buffer for reading data from the SD card later
buf = bytearray(1024)

spi = machine.SoftSPI(sck=sck, mosi=mosi, miso=miso)

def create_cmd(cmd, arg, crc, read_bytes=0):
    cs.value(0)

    write_buf = bytearray(8 + read_bytes)
    read_buf = bytearray(8 + read_bytes)

    write_buf[0] = 0xFF # sending an empty byte before the cmd seems to help
    # the two starting bits of the command is always 01, so we OR it with 0x40
    write_buf[1] = 0x40 | cmd

    # the cmd is 32 bits
    write_buf[2] = arg >> 24
    write_buf[3] = arg >> 16
    write_buf[4] = arg >> 8
    write_buf[5] = arg

    # write the crc bit
    write_buf[6] = crc
    # another "dummy" byte that allows the SD card to internally produce its response
    write_buf[7] = 0xFF # "wait" byte

    # depending on how many bytes we are trying to read,
    # write 0xFF for those (MOSI = HIGH)
    for i in range(8, 8+read_bytes):
        write_buf[i] = 0xFF
```

```
    spi.write_readinto(write_buf, read_buf)

    # return any of the read bytes
    return bytes(read_buf[7:])

# and apply 74 or more clock pulses to SCLK.
spi.read(16, 0xFF)

# The card will enter its native operating mode
# and go ready to accept native command.

def read_res(num_bytes=1):
    return spi.read(num_bytes, 0xFF)

time.sleep_ms(10)

# CMD0, software reset
print(create_cmd(0, 0, 0x95, 1))

# CMD8 voltage check
print(create_cmd(8, 0x01AA, 0x87, 5))

# Send the init command (ACMD41) until the IDLE bit is cleared in the SD response
for _ in range(100):
    create_cmd(55, 0, 0, 2)
    r = create_cmd(41, 0x40000000, 0, 2)
    print(r[1])
    if r[1] == 0:
        print("done init!")
        break

# read from the SD card! In this case, we are reading from address 0x4000!
print(create_cmd(17, 0x004000, 0, 4))
spi.readinto(buf, 0xFF)
print(buf)
```