Lab 1 Report

EE 4341

Spring 2016

Cole Nielsen niels538@umn.edu
Jacob Voges: voges008@umn.edu

Abstract

A PIC32 USB STARTER KIT II was used with a I/O expander board and a PICTAIL SD Card board in order to connect to a SD card. The functions of a given SD CARD library were rewritten to work with the PIC32 used via SPI. A main file was then written to write and read back a block of data to and from the SD card using the library functions. The data was then verified by displaying the first three bits of the first 8 bytes of the receiced block on the Starter kit board's LEDs. The written code proved non functional when tested, however, despite debugging efforts, due to issues in the SD card initialization.

1 Introduction

This lab focused on using a PIC32 microcontroller to store and read data from a SD card. Storing data to and from a SD card/mass storage device is of significant value for embedded systems as the internal memory of microcontrollers is typically small, which can be a limitation unless external memory is used. The SD (Secure-Digital) card is a industry standard non-volatile storage format that is built in a small form factor. It utilizes either SPI or a 4-bit parallel data port for communication. For this lab SPI was used as the communication method to the SD card using a PIC32 hardware peripheral. SPI-based communication to the SD card was implemented in code by writing a library encompasing all functions needed to initialize the SD card and perform data transfers with the card. A main file was then writen using this library to initialize the card, perform a block [512 Byte] write to the card, and then read the written block back and display the data read on the PIC32 Starter Kit's LEDs. The process of writing these codes will be elaborated in the following section.

2 Implementation

Implementation for this lab was performed in two parts, the first being a library encapsulating functions for initialization, read/writing and sending commands to the SD Card, and second a main file that was used to test the written library.

SD Communicaiton Library

This library was based on the SD communication library for an Arduino-like board utilizing the PIC 32 provided for this lab. The original library depended on function calls to other libraries implementing Arduino sketch like functions, so they were rewritten to achieve the same function by operating directly on the PIC32's registers. Not all the library functions are integral, so only the important ones used will be covered, being initSD(), which initializes the PIC for serial communication to the card, initMedia(), which initializes the SD card, writeSPI(), which sends and receives data via SPI and readSECTOR() and writeSECTOR() which are the read and write functions for the SD card memory.

The initSD() function is the first function that needs to be understood. It is a void initialization function that configures the PIC32's SPI2 hardware peripheral to transmit data to and from the card. This is done by configuring the I/O direction for the SDI, SDO, SCK pins for the SPI2 peripheral, and then configuring GPIO pin RB9 to act as a chipselect (CS) output, which is active low. SPI2CON is set such that it the hardware is on, CKE (clock edge) is set from active to idle transition, CKP (clock polarity) is active high and the signal is sampled in the middle of the data pulse. These are in accordance to the SD card standard's usage of the SPI bus . The baud rate generator is also set for 71, which corresponds to 250kHz, which is within the 100kHz to 400kHz range recommended for initial commands to the SD card.

```
void initSD(void){
    PORTBbits.RB9 = 1; //set CS high
    TRISBbits.TRISB9 = 0; //CS output

    TRISGbits.TRISG6 = 0; //SCK output, master mode clock is output
    TRISGbits.TRISG7 = 1; //SDI input
    TRISGbits.TRISG8 = 0; //sdo output

   // init the SPI module for a slow (safe) clock speed first
    SPI2CON = 0x8120; //initialize spi2 control reg
    SPI2BRG = 71;
}   // initSD
```

The next function is writeSPI(), which is used for sending and receiving data via SPI bus. It takes a byte of data (char) that is to be transmitted, and returns a byte if data is sent back to the PIC. It works by writting the byte to be transmitted to the SPI buffer register, which is configured to automatically transmit when a word is loaded into it. The while statement then waits until the transmission is complete

```
unsigned char writeSPI(unsigned char b){
   SPI1BUF = b;                // write to buffer for TX
   while(!SPI1STATbits.SPITBF);  // wait transfer complete
   return SPI1BUF;             // read the received value
}  // writeSPI
```

The initMedia() function is used to initialize the SD card so it can be used by the PIC. The initialization is done in a defined sequence as described by in the SD card standard specification document. First, the SD card is disabled, with the disableSD() function, which just pulls CS high. The clock pin is then clocked more than 74 times, which is understood by the SD card as a indication to begin accepting commands. This is achieved by repeatedly using a function clockSPI(), which calls invokes writeSPI() to send the byte 0xFF, which takes 8 cycles to perform. Since clockSPI() is invoked 10 times by the loop, 8*10 = 80 clock pulses should be sent to the SD card, causing it to begin accepting commands. The card is then enabled with enableSD(), which simply pulls CS low (active). A command is then send using the sendSDSSCmd function to reset (CMD0) the device (the SD card commands are defined in the .h file), the returned value from the SD card is stored and the card is again

disabled. The returned value, indicating whther the command was rejected is verified, and in the case of a rejection a value is returned to indicate to the calling function that initialization failed. In the for loop, the SD card command CMD1 (init) repeatedly until it either reaches a predefined timeout value, which is greater than it should take for a response, or until the card exits the idle state and is ready to read/write. If this fails, a value indicating that is again returned by the function. Finally, the SPI2 peripheral is reconfigured at a higher clock rate after initialization, to yield higher data transfer rates. The clock rate is initially slower to ensure proper/safe initialization of the SD card.

```
int initMedia(void){
   int i, r;

   disableSD(); // card starts inactive

   //clock card atleast 74 times
   // The card will enter its native operating mode and go ready to accept native
       commands.
   for (i=0; i<10; i++)
      clockSPI();

   enableSD(); //select card

   // reset card
   r = sendSDCmd(RESET, 0);
   disableSD();
   if (r != 1){      // must return Idle
      return E_COMMAND_ACK; // comand rejected

    }
   // 5. send repeatedly INIT until Idle terminates
   for (i=0; i<I_TIMEOUT; i++) {
      r = sendSDCmd(INIT, 0);
      disableSD();
      if (!r)
         break;
   }
   if (i == I_TIMEOUT)
      return E_INIT_TIMEOUT; // init timed out

   // 6. increase speed
    SPI2CON = 0; // disable the SPI2 module
    SPI2BRG = 0; // Fpb/(2*(0+1))= 36/2 = 18 MHz
    SPI2CON = 0x8120; // re-enable the SPI2 module
    return 0;

}
```

The writeSECTOR() function reads a block (512 bytes) of data in an array given by *p to the address given by argument a. This is done first by sending a command with the address to be read from and the single block read command to the SD card. If the card responds without timing out, each of the 512 elements of the data array are read using the readSPI() funtion, which are then stored into sequential addresses given by pointer p. The CRC for the received data is then taken from the card and ignored. The CRC can be used to verify if the data is not corrupt. The card is then again disabled until further operations.

```
int readSECTOR(LBA a, char *p){
   int r, i;

   r = sendSDCmd(READ_SINGLE, (a << 9));
   if (r == 0)   // check if command was accepted
   {
   // 2. wait for a response
   for(i=0; i<R_TIMEOUT; i++){
   r = readSPI();
   if (r == DATA_START)
   break;
   }

   // 3. if it did not timeout, read 512 byte of data
   if (i != R_TIMEOUT)
   {
      i = 512;
      do{
         *p++ = readSPI();
      } while (--i>0);

      // 4. ignore CRC
      readSPI();
      readSPI();

   } // data arrived

   } // command accepted

   // 5. remember to disable the card
   disableSD();

   return (r == DATA_START); // return TRUE if successful
}
```

The final function of notice is the writeSECTOR() function, which takes an array given by p and writes its contents to the block starting at address a in the SD card. This is done by sending a single write block command with the desired address to the SD card. Then using

a for loop and the writeSPI() function, the 512 elements of array p are sent via SPI to the SD card. The returned value from the SD card is then checked after the block transmission to see if the data was accepted, and a corresponding value is returned from the function to indicate so to the calling function. Finally, the card is again disabled.

```
int writeSECTOR(LBA a, char *p){
   unsigned r, i;

   // 1. send WRITE command
   r = sendSDCmd(WRITE_SINGLE, (a << 9));
   if (r == 0)   // check if command was accepted
   {
      // 2. send data
      writeSPI(DATA_START);

      // send 512 bytes of data
      for(i=0; i<512; i++)
      writeSPI(*p++);

      // 3. send dummy CRC
      clockSPI();
      clockSPI();

      // 4. check if data accepted
      r = readSPI();
      if ((r & 0xf) == DATA_ACCEPT)
      {
         // 5. wait for write completion
         for(i=0; i<W_TIMEOUT; i++)
         {
            r = readSPI();
            if (r != 0 )
               break;
         }
      } // accepted
      else
      {
         r = 0;
      }
   } // command accepted

   // 6. disable the card
   disableSD();

   return (r);     // return TRUE if successful
} // writeSECTOR
```

The main file was written to perform a simple test procedure for the SD card SPI library.

This is performed by initializing the PIC, SD card, read and writing a block, and then displaying the received block for user verification. This was done in code first by creating two arrays 512 bytes in length, one for saving values to be transmitted (data) and the other to store recieved blocks (buffer). The LEDs on PORTD are also initialized right away as outputs to be ready for displaying the read back data. Next, the initSD() and initMedia() library functions are called to intialize the PIC and the SD card for usage. Next, data array is filled with values corresponding their indice values. Then, the writeSECTOR() function is used to write the data array at address 0, and the readSECTOR() function is called to read it back and store it to array buffer. Finally, in an infinite loop, the first 7 elements of the read back array are cycled through and displayed on PORTD (the three LEDs) with a delay between displayed values. If the code works properly, the LEDs should cycle through 0-7 in binary.

Main File

```
#define B_SIZE 512 // data block size

#define START_ADDRESS 0 // start block address
#define N_BLOCKS 1 // number of blocks

char data[ B_SIZE];
char buffer[ B_SIZE];

#include <stdio.h>
//#include <plib.h>
#include <stdlib.h>
#include <p32xxxx.h>
#include <sdmmc.h>
/*
 *
 */
main()
{
    TRISDbits.TRISD0=0;
    TRISDbits.TRISD1=0;
    TRISDbits.TRISD2=0;
    TRISDbits.TRISD6 = 1;
    CNPUEbits.CNPUE6 = 1;
    PORTD = 0;

    initSD();
    initMedia();


    LBA addr;
    int i,j;
```

```
    i = j = 0;
    for( i=0; i<B_SIZE; i++)
        data[i]= i;

    addr = START_ADDRESS;

    writeSECTOR(addr, data);
    readSECTOR( addr, buffer);

    while(1){ //display first 3 bits of first 8 bytes
        for(j = 0; j<1000000; j++); //delay between displaying bytes
        PORTD = buffer[i]; //display ith element of block

        if(i = 7) i = 0; //loop through first 8 elements
        else i++;
    }
}
```

3 Summary

A complete code was written for SD card communication with the PIC in this lab, including a complete library for SPI based communication and a main file for verification of the library. However, when tested with hardware, the written code proved to be non-functional. The issue seems to be in the initialization of the SD card. It was determined that the PIC itself was working properly, as it was observed on the oscilloscope that it was transmitting over SPI exactly as configured, however no data or any response was observed from the SD cards following actions by the PIC. This is where the assumption of failed SD initialization was drawn from. Attempts were made to debug the issue with SD card initialization by adding breaks in the card initilization function by using one of the push buttons as a wait (code gets stopped at points in code where there is a while polling the button, and when the button is pressed the code proceeds) to see on the oscilloscope how the intializtion occurs in the hardware. It was thought this might expose a flaw in the initialization or the configuration of the PIC SPI. Time ran out however before this issue was fully diagnosed, so we concluded the lab with the code in its present state, not fully functional.