**CpE5160 Experiment #2**
**SPI Peripheral and SD Card Initialization**

## Introduction

The purpose of this experiment is to set up the SPI peripheral and use it to initialize the SD card. The serial port routines developed in experiment #1 will be used to display the information read from the SD card. Information read from the SD card and displayed will be used in the fourth experiment to investigate the FAT file system. This experiment will also encourage the use of some hardware debugging tools such as an oscilloscope or a logic analyzer.

## Schematic

The schematic shows the connection of the SD card using a surface mount socket and surface mount adapter. The details of the pin connections are also shown in a diagram on the schematic. The SD card pins and corresponding SPI connections are given in this diagram as well. The port pins on the schematic and diagram are the predefined SPI peripheral pins from the microcontroller datasheet. The nCS signal needs to be connected a general purpose I/O pin. It will be switched low for each SPI data transfer to the SD card. The CARD SWITCH, LOCK SWITCH and SWITCH COMMON pins refer to two switches on the SD Card socket that are closed when an SD Card is inserted and when the write protect slider is in the lock position. The programmer can connect these to general purpose I/O or they can be ignored for this project.

The functions needed for using the SPI to initialize and communicate with the SD card are discussed in the following paragraphs.

## SPI Peripheral functions

The functions that access the SPI registers are hardware specific and should be placed in their own source code file for easier portability.

Any peripheral device needs an initialization routine to set it up for the specified operating parameters. The SPI peripheral requires three items to be set up. The first is the clock rate for transmission. The AT89C51RC2 has seven choices for clock rate that divide the peripheral clock by a power of 2 from $2^1$ to $2^7$ or 2 to 128. When initializing the SD card, the initial setting of the clock should be 400 KHz or less. It can be increased after the SD card has been initialized. The second item is setting the clock phase and polarity. The clock polarity bit (CPOL) determines the state of the clock during the idle state. If CPOL='0', then the clock will be at '0' when no transmission is occurring and the first clock edge will be from low to high. If CPOL='1', then the clock will be at '1' when no transmission is occurring and the first clock edge will be from high to low. There are two possible choices of clock phase. If the clock phase bit (CPHA) is '0', then the data is sampled on the first clock edge so the data must be shifted before the first clock edge. If CPHA='1', then the data is shifted on the first clock edge and sampled at the second clock edge. The SD card specification does not have the required timing information for determining the proper clock phase and polarity, but CPHA='0' and CPOL='0' worked for me. The final item is to determine if the microcontroller will act

as a master or slave device and in this experiment the microcontroller is the master device. Make sure the SPEN bit (SPI Port Enable) is set to '1' to enable the SPI port, the MSTR bit is set for master mode and make sure the SSDIS (Slave Select Disable) bit is set to disable the slave select pin.

Another SPI function that is needed is a transmit/receive function. Each time a transmitted value is shifted out of the SPI port, a received value is shifted in. Therefore separate transmit and receive functions are not needed. A single SPI Transfer function will handle both functions. A transfer is started by writing a byte value to SPDAT. The SPIF bit in the SPSTA register will be low during the transfer and it will be set to '1' when the transfer is complete. The received data can then be read from SPDAT.

The programmer may wish to do some error checking. The WCOL bit in SPSTA will be '1' when a write collision has occurred which means a value was written to SPDAT while a transfer was in progress. The MODF (mode fault) bit in SPSTA is set to '1' when the /SS (Slave Select) pin is pulled low, which indicates that an external master device is selecting the microcontroller as a slave device, and the SPI port is set to master mode. In single master systems, this can be prevented by setting the SSDIS (Slave Select Disable) bit to '1' in SPCON to disable the slave select pin. The other bit is used when the microcontroller is acting as a slave device. The SSERR (slave select error) bit is set to '1' if the /SS (Slave Select) pin, which would be controlled by an external master device, goes to '1' before the transfer is complete.

**SD Card functions**

The first SD card function that will be needed is one that can send a command to the SD card. Every SD card command consists of 6 bytes. The first byte starts with a leading '0' and '1' bit for the START and TRANSMISSION bits. The rest of the byte is the six bit command number. The next four bytes are the argument as a 32-bit number which is sent most significant byte (MSB) first. Not all commands require an argument, but the 32-bit number is always sent. The command summary table found in the SD card specification refers to arguments that are not needed as "stuff bits" and they are sent as zeros. The function will need to break the 32-bit value into four bytes that can be sent MSB first. The final byte is a checksum. The SD card specification shows how to calculate the checksum, but it is only required for the CMD0 and CMD8 commands. The checksums can be defined in the C code to be sent when CMD0 or CMD8 is sent. The checksum for CMD0 is '1001010.' The final bit of the last byte is the end bit which should always be a '1.' Therefore, the last byte sent for CMD0 is 0x95. The checksum for CMD8 depends upon the argument sent. The first two bytes and the upper half of the third byte of the argument are reserved bits and should be '0.' The lower half of the third byte indicates the host supply voltage. The least significant bit of the third byte should be a '1' to indicate a 2.7V to 3.6V host supply voltage. The least significant byte is a check pattern which will be echoed back in the response. If a check pattern of 0xAA is used resulting in an argument of 0x000001AA, then the checksum will be '1000011.' The END bit of '1' is placed in the least significant bit (lsb) of the final byte which results in the value of 0x87 for CMD8. The last byte sent for all other commands is 0x01.

Each command sent to the SD card causes the SD card to send a response. The response may be 1 to 5 bytes long depending on the type of response expected. Some of the commands will cause the SD card to respond with several data bytes. These responses are a little different and will be handled by a different function. The first byte of a response is always the R1 response which has the most significant bit (msb) cleared to '0.' There is a delay between the command being sent and the response being received. Therefore, the function should repeatedly send 0xFF to the SD card while it is waiting for a response and while it is receiving the response. The SD card will respond with 0xFF or the R1 response which can be detected by the msb being '0.' As with any wait loop, good programing practice should include a loop timeout to exit the loop in the event of an error or hardware failure. The R1 response indicates if an error has occurred by setting bits in the response. This response should be checked when more bytes are expected because if an error occurs, then only the R1 response is sent. If more bytes are expected in the response and the R1 response shows no errors, then the function should keep sending 0xFF and reading the data received for the other bytes of the response. After all of the bytes of the response are received, a final 0xFF is sent to allow the SD card to go to standby mode. The response needs to be returned by the function, but multiple values cannot be returned from a function. There are two possible approaches. One is to return a pointer to an array which holds the response. The pointer cannot be to a local array, because the local array will no longer be valid after the function exits. Another approach is to place the response values in an array sent by the calling function and return an error value which indicates if a timeout or other error occurred. One required input parameter for this function is either the type of response expected or the number of bytes expected in the response. A pointer to an array outside of this function should also be passed to the function to hold the response values.

A different response function should be used when the SD card is expected to respond with bytes of data. The response when the SD card is sending data starts with the R1 response after a short delay like the other responses. The SD card must be in active mode to read data from it so the R1 response must be 0x00. This is followed by another short delay. The block of data will start with the START BLOCK token which is the byte 0xFE. The data follows immediately with the next byte. The number of bytes that follow depend on the command that was sent and the SD card setup. For example, the SEND_CSD (CMD9) or SEND_CID (CMD10) commands cause the SD card to respond with 16 bytes of data. The READ_SINGLE_BLOCK (CMD17) command causes the SD card to respond with one block of data. On a standard capacity SD card, a single data block may be configured to be one byte, which is the default, up to 1024 bytes. A high capacity SD card has a fixed block size of 512 bytes. Since the number of bytes expected in a data block is not always the same, the number should be specified as an input parameter. The data bytes are immediately followed by two bytes which are the CRC16 checksum. Just like the previous response function, a final 0xFF should be sent to allow the SD card time to go to standby mode. Possible input parameters for this routine could be the number of bytes to be read and the address of an array where the data bytes are to be stored. An output parameter could be an error flag that indicates an R1 response that is different than expected, a data error token was received or that a timeout error occurred.

**SD Card Initialization**

The steps for SD card initialization can be found in a flow chart in figure 7-2 in section 7.2 SPI Bus Protocol of the SD Specifications document. Those steps are explained in the following paragraphs. The initialization can be written as a function that has a return value that can indicate if the initialization was successful or if an error occurred. A suggestion is to proceed each step with an "if statement" that checks if any errors have occurred. If an error occurs, then none of the rest of the steps will be executed. Another suggestion is to have a printed output after each step to show what command was sent and what the response was from the SD card. This will help the programmer know where the initialization stopped.

The first step is to set the SPI peripheral to a clock rate of 400KHz or less. The nCS pin on the SD card should be set high and then at least 74 clock pulses are applied to the SCK pin. This will initialize the SD card into SD card mode. The clock pulses can be manually generated or at least 10 bytes can be sent through the SPI port (10*8=80 clock pulses). Set the nCS pin to '0' and send the SD card command 0 (CMD0). The argument for CMD0 is 0x00000000. The expected response for this command is R1=0x01 which indicates that the SD card is in the idle state. If the R1 response is not 0x01 or if there is no response, then an error flag should be set and the initialization routine should stop. The nCS pin should remain at '0' for the command and response and then it can be switched back to '1.' The nCS pin should be switched to '0' before each SD Card command and remain at '0' until the entire response is received.

The next command is CMD8. This command checks the interface voltage for the SD card. The argument consists of two parts and some reserved bits. Bits 31 through 12 are reserved bits and should be set to '0.' Bits 11 through 8 are the VHS (host supplied voltage) field. Table 4-16 in the SD card specification gives the valid values for this field. Bits 7 through 0 are a check field. Any values can be placed in this field and they will be echoed back in the response. The response should be the R7 response which is five bytes long. The details of all of the possible responses can be found in section 4.9 Responses in the SD Specifications document. The first byte is the R1 response and should be 0x01. If this byte is 0x05, it indicates an illegal command and that the SD card is an older version card (v1.x). No more bytes will follow this response. The programmer may wish to have a variable to store the SD card version. This will allow the initialization routine to support a wider range of SD cards. If the R1 response is the 0x01 idle state response, then four more bytes will immediately follow. Bits 31 through 28 indicate the command version and should be '0000.' Bits 27 through 12 are reserved bits and will also be '0.' Bits 11 through 0 should be an echo of the bits sent in the command indicating that the voltage is compatible. If the R1 response is neither 0x01 nor 0x05, then an error flag should be returned and the initialization should stop. If the voltage is not compatible, then an error flag should be returned and the initialization should stop. If the check byte does not match, then the initialization should stop.

If CMD8 is successful, then CMD58 is sent. The argument is 0. This command reads the operating conditions register and the response is the R3 response. The response

consists of five bytes. Again the first byte is the R1 response and this is followed by the four byte operating conditions register. See section 4.9 Responses and section 5.1 OCR register in the SD Specifications document for details on the operating conditions register. The main concern at this point is the voltage range specified by this register and that it matches our 3.3V power supply. As with the previous steps, if the R1 response is not 0x01 or if the voltage is incompatible with 3.3V, then an error flag should be returned and the initialization should stop.

The next step is to send ACMD41. Commands that start with an "A" are application specific commands. This means that they must be preceded with CMD55 (APP_CMD). The argument for CMD55 is 0 and the response is the R1 response. Send the ACMD41 next with an argument that has bit 30 (the host capacity support (HCS) bit) set to '1' to indicate that the host supports high capacity SD cards. If the programmer wishes to support older SD cards, then the ACMD41 argument should be 0 when v1.x SD cards are detected. This command also starts the SD card initialization process. The response should be the R1 response; however it could be 0x01 to indicate the idle state or 0x00 to indicate the active state. The nCS pin should remain '0' for both CMD55 and ACMD41 and their responses. If the SD card responds with 0x01 indicating the idle state, then the command sequence of CMD55 and ACMD41 should be repeated and continue to be repeated until the active state response is returned or a loop timeout occurs. The nCS pin may remain at '0' for the entire time of these commands if desired. If the R1 response is neither 0x01 nor 0x00 or if a timeout occurs, then an error flag should be returned and the initialization should stop.

If a v2.0 SD card has been detected and the HCS bit was set in ACMD41, then CMD58 should be sent again. This time the card capacity status (CCS) bit and the power up status bit should be checked when the R3 response is received. The CCS bit is only valid if the power up status bit is a '1' indicating the power up routine is finished. If the CCS bit is '1', it indicates that the SD card is a high capacity card and a '0' indicates that the SD card is a standard capacity card. High capacity cards have a fixed block size of 512 bytes and standard capacity cards have a user defined block size. It is recommended to switch standard capacity cards to have a block size of 512 bytes to be more compatible with the rest of the software for this project. This is done with CMD16 and the argument is the block size in bytes.

This completes the initialization. The SPI port can now be switch to a higher frequency. The maximum frequency for an SD Card is 25MHz, however, the maximum SPI frequency of our microcontroller is the peripheral clock frequency divided by 2. If you are using the 18.432MHz crystal and x2 mode, this sets the maximum peripheral clock frequency of 18.432MHz and the maximum SPI clock frequency of 9.216MHz.

Information can then be read from the SD card after the initialization is complete. There are three recommended commands that can be used to read information from the SD card. CMD9 (SEND_ CSD) reads the card specific data register. It is a 16-byte register which is sent as a data block. As stated earlier, the response when receiving a data block is a little different. The SD card will first send the R1 response after a short delay. The

R1 response must be 0x00 or no other information will be sent. After another short delay, then the data block starts with the data block start character 0xFE. If the data block start character is not 0xFE, then this means an error has occurred and no data will be sent. The data block immediately follows the data start character and it is followed by a 2-byte checksum. A final 0xFF should be sent to the SD card to allow it to go to its standby state. CMD10 (SEND_CID) reads the card identification register and also responds with a 16-byte data block. CMD17 is the READ_SINGLE_BLOCK command and responds with the data block specified in the argument of the command. In high capacity cards the argument in the data block number and the response is a 512-byte block.

In standard capacity cards the argument is the byte address of the start of a data block and the response depends on the data block length. If the data block length is set to 512 bytes to be the same as a high capacity card, then the data block number can be converted to the byte address by multiplying by 512 (or left shifting by 9). The calculations that the file system software will do find the block number of data on the SD Card. Using the conversion between block number and byte address can allow standard capacity cards to be used with the file system software.

**Using the Serial Port to Enter Integer Numbers**

In this experiment, we will need to enter an unsigned long (uint32) through the serial port. The values that are received by the serial port are ASCII values. A software routine is needed to determine which received values are numeric ASCII characters and convert them into an integer. A number is entered as a string of ASCII characters such as: 123 which are 0x31, 0x32 and 0x33 in ASCII. The following paragraphs discuss some different options for converting a series of numeric characters into an integer value.

The most direct method is to keep a running total of numeric values until the enter key is pressed. In this method the running total is set to zero until a numeric value is received. Any non-numeric characters can be ignored. Numeric values range from 0x30 (48 decimal) for a 0 to 0x39 (57 decimal) for a 9. The upper nibble is masked off leaving a value from 0 to 9. The running total is multiplied by 10 and the new value is added to it. This continues until the enter key is pressed which would send either a linefeed (LF, ASCII 0x0A) or a carriage return (CR, ASCII 0x0D) from the terminal to the UART. At that point, the running total is saved as an unsigned long (uint32) and then the running total is cleared to zero.

The source code that is given has another method that uses the standard library (stdlib.h) function ***long** atol (**const char ***str)*. This function converts a string of numeric characters into an unsigned long (uint32_t). The string should be terminated by a null (0x00) or some other non-numeric ASCII value. If there are no numeric values in the string, then the function returns zero. A character string is an array of byte values where the last byte is a null (0x00). The name of the array is also the pointer to the string which is used as the parameter sent to this function. This function can be used by creating an array of characters and filling the array with entered numeric values. When a non-numeric value is received, it is ignored. When a linefeed (LF, 0x0A) or a carriage return

(CR, 0x0D) is received, the function is called with the name of the array as the input parameter. Note that a negative sign (-, 0x2D) and the decimal point (., 0x2E) are both non-numeric values and are ignored, so only positive integers can be entered. The programmer could add statements to detect these values, but they are not needed for this project. An additional feature that is nice to have is the ability to back space when an error is made. When backspace is pressed on the keyboard, Putty will send either a delete (DEL, 0x7F) or a backspace (BS, 0x08). The function should respond by moving back one element in the array and overwrite that value with a null (0x00).

**Procedure**

The following equipment will be required for this part of the experiment:

- Kit of parts with SD card, SD card socket and adapter
- Breadboard or circuit board with basic microcontroller circuit from Exp. #1
- 9V battery or external power supply
- Serial Cable
- Long_Serial_In.c
- Long_Serial_In.h
- UART and LCD files from Exp#1 solution
- PC with
    - Keil uVision2 (non-code limited version)
    - F.L.I.P.
    - Terminal Program (Putty)

1) Add the SD card socket to the basic microcontroller circuit using the schematic included with this handout. The AUXR register must be initialized to access all of the 1K of on-chip XRAM. Bits 2 and 3 (XRS0 and XRS1) of the AUXR register control how much of the on-chip XRAM can be accessed (see the chart given below. Setting both of these to '1' enables access to all of the on-chip XRAM. All of the other bits of this register can be set to '0.' Note that if bit 1 (EXTRAM) is set to a '1' then the on-chip XRAM is disabled.

AUXR - Auxiliary Register (8Eh)

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| DPU | - | M0 | - | XRS1 | XRS0 | EXTRAM | AO |

**XRAM Size**

| XRS1 | XRS0 | XRAM size |
|---|---|---|
| 0 | 0 | 256 Bytes (default) |
| 0 | 1 | 512 Bytes |
| 1 | 0 | 768 Bytes |
| 1 | 1 | 1024 Bytes |

The SPI functions are part of the hardware layer and should be in separate source code file than the SD Card functions.

2) (3pts) Write an initialization function for the SPI peripheral. It should set the clock to a frequency less than or equal to the value specified by the clock_rate input parameter. The CPOL and CPHA bits should be set to '0.' The SPI peripheral should also be enabled and set to operate in the master mode. The slave select pin should be disabled to prevent mode fault errors. A possible prototype for this function is:

   *uint8_t SPI_Master_Init(uint32_t clock_rate);*

3) (3pts) Write an SPI transfer function. This function should have an input parameter of the byte that is to be sent out. The received value can be returned using a pointer sent by the calling function or as the return value. The programmer may wish to do some error checking and return error flags in addition to the received byte. I chose to return the error flags and to use a pointer from the calling function for the return value which gives this possible prototype:

   *uint8_t SPI_Transfer(uint8_t send_value, uint8_t *received_value);*

   An SPI transfer is started by writing the byte to be sent to SPDAT. The function should wait until the SPIF flag in SPSTA is set and then read the received byte from SPDAT. It is good programming practice to put a timeout in the wait loop just in case SPIF is never set. This is an example of where returning an error flag in addition to the received byte would show that a byte was never received because a timeout error occurred. Other error flags can be found in SPSTA and their descriptions can be found in the AT89C51RC2 datasheet.

The SD Card functions are part of the hardware application layer. The hardware application layer should call standard functions to communicate with the hardware layer. This allows the hardware layer functions to be changed to port to a new microcontroller and the hardware application layer remains the same.

4) (6pts) Write a function that allows you to send a command to the SD card. A command consists of six bytes. The six-bit command with the start bit ('0') and transmission bit ('1') in the two most significant bits is the first byte. This is followed by a 32-bit (4-byte) argument sent most significant byte first. The final byte is a checksum or zeroes with the end bit ('1') as the least significant bit. The CMD0 and CMD8 commands require a checksum while all other commands just need the end bit with seven leading zeroes (0x01). The recommended function prototype for this function is:

   *uint8_t send_command (uint8_t command, uint32_t argument);*

   The function should perform the following tasks:

   a. Check to see if the command is only 6 bits (63 or less). If not, an error flag should be set to indicate an illegal command value and the function should exit.

b. The command should be OR'ed with 0x40 to append the start and transmission bits to the first byte to send.
c. Send the first byte using the SPI transfer function described in an earlier step. An error flag could be returned if a timeout or an SPI error occurs.
d. The 32-bit argument should be sent next starting with the MSB. The programmer may wish to check for errors and stop the transmission if an error occurs.
e. The checksum byte with the lsb set to '1' for the end bit is sent last. If the command is 0 or 8, then a checksum must be sent, otherwise 0x01 can be sent.
f. The return value is the error status.

This function can be checked for proper byte transmission order using the debugger (simulator), by examining the values written to the SPI port. An oscilloscope or logic analyzer can be used to check the output on the SCK and MOSI pins. This is not required, but it can help with debugging later on in this experiment.

5) (5pts) Write another function that can receive the response from the SD card after a command is sent. The response that will be returned depends on the command. The most common response is the one byte R1 response. Other possible responses may have up to four additional bytes. The response may have a short delay so the response function should repeatedly send 0xFF to the SD card and read the received byte until the msb of the received byte is '0.' A timeout should be used so that this does not become an infinite loop. The recommended approach is to have a function that is passed the expected number of bytes of the response. Keep in mind that if an error occurs, the only response will be the one byte R1 response that indicates the error condition. An additional 0xFF byte should be sent after the response has been received. The response can be returned to the main function using an unsigned char array that is passed to the function as a pointer. This would make the function prototype look like this:

*uint8_t receive_response (uint8_t number_of_bytes, uint8_t * array_name);*

The function should perform the following tasks:

a. The microcontroller will repeatedly send 0xFF out of the SPI port and read the value that is received using the SPI transfer function. This repetition should continue until the msb of the received byte is '0' or until a timeout occurs. A byte with the msb as '0' is the R1 response and it should be stored in the array. If the R1 response is 0x01 (idle state) or 0x00 (active state), then no errors have occurred. If an error or a timeout occurs, then an error flag should be set and the function should send the final 0xFF and exit.
b. If more than one byte is expected, then a 0xFF should be sent out of the SPI port and each received byte should be stored in the array. This step should be repeated until all of the expected bytes have been received.
c. The function should end with one addition 0xFF being sent out of the SPI

port. The received value does not matter.

d. The return value is the error status.

I recommend that the SD card initialization should not be all written at debugged at once. Start with the power on (74 clock cycles with nCS=1) step and the CMD0 command and response first. Once this is working, move on to the next step and make sure it is working. It will be easier to debug a small section of code and correct a mistake early than to repeat the mistake over and over again and have to correct it in multiple locations. There are some images captured by the logic analyzer given in the "Debugging SD Card Communication" section later in this document that show what some of the commands and responses should look like.

6) (16pts) The *send_command* and *receive_response* functions can be used by an SD card initialization function. The first steps in initializing the SD card are to send it at least 74 clock pulses on SCK with the nCS signal set high. Then switch the nCS signal low and send the SD card CMD0. The response for this command is the one byte R1 response. If there are no errors, then the response should be 0x01 which indicates the SD card is in idle mode. The programmer may wish to print the command and response for each step taken during the SD card initialization for debugging purposes. For this step the output may look like:

    Initializing SD card….
    CMD0 sent…. Response is 0x01

   It is recommended that the steps given in the initialization flow chart in section 7.2.1 Mode Selection and Initialization of the SD Specifications document be written inside "if statements." The condition on each "if statement" is that if any errors have occurred, then that step is not executed. This is one way that the initialization can stop if an error occurs. The programmer may also want to place debugging outputs, as shown above, in each step to help find an error when it occurs. The prototype for the SD card initialization function is:

    *uint8_t SD_Card_Init (void);*

   The initialization function should perform the following tasks:

   a. Set nCS=1 and send at least 74 clock cycles on SCK.
   b. Clear nCS=0 and send CMD0. Read the R1 response and only continue if it is 0x01. Set nCS=1 after the response is received.
   c. Clear nCS=0 and send CMD8. Read the R7 response. Set nCS=1 after the response is received. If the R1 response is 0x05 (illegal command) then the SD card is a v1.x type. It is up to the programmer how they want to handle this. The function can designate this card as unusable or the card type can be stored so that the high capacity support bit is not set when ACMD41 is sent. Otherwise, if the R1 is not 0x01, then the initialization function should exit with an error flag. The R7 response should also be checked to make sure the voltage range is correct and the check value has been echoed correctly. If the voltage range does not match, then designate

the card as unusable.

d. Clear nCS=0 and send CMD58. Read the R3 response. Set nCS=1 after the response is received. If the R1 response is not 0x01, then the initialization function should exit with an error flag. The R3 response should also be checked for voltage compatibility with your system and designate the card as unusable if it is not.
e. Clear nCS=0 and send ACMD41. An application specific command (ACMD) is sent by sending CMD55 first and receiving the R1 response. Then ACMD41 is sent as CMD41 and the R1 response is received. The argument for ACMD41 depends on whether this is a v1.x or v2.0 SD card. If it is a v1.x SD card, then the argument is 0. If it is a v2.0 SD card, then the HCS bit (bit 30) should be set. This command places the SD card into its active state, but this may take a little while. This command sequence should be repeated until the R1 response indicates that the card is active (0x00) or that an error has occurred. A timeout should also be used to exit the loop just in case the card fails to go to its active state. The nCS can be set to '0' for the entire loop or it can be set to '0' for CMD55 and ACMD41 and their responses. Note that it must be '0' for both commands together. Exit the initialization function if any errors occur.
f. If the SD card is a v2.0 card, then CMD58 should be sent again to re-examine the OCR register in the R3 response. The msb of the OCR (bit 31) is the power up status bit. It must be set to '1' in order for the card capacity status (CCS, bit 30) bit to be valid. If both bits are set, then the SD card is a high capacity (SDHC) card. If only the power up status bit is set, then the SD card is a standard capacity card. The SD cards given out in the kits should all be SDHC cards, but the programmer may wish to add support for standard capacity. For ease of compatibility between the two types of cards, the block size on the standard capacity should be set to 512 bytes using CMD16. The block address used for a SDHC card would need to be multiplied by 512 (block size) to convert it to the byte address expected by the standard capacity card. A global variable would be needed to keep track of the SD card type for later accesses.
g. The return value is the error status.

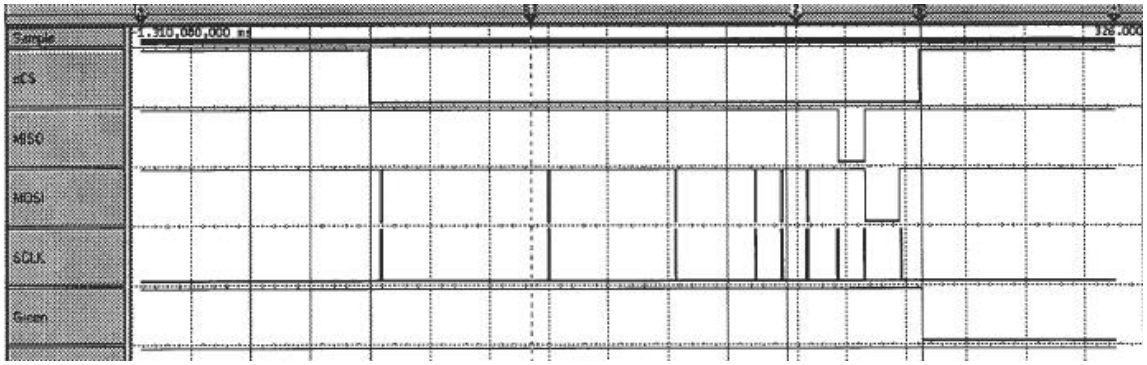
7) (2pts) Create the main function for this experiment. It should perform the necessary steps for initialization first. This includes AUXR, UART, SPI and SD card. The SPI should be initialized to a clock rate of 400KHz or less for the SD card initialization. My approach during the initialization was to switch on the red LED, print a message and halt in an infinite loop if any errors occurred. This helps identify where the problem is at. After the SD card initialization is complete, the SPI clock rate can be increased up to 25MHz. Note that the fastest clock rate our system can output is one half of the oscillator frequency. After all parts of the system are initialized, the program will enter a "super loop" or *while(1)* loop. At this point, this loop does not have anything in it, so the program just stops here.

**Debugging the SD Card Communication**

The following figures are waveforms captured by the logic analyzer and may help you in determining what to look for if your system is not working. Most of these waveforms were captured with the sample clock rate set to 50ns between samples. It is recommended to use an I/O pin, such as an LED output, going low right before the send command function or right after the receive response function as a trigger to capture the signals. This trigger signal can only go low once during the program, so that it identifies the SD card communication you are trying to see. On the logic analyzers, the sample rate and trigger position can be adjusted to capture more or less detail in a command and response.

If you are using a mixed signal oscilloscope, set the trigger mode to normal triggering so that it captures and holds the waveforms. The sample rate and trigger position cannot be adjusted. The memory depth is large enough that signals can be expanded enough to see the detail of the communication. The trigger position is always in the middle of the captured waveforms. If needed, the trigger signal could go low after the send command and before the receive response functions to center the waveforms. You may need to connect a wire from the microcontroller reset pin (pin 9) to Vcc to create a manual reset. This creates a "clean" startup that allows edge triggered signals to be captured more reliably.

Note that printing to the serial port takes a long time relative to the signals being captured. If you are printing commands and responses, then make sure the print statements are outside of the actual send command and receive response functions and the trigger used to capture these waveforms.

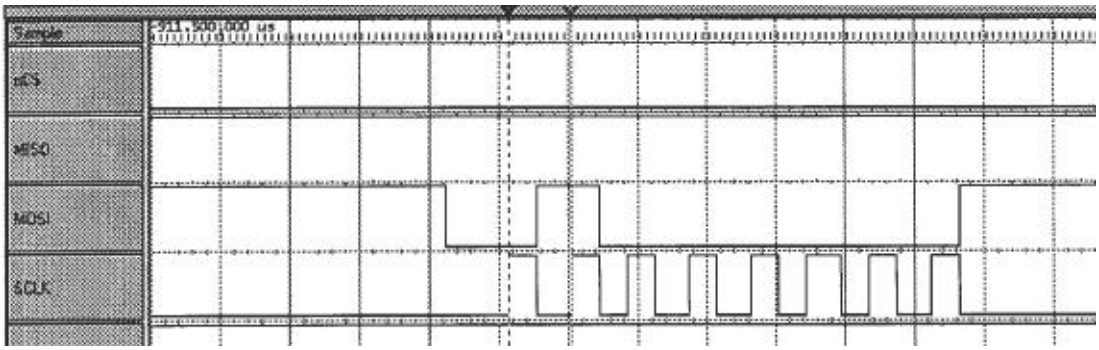



**Complete CMD0 and R1 response**

The first figure shows a complete CMD0 and R1 response. In this figure you will see nine bursts of activity on SCLK which represent nine SPI transfers. The first six are the six bytes of CMD0. It takes two SPI transfers to get the R1 response and the final transfer is just to put the SD card into standby mode.

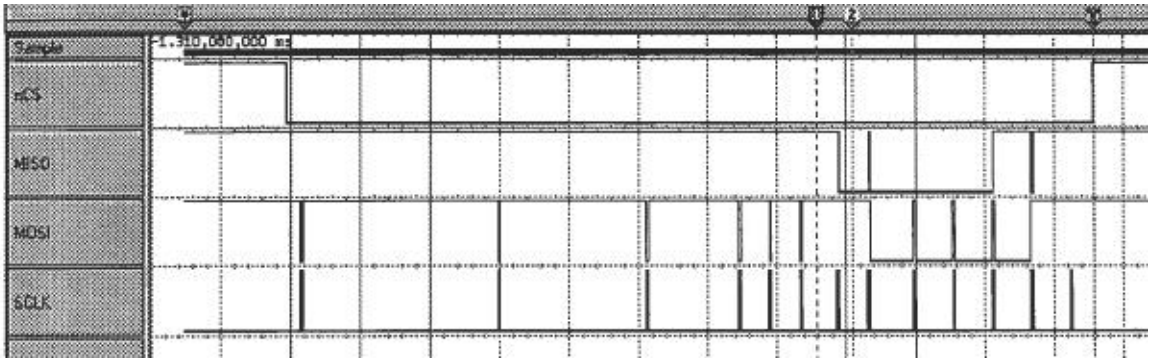
When viewing the SD card communications, verify that the values sent by the send command function are the expected values. These will be on the MOSI pin

and the sent bit value is read when SCLK transitions from low to high. Examine the received values on the MISO pin. A common mistake is to use a pointer instead of declaring an array to store the received values. If you print the hexadecimal received values and they do not match what was seen on the MISO pin, it is possible that the values were not stored properly in memory. An array reserves locations in memory for these values, a pointer does not.



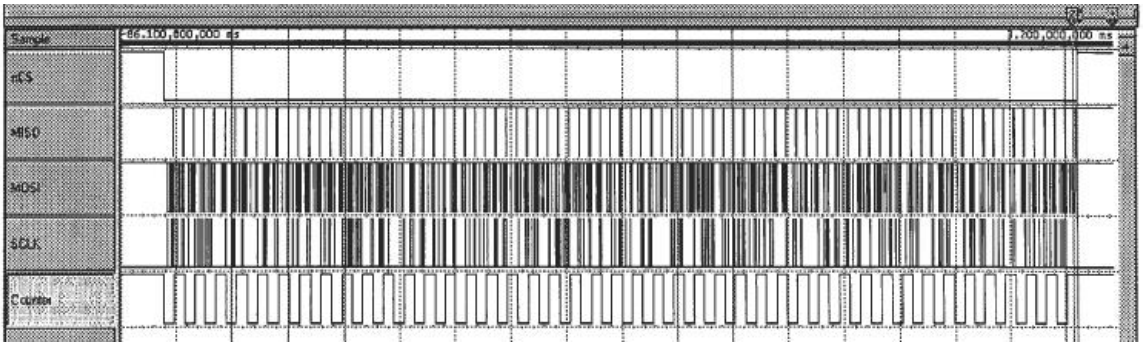

**Detail of the first byte of CMD0**

The next figure is from the previous captured waveform but it is zoomed in on the first byte of CMD0 to show you how you can use this to check each byte that is sent and received during a command and response. The bits are sampled by the receiving device on the rising edge of SCLK. The cursors were placed to show the first two bits are '0' and '1' for the Start and Transmission bits. The other six bits are '0' for CMD0.




**Complete CMD8 and R7 response**

This figure shows CMD8 being sent and the R7 response being received. Again the first six bytes are CMD8. You may be wondering why the first few bytes are more widely spaced than the later bytes. This shows how much processing time it takes to shift the 32-bit argument to get the next byte to send. As you can see, this extra little bit of processing time does not affect the transmission. The first cursor marks where CMD8 ends and the second cursor marks where the response begins. Again there is one SPI transfer before the response begins and there is one SPI transfer after the response is received.




**ACMD41 loop until SD card is active**

In the final figure I increased the time in between samples on the logic analyzer until I could capture the entire do-while loop used in sending ACMD41 to put the SD card into active mode. The sample rate is now too slow to see any detail of a single SPI transfer, but I can see how long it takes to place the SD card into active mode. If I had left the sample rate the same as for the previous figures then I could have seen the detail of one ACMD41 transmission. For this figure, I created a signal that I labeled Counter. It is an I/O pin that I set low just before CMD55 was sent and then set back high immediately after the R1 response was received for that command. This allowed me to see about how many times ACMD41 had to be sent before the card switched to active mode. A count of the pulses shows that in this case it took about 44 times before the card became active. This shows you that an 8-bit timeout is probably sufficient, but make sure that you give the SD card adequate time to switch to active mode. Some of the SanDisk SD cards required more than a 16-bit timeout. Also note the chip select (nCS) is low (active) during the entire time.

After the SD card is initialized, then data can be read from it using the function described in the next step.

8) (10pts) Write another function that can be used to read a data block of specified size and store the data into an array. The details of a data block response are given in this handout. The function will be passed the number of bytes expected in the response and the address of the array where the data is to be stored. The function may also return a value that indicates if the read was successful or if an error occurred. The function prototype could look like this:

*uint8_t read_block (uint16_t number_of_bytes, uint8_t * array);*

The function should perform the following tasks:

a. The SD card will first respond with the R1 response. Therefore, just like the *receive_response* function, the microcontroller should send 0xFF out through the SPI port until the R1 response is received. If the R1 response is not 0x00 or if a timeout occurs, then the function should exit with an error value returned.
b. After another delay, the SD card will send a data start token which is 0xFE or a data error token will be sent. The data error token has the upper four

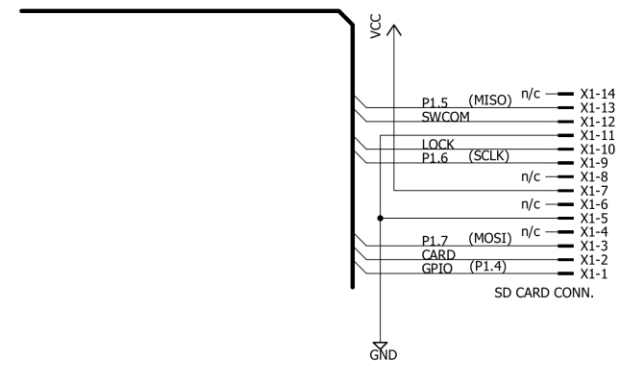
bits cleared to '0' and the lower four bits indicate what error occurred. The microcontroller should send 0xFF out through the SPI port until either value is received. If the data error token is received then the function should exit with an error value returned. If the data start token is received, then the next value received is the first byte of the data block to be received.

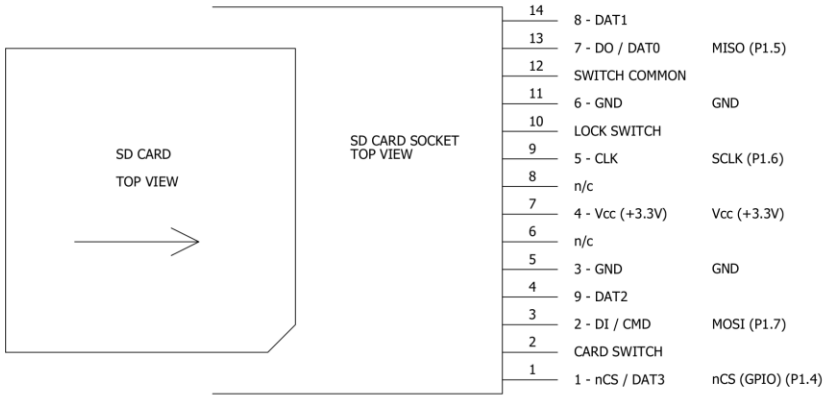
c. The microcontroller should continue to send 0xFF out through the SPI port for each byte of data to be received. Each byte of data can then be read from the SPI port and placed in the array.
d. After all of the expected data bytes have been received from the SD card, the microcontroller should send three more 0xFF values out through the SPI port. The first two are to receive a 16-bit checksum of the received data. The programmer can calculate and compare the checksum or just discard these values. The final 0xFF is to allow the SD card to return to its standby state.
e. The return value is the error status.

9) A function is given with the download files for this experiment that allows the user to enter an unsigned long (uint32_t) through the serial port (UART) as described in the background material. The function does not have an input parameter and returns an unsigned long (uint32_t). The value is entered using Putty and typing numbers on the keyboard. The *UART_Receive()* function from the experiment #1 solution is used to input an ASCII character through the UART. Any non-numeric characters are ignored and not echoed. When the enter key is pressed the string of ASCII characters is converted into an unsigned long (uint32_t) and returned.

10) (4pts) Modify the main function by adding to the "super loop" or while(1) loop. This loop should prompt the user to enter a block number to read from the SD card. The number is entered using the long serial input function given with the project files. After the number is entered, the send command function can be used to send the read block command (CMD17) and the *read_block* function will be used to read the 512 bytes of one SD card data block. The nCS pin on the SD card should be low during the send command and read block functions. The *print_memory* function from experiment #1 should then be used to display the block contents on Putty. The loop repeats by asking the user to enter another block number. The blocks that are displayed are the SD card information blocks and the data stored on the SD card. This information will be used in the fourth experiment to access the files stored on the SD card. You can verify that you are reading block 0 correctly if the block is mostly 0's except for line 0x01C0. Also, the bytes at addresses 0x01FE and 0x01FF should be 0x55 and 0xAA, respectfully. It is possible that there are other bytes that are non-zero, but typically these are the only non-zero bytes.

11) Demonstrate the results of this experiment to the instructor. The SD Card should be initialized and demonstrate that blocks of data be read from the card.




| | SD CARD PIN | SPI CONNECTION |
|---|---|---|
| 14 | 8 - DAT1 | |
| 13 | 7 - DO / DAT0 | MISO (P1.5) |
| 12 | SWITCH COMMON | |
| 11 | 6 - GND | GND |
| 10 | LOCK SWITCH | |
| 9 | 5 - CLK | SCLK (P1.6) |
| 8 | n/c | |
| 7 | 4 - Vcc (+3.3V) | Vcc (+3.3V) |
| 6 | n/c | |
| 5 | 3 - GND | GND |
| 4 | 9 - DAT2 | |
| 3 | 2 - DI / CMD | MOSI (P1.7) |
| 2 | CARD SWITCH | |
| 1 | 1 - nCS / DAT3 | nCS (GPIO) (P1.4) |

| | |
|---|---|
| TITLE: SD_Card_Socket | |
| Document Number: | REV: |
| Date: not saved! | Sheet: 1/1 |

**Grading:**

| | | |
|---|---|---|
| Functionality: | Code works as described in the assignment: Registers set as needed and actions occur as specified; Questions answered and calculations shown. | Points awarded as specified in the steps marked graded. (50 points) |
| Compile Errors: | If submitted code does not compile, then a minimum of 25% of the functionality points will be deducted. More may be deducted for multiple errors. | Minimum of 25% of functionality point deducted. |
| Organization: | Seperating device drivers (microcontroller, SPI), hardware drivers (SD Card) and application code. | 10 points |
| Readability: | Source code is well commented. Descriptive names are used for functions, constants and variables. | 10 points |
| Correctness | No patches or work-arounds are used in the source code to get the correct functionality. | 10 points |