Kenneth Lee (kl22943)
Dalton Altstaetter (dea528)
EE 445M
4/10/15

Lab 5 Report

A) Objectives (1/2 page maximum)

The goals for this lab were to interface a micro SD card to the TM4C to create a file system. This file system is useful for real-time debugging for use as a dump or as an alternative to printing to the LCD which is a more intrusive debugging technique for real-time applications. We built an abstraction layer that maps from a logical address to a physical address in the SD card by using a File Allocation Table (FAT) scheme. We made a file driver that allows for file creation, removal, viewing, writing, output redirection, formatting, and directory listing. We built our FAT in a manner that accounts for block usage and spreads file writes across all available blocks to preserve the integrity and longevity of memory blocks. This can all be access in real-time through the interpreter.

B) Hardware Design (none)

C) Software Design (printout of these software components)

1) Pictures illustrating the file system protocol, showing: free space management; the directory; and file allocation scheme

File System Parameters:	
File System Size:	1 Mebibyte (1048576 bytes) = 4096 blocks
Directory:	1 block = 512 bytes 4000 blocks for files to be stored in 2 bytes required to store block number in FAT
Directory Entry:	Name (8 bytes) StartBlock (2 bytes) EndBlock (2 bytes) 12 bytes/file entry 512/12 = 42 files can be stored
File Allocation Table	1 Entry holds 2 bytes 2 bytes/block * 4000 blocks = 8000 bytes 8000 bytes / 512 bytes per block = 16 blocks for FAT

Block 0	Directory
Block 1	FAT
…	FAT
Block 16	FAT
Block 17	Files
…	Files
Block 4017	Files

Free Space Management

Initial State:

File Allocation Table

	Index	Element
	0	x
StartFree->	1	0
	2	3
	3	4
	4	5
	…	…
EndFree->	4017	0

After Creating a File

	Index	Element
	0	x
StartOfFile->	1	2
	2	3
EndOfFile->	3	0
StartFree->	4	5
	…	…
EndFree->	4017	0

After Deleting File

	Index	Element
	0	x
	1	2
	2	3
EndOfFree->	3	0
StartOfFree->	4	5
	…	…
	4017	1

2) Middle level file system (eFile.c and eFile.h files)

```
// filename ********** eFile.h ****************************
// Middle-level routines to implement a solid-state disk
// Jonathan W. Valvano 3/16/11
// Modified by Kenneth Lee, Dalton Altstaetter 4/9/2015
#include "stdint.h"

#define DIRENTRYSIZE 12
#define DIRSIZE 42
#define DIRECTBLOCK 0
#define FATSIZE 16
#define FATSTART 1
#define FATEND 16
#define FREE 0
#define NAMESIZE 8
#define BLOCKSIZE 512

struct directory {
        char name[NAMESIZE];
        uint16_t startFAT;
        uint16_t endFAT;
};

typedef struct directory DIRECTORY;


//---------- eFile_Init----------------
// Activate the file system, without formating
// Input: none
// Output: 0 if successful and 1 on failure (already initialized)
// since this program initializes the disk, it must run with
//    the disk periodic task operating
int eFile_Init(void); // initialize file system

//---------- eFile_Format----------------
// Erase all files, create blank directory, initialize free space manager
// Input: none
// Output: 0 if successful and 1 on failure (e.g., trouble writing to flash)
int eFile_Format(void); // erase disk, add format
```

```
//---------- eFile_Create----------------
// Create a new, empty file with one allocated block
// Input: file name is an ASCII string up to seven characters
// Output: 0 if successful and 1 on failure (e.g., trouble writing to flash)
int eFile_Create( char name[]);  // create new file, make it empty


//---------- eFile_WOpen----------------
// Open the file, read into RAM last block
// Input: file name is a single ASCII letter
// Output: 0 if successful and 1 on failure (e.g., trouble writing to flash)
uint16_t eFile_WOpen(char name[]);      // open a file for writing


//---------- eFile_WOpenFront----------------
// Open the file, read into RAM last block
// Input: file name is a single ASCII letter
// Output: starting index into the block that matches 'name' in the directory else returns -1
uint16_t eFile_WOpenFront(char name[], uint8_t* buf);

//---------- eFile_Write----------------
// save at end of the open file
// Input: data to be saved
// Output: 0 if successful and 1 on failure (e.g., trouble writing to flash)
int eFile_Write( char data);

//---------- eFile_Close----------------
// Deactivate the file system
// Input: none
// Output: 0 if successful and 1 on failure (not currently open)
int eFile_Close(void);


//---------- eFile_WClose----------------
// close the file, left disk in a state power can be removed
// Input: none
// Output: 0 if successful and 1 on failure (e.g., trouble writing to flash)
int eFile_WClose(void); // close the file for writing
```

```
//---------- eFile_ROpen----------------
// Open the file, read first block into RAM
// Input: file name is a single ASCII letter
// Output: 0 if successful and 1 on failure (e.g., trouble read to flash)
int eFile_ROpen( char name[]);      // open a file for reading

//---------- eFile_ReadNext----------------
// retreive data from open file
// Input: none
// Output: return by reference data
//         0 if successful and 1 on failure (e.g., end of file)
int eFile_ReadNext( char *pt);       // get next byte

//---------- eFile_RClose----------------
// close the reading file
// Input: none
// Output: 0 if successful and 1 on failure (e.g., wasn't open)
int eFile_RClose(void); // close the file for writing

//---------- eFile_Directory----------------
// Display the directory with filenames and sizes
// Input: pointer to a function that outputs ASCII characters to display
// Output: characters returned by reference
//         0 if successful and 1 on failure (e.g., trouble reading from flash)
int eFile_Directory(void(*fp)(unsigned char));

//---------- eFile_Delete----------------
// delete this file
// Input: file name is a single ASCII letter
// Output: 0 if successful and 1 on failure (e.g., trouble writing to flash)
int eFile_Delete( char name[]);  // remove this file

//---------- eFile_RedirectToFile----------------
// open a file for writing
// Input: file name is a single ASCII letter
// stream printf data into file
// Output: 0 if successful and 1 on failure (e.g., trouble read/write to flash)
int eFile_RedirectToFile(char *name);

//---------- eFile_EndRedirectToFile----------------
```

```
// close the previously open file
// redirect printf data back to UART
// Output: 0 if successful and 1 on failure (e.g., wasn't open)
int eFile_EndRedirectToFile(void);

// filename ************** eFile.c ****************************
// Middle-level routines to implement a solid-state disk
// Kenneth Lee, Dalton Altstaetter 4/9/2015
#include "efile.h"
#include "edisk.h"
#include <string.h>

//Globals used to format the file system
DIRECTORY dir[DIRSIZE];
uint16_t FAT[256];
unsigned char FormatBuffer[BLOCKSIZE];

//Globals used by File Writing operations (Create, WOpen, Write, Delete)
unsigned char LastWBlock[BLOCKSIZE];
uint16_t LastWBlockNum;
char* FileWName;
unsigned char tempDir[BLOCKSIZE];
int endOfFileIndex;
uint8_t buf[BLOCKSIZE];

//Globals used by File Reading operations (ROpen, Read)
unsigned char LastRBlock[BLOCKSIZE];
uint16_t LastRBlockNum;
char* FileRName;
unsigned char tempDirRead[BLOCKSIZE];
unsigned char FATReadBuf[BLOCKSIZE];
int ReadingPos=0;


//Globals for redirect flag
int RedirectFlag = 0;
```


Given a FAT index (a.k.a. startBlock/endBlock for a file), divide by 256 to get the FAT Block that the index refers to plus 1. Ex: StartBlock of File A is 2000. 2000/256 = 7.8125.

Therefore the index refers to block 8 in the FAT.

To transform the StartBlock into an index into
a 512 byte array, modulo divide block number by 256. Ex:
2000%256 = 208.

Step 1: BlockNumber/256 + 1 = Block of FAT that contains the corresponding element

Step 2: BlockNumber%256 = byte number within that block that contains the corresponding element

Step 3: BlockNumber + SizeOfFAT = Corresponding Block in File
*/

```
//---------- eFile_Init-----------------
// Activate the file system, without formating
// Input: none
// Output: 0 if successful and 1 on failure (already initialized)
// since this program initializes the disk, it must run with
//    the disk periodic task operating
int eFile_Init(void){
        int status = 0;
        status = eDisk_Init(0);// initialize file system
        return status;
}

//---------- eFile_Format-----------------
// Erase all files, create blank directory, initialize free space manager
// Input: none
// Output: 0 if successful and 1 on failure (e.g., trouble writing to flash)
int eFile_Format(void){

        int i,j,k;
        int status = 0;

        /**********Format the Directory**********/
        strcpy(dir[FREE].name,"FREE");   //First directory entry is "FREE"
        dir[FREE].startFAT = 1; // FAT index corresponding to the start of the file space
        dir[FREE].endFAT = 4000;  // FAT index corresponding to the end of the file space

        //Fill in a blank directory
```

```
for(i = FREE+1; i < DIRSIZE; i++)
{
        strcpy(dir[i].name,"");
        dir[i].startFAT = 0;
        dir[i].endFAT = 0;
}
// Convert array of structures (directory) into an array of 512 bytes
char* ptr;
for(i=0; i<DIRSIZE; i++){
                ptr = &FormatBuffer[i*DIRENTRYSIZE];
                strcpy(ptr,dir[i].name);
                ptr = ptr + 8;
                *ptr++ = dir[i].startFAT>>8;
                *ptr++ = dir[i].startFAT & 0xFF;
                *ptr++ = dir[i].endFAT>>8;
                *ptr = dir[i].endFAT & 0xFF;
}
// Write the directory to Disk
status |= eDisk_WriteBlock(FormatBuffer,DIRECTBLOCK);

/*******************Format the FAT************************/
//1st Block
for(i = FATSTART; i < 256; i++)
{
        FAT[i] = i+1;
}

for(j=0; j<BLOCKSIZE; j++){
        FormatBuffer[j] = FAT[j/2]>>8;
        FormatBuffer[++j] = FAT[j/2] & 0xFF;
}
status |= eDisk_WriteBlock(FormatBuffer,1);

//2-15 blocks
uint16_t temp = 256;
for(k = 2; k < 16; k++)
{
        for(i = 0; i < 256; i++)
        {
                FAT[i] = temp++;
```

```
                }

                for(j=0; j<BLOCKSIZE; j++){
                        FormatBuffer[j] = FAT[j/2]>>8;
                        FormatBuffer[++j] = FAT[j/2] & 0xFF;
                }
                status |= eDisk_WriteBlock(FormatBuffer,k);
        }

        //16th block
        for(i = 0; i < 159; i++)
        {
                        FAT[i] = temp++;
        }
        FAT[160] = 0;
        for(i=161; i<256; i++){
                FAT[i]=0;
        }
        for(j=0; j<BLOCKSIZE; j++){
                FormatBuffer[j] = FAT[j/2]>>8;
                FormatBuffer[++j] = FAT[j/2] & 0xFF;
        }
        status |= eDisk_WriteBlock(FormatBuffer,16);

        return status;
}

//---------- eFile_Create-----------------
// Create a new, empty file with one allocated block
// Input: file name is an ASCII string up to seven characters
// Output: 0 if successful and 1 on failure (e.g., trouble writing to flash)
int eFile_Create( char name[]){  // create new file, make it empty

        int startBlock,i;
        int FATBlock, FATIndex, nextBlock;
        int status = 0;

        status |= eDisk_ReadBlock(tempDir,DIRECTBLOCK);
        //Search the directory for free space for a new file
        for(i = 0; i < BLOCKSIZE; i += DIRENTRYSIZE)
```

```
        {
                if(!strcmp(&tempDir[i], ""))
                {
                        strcpy(&tempDir[i], name);
                        break;
                }
        }
        if(i==BLOCKSIZE) status = 1; //no more room in directory;

        //Free space starts with startBlock
        startBlock = (tempDir[8]<<8) + tempDir[9]&0xFF;

        // startblock of file in directory
        tempDir[i+NAMESIZE] = startBlock >> 8; // store high byte
        tempDir[i+NAMESIZE+1] = startBlock & 0x00FF; // store low byte

        // endblock of file in directory
        tempDir[i+NAMESIZE+2] = startBlock >> 8; // store high byte
        tempDir[i+NAMESIZE+3] = startBlock & 0x00FF; // store low byte

        // Free space management
        FATIndex = startBlock%256*2;                    //Index of FAT that points to
the next block in the free list
        FATBlock = startBlock/256+1;                    //Block # for the FAT holding that
index
        status |= eDisk_ReadBlock(buf,FATBlock);  //Read the FAT block
        nextBlock = (buf[FATIndex]<<8) + buf[FATIndex+1]&0xFF;          //Get the next
Block in the Free List
        buf[FATIndex] = 0;                      //Set the contents of that block equal to null
(the file contains only one block)
        buf[FATIndex+1] = 0;
        status |= eDisk_WriteBlock(buf,FATBlock);
        tempDir[NAMESIZE] = nextBlock >> 8;
        //Update free list in the directory
        tempDir[NAMESIZE+1] = nextBlock & 0xFF;
        status |= eDisk_WriteBlock(tempDir,DIRECTBLOCK);        //write the directory
back to disk

        //Write zeroes to the first block in the new file
        for(i=0; i<BLOCKSIZE; i++){
```

```
            buf[i]=0xFF;
        }
        status |= eDisk_WriteBlock(buf,startBlock+FATSIZE);          //Write the FAT block back to disk
        return status;
}


//---------- eFile_WOpen----------------
// Open the file, read into RAM last block
// Input: file name is a single ASCII letter
// Output: end index into the block that matches 'name' in the directory else returns -1
uint16_t eFile_WOpen(char name[]){      // open a file for writing

        int i;
        uint16_t startBlock, endBlock, status = 0;
        FileWName = name;
        status |= eDisk_ReadBlock(tempDir,DIRECTBLOCK);    //Read in the directory

        // search directory for matching file name
        for(i = 0; i < BLOCKSIZE; i += DIRENTRYSIZE)
        {
            if(!strcmp(name,&tempDir[i]))
            {
                startBlock = i + NAMESIZE;
                break;
            }
        }
        endOfFileIndex = startBlock + 2;            // obtain endblock of file
        endBlock = startBlock + 2;
        endBlock = (tempDir[endBlock]<<8) + tempDir[endBlock+1];
        LastWBlockNum = endBlock+FATSIZE;
        status |= eDisk_ReadBlock(LastWBlock,endBlock+FATSIZE);   //read in the last block in the file

        return status;
}

//---------- eFile_Write----------------
// save at end of the open file
```

```
// Input: data to be saved
// Output: 0 if successful and 1 on failure (e.g., trouble writing to flash)
int eFile_Write( char data){
	int startBlock,i,j;
	int FATBlock, FATIndex, nextBlock;
	int FATPrevBlk, FATPrevIndex;
	int endFreeBlock;
	int status = 0;
	status |= eDisk_ReadBlock(LastWBlock,LastWBlockNum);  //Read in last block of file
	for(j=0; j<512; j++){
		//Search block until end of file char (0xFF) reached
		if(LastWBlock[j]==0xFF){
			LastWBlock[j]=data;    //write the data to the file
			status |= eDisk_WriteBlock(LastWBlock,LastWBlockNum);   //write the
block back to the disk
			break;
		}
	}
	//End of Block reached
	if(j==512){
		status |= eDisk_WriteBlock(LastWBlock,LastWBlockNum);
		startBlock = (tempDir[8]<<8) + tempDir[9]&0xFF;  //start of free space
		endFreeBlock = (tempDir[10]<<8) + tempDir[11]&0xFF;  //end of free space
		if(startBlock==endFreeBlock) return 1;   //no more space left on disk

		FATPrevBlk = (LastWBlockNum-FATSIZE)/256+1;   //previous block written to
		FATPrevIndex = (LastWBlockNum-FATSIZE)%256*2;     //index into FAT for
previous block written to
		FATIndex = startBlock%256*2;                              //Index within FAT
Block that contains the second block of the Free List.
		FATBlock = startBlock/256+1;                        //Block # for corresponding
FAT block
		status |= eDisk_ReadBlock(buf,FATBlock);  //Read the FAT block
		nextBlock = (buf[FATIndex]<<8) + buf[FATIndex+1]&0xFF;                //Get
the next Block in the Free List
		buf[FATIndex] = 0;                              //Set the contents of that block equal
to null (the new last block in the file)
		buf[FATIndex+1] = 0;
		buf[FATPrevIndex] = startBlock>>8;                //The new block of the file is
taken from the beginning of the free list
```

```
                buf[FATPrevIndex+1] = startBlock&0xFF;
                status |= eDisk_WriteBlock(buf,FATBlock);
                tempDir[NAMESIZE] = nextBlock >> 8; //Update directory for free list
                tempDir[NAMESIZE+1] = nextBlock & 0xFF;
                // Modify and re-write directory to disk
                tempDir[endOfFileIndex] = startBlock >> 8; // store high byte
                tempDir[endOfFileIndex+1] = startBlock & 0x00FF; // store low byte
                status |= eDisk_WriteBlock(tempDir,DIRECTBLOCK);

                //Write EOFs to the first block in the new file
                buf[0] = data;
                for(i=1; i<BLOCKSIZE; i++){
                        buf[i]=0xFF;
                }
                status |= eDisk_WriteBlock(buf,startBlock+FATSIZE);
                LastWBlockNum = startBlock+FATSIZE;

        }
        return status;
}

//---------- eFile_Close-----------------
// Deactivate the file system
// Input: none
// Output: 0 if successful and 1 on failure (not currently open)
int eFile_Close(void){

}


//---------- eFile_WClose-----------------
// close the file, left disk in a state power can be removed
// Input: none
// Output: 0 if successful and 1 on failure (e.g., trouble writing to flash)
int eFile_WClose(void){
        int status = 0;
        //writes the current block being modified back to disk
        status |= eDisk_WriteBlock(LastWBlock,LastWBlockNum);
} // close the file for writing
```

```
//---------- eFile_ROpen----------------
// Open the file, read first block into RAM
// Input: file name is a single ASCII letter
// Output: 0 if successful and 1 on failure (e.g., trouble read to flash)
int eFile_ROpen( char name[]){
	int i,status=0;
	uint16_t startBlock, endBlock;
	FileRName = name;
	status |= eDisk_ReadBlock(tempDirRead,DIRECTBLOCK);

	// search directory for matching file name
	for(i = 0; i < BLOCKSIZE; i += DIRENTRYSIZE)
	{
		if(!strcmp(name,&tempDirRead[i]))
		{
			startBlock = i + NAMESIZE;
			break;
		}
	}
	if(i>=BLOCKSIZE) return 1; //file not found
	startBlock = (tempDirRead[startBlock]<<8) + tempDirRead[startBlock+1];
	LastRBlockNum = startBlock+FATSIZE;
	status |= eDisk_ReadBlock(LastRBlock,startBlock+FATSIZE); //read in start block of
file

	return status;
}

//---------- eFile_ReadNext----------------
// retreive data from open file
// Input: none
// Output: return by reference data
//         0 if successful and 1 on failure (e.g., end of file)
int eFile_ReadNext( char *pt){
	int lastBlock,FATIndex,nextBlock,FATBlock,status=0;
	if(LastRBlock[ReadingPos]==0xFF){
		return 1; //if EOF reached return failure
	}else{
		*pt = LastRBlock[ReadingPos++];  //else read the next character in the file and
update current position in block
```

```
        }
        if(ReadingPos==512){
                lastBlock = LastRBlockNum - FATSIZE;    //Look in FAT for the next block to
read in the file
                FATBlock = lastBlock/256+1;
                FATIndex = (lastBlock%256)*2;
                status |= eDisk_ReadBlock(FATReadBuf,FATBlock);
                nextBlock = (FATReadBuf[FATIndex]<<8) +
FATReadBuf[FATIndex+1]&0xFF;            //Get the next Block in the file
                if(nextBlock!=0){
                        status |= eDisk_ReadBlock(LastRBlock,nextBlock+FATSIZE);
                }
                else return 1; //if no more blocks in file, no more reading to be done
                ReadingPos=0;
        }
        return status;
}

//---------- eFile_RClose-----------------
// close the reading file
// Input: none
// Output: 0 if successful and 1 on failure (e.g., wasn't open)
int eFile_RClose(void){
        ReadingPos = 0; //re-initialize current reading position within a block to zero
}
//---------- eFile_Directory-----------------
// Display the directory with filenames and sizes
// Input: pointer to a function that outputs ASCII characters to display
// Output: characters returned by reference
//       0 if successful and 1 on failure (e.g., trouble reading from flash)
int eFile_Directory(void(*fp)(unsigned char)){
                        int i,j,status=0;
                        char name[8];
                        unsigned char startBlockHi,startBlockLo, endBlockHi,endBlockLo;
                        status |= eDisk_ReadBlock(tempDirRead,DIRECTBLOCK);
                        //Iterates through the directory, printing the name of each file
                        for(i = 0; i < BLOCKSIZE; i += DIRENTRYSIZE)
                        {
                                strcpy(name,&tempDirRead[i]);
                                if(!strcmp(name,""))
```

```
                        {
                                continue;
                        }
                        for(j=i; tempDir[j]!=0; j++){
                                (*fp)(tempDir[j]);
                        }
                        (*fp)('\n');
                        (*fp)('\r');
                }
}

//---------- eFile_Delete-----------------
// delete this file
// Input: file name is a single ASCII letter
// Output: 0 if successful and 1 on failure (e.g., trouble writing to flash)
int eFile_Delete( char name[]){
        int i,status=0;
        uint16_t StartOfFileBlck,EndOfFileBlck,EndOfFreeBlk;
        uint16_t FATBlock,FATIndex;

        //Read in directory and obtain last block of the free space linked list
        //We want to add the file being deleted to the end of the free list
        status |= eDisk_ReadBlock(tempDir,DIRECTBLOCK);
        EndOfFreeBlk= (tempDir[NAMESIZE+2]<<8) + tempDir[NAMESIZE+3]&0xFF;  //end
of free space

        //Search through directory and determine startblock and endblock of file
        for(i = 0; i < BLOCKSIZE; i += DIRENTRYSIZE)
        {
                if(!strcmp(name,&tempDirRead[i]))
                {
                        StartOfFileBlck = (tempDir[(i +
NAMESIZE)]<<8)+tempDir[(i+NAMESIZE+1)]&0xFF;
                        EndOfFileBlck = (tempDir[(i +
NAMESIZE+2)]<<8)+tempDir[(i+NAMESIZE+3)]&0xFF;
                        break;
                }
        }
        //Make endblock of free point to front of file
        FATBlock = EndOfFreeBlk/256 + 1;
```

```
        FATIndex = EndOfFreeBlk%256*2;
        status |= eDisk_ReadBlock(buf,FATBlock);
        buf[FATIndex] = EndOfFileBlck>>8;
        buf[FATIndex+1]=EndOfFileBlck&0xFF;
        status |= eDisk_WriteBlock(buf,FATBlock);
        EndOfFreeBlk = EndOfFileBlck;
        //Update endblock of free to point to endblock of file
        tempDir[NAMESIZE+2] = EndOfFreeBlk>>8;
        tempDir[NAMESIZE+3] = EndOfFreeBlk&0xFF;
        //Clear file entry
        tempDir[i]=0;
        tempDir[i+NAMESIZE] = 0;
        tempDir[i+NAMESIZE+1]=0;
        tempDir[i+NAMESIZE+2]=0;
        tempDir[i+NAMESIZE+3]=0;
        status |= eDisk_WriteBlock(tempDir,DIRECTBLOCK);
        return status;
}  // remove this file

//---------- eFile_RedirectToFile-----------------
// open a file for writing
// Input: file name is a single ASCII letter
// stream printf data into file
// Output: 0 if successful and 1 on failure (e.g., trouble read/write to flash)
int eFile_RedirectToFile(char *name){
                RedirectFlag = 1;   //checked by fputc
                return eFile_WOpen(name);

}

//---------- eFile_EndRedirectToFile-----------------
// close the previously open file
// redirect printf data back to UART
// Output: 0 if successful and 1 on failure (e.g., wasn't open)
int eFile_EndRedirectToFile(void){
                RedirectFlag = 0;   //checked by fputc
                return eFile_WClose();

}
```

3) High level software system (the new interpreter commands)

```
// Interpreter.c
// Runs on LM4F120/TM4C123
// Tests the UART0 to implement bidirectional data transfer to and from a
// computer running HyperTerminal.  This time, interrupts and FIFOs
// are used.
// Daniel Valvano
// September 12, 2013
// Modified by Kenneth Lee, Dalton Altstaetter 4/9/2015

/* This example accompanies the book
   "Embedded Systems: Real Time Interfacing to Arm Cortex M Microcontrollers",
   ISBN: 978-1463590154, Jonathan Valvano, copyright (c) 2014
   Program 5.11 Section 5.6, Program 3.10

 Copyright 2014 by Jonathan W. Valvano, valvano@mail.utexas.edu
    You may use, edit, run or distribute this file
    as long as the above copyright notice remains
 THIS SOFTWARE IS PROVIDED "AS IS".  NO WARRANTIES, WHETHER EXPRESS, IMPLIED
 OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF
 MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE.
 VALVANO SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL,
 OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.
 For more information about my classes, my research, and my books, see
 http://users.ece.utexas.edu/~valvano/
 */

// U0Rx (VCP receive) connected to PA0
// U0Tx (VCP transmit) connected to PA1
#include <stdio.h>
#include <stdint.h>
#include "PLL.h"
#include "UART.h"
#include "ST7735.h"
#include "ADC.h"
```

```
#include <rt_misc.h>
#include <string.h>
#include "OS.h"
#include "ifdef.h"
#include "efile.h"
//#define INTERPRETER

void Interpreter(void);

//---------------------OutCRLF---------------------
// Output a CR,LF to UART to go to a new line
// Input: none
// Output: none
void OutCRLF(void){
  UART_OutChar(CR);
  UART_OutChar(LF);
}

#define PE4  (*((volatile unsigned long *)0x40024040))
        // 1) format
// 2) directory
// 3) print file
// 4) delete file
// execute   eFile_Init();  after periodic interrupts have started
#ifdef INTERPRETER
void Interpreter(void){
        char input_str[30];
        char ch;
        int input_num,i,device,line;
        int freq, numSamples;
        UART_Init();            // initialize UART
        OutCRLF();
        OutCRLF();

        //Print Interpreter Menu
        printf("Debugging Interpreter Lab 1\n\r");
        printf("Commands:\n\r");
        printf("LCD\n\r");
        printf("OS-K - Kill the Interpreter\n\r");
        #ifdef PROFILER
```

```
        printf("PROFILE - get profiling info for past events\n\r");
        #endif
        printf("FORMAT - format the file system\n\r");
        printf("LS - prints directory\n\r");
        printf("CAT - prints file contents\n\r");
        printf("RM - delete\n\r");
        printf("TOUCH - create a file\n\r");
        printf("INIT - initialize file system\n\r");
        printf("WRT - write to a file\n\r");

        while(1){
                //PE4^=0x10;
                printf("\n\rEnter a command:\n\r");
                for(i=0;input_str[i]!=0;i++){input_str[i]=0;}           //Flush the input_str
                UART_InString(input_str,30);
                if(!strcmp(input_str,"LCD")){
                        printf("\n\rMessage to Print: ");
                        for(i=0;input_str[i]!=0;i++){input_str[i]=0;}           //Flush the input_str
                        UART_InString(input_str,30);
                        printf("\n\rNumber to Print: ");
                        input_num=UART_InUDec();
                        printf("\n\rDevice to Print to: ");
                        device = UART_InUDec();
                        printf("\n\rLine to Print to: ");
                        line = UART_InUDec();
                        ST7735_Message(device,line,input_str,input_num);
                } else if(!strcmp(input_str,"OS-K")){
                        OS_Kill();
                        #ifdef PROFILER
                } else if(!strcmp(input_str,"PROFILE")){
                        printf("\n\rThreadAddress\tThreadAction\tThreadTime\n\r");
                        for(i=0; i<PROFSIZE; i++){
                                printf("%lu\t\t%lu\t\t%lu\n\r",(unsigned
long)ThreadArray[i],ThreadAction[i],ThreadTime[i]/80000);
                        }
                        #endif
                } else if(!strcmp(input_str,"FORMAT")){
                                eFile_Format();                 //formats the file system
                } else if(!strcmp(input_str,"LS")){
                          eFile_Directory(&UART_OutChar);   //prints directory
```

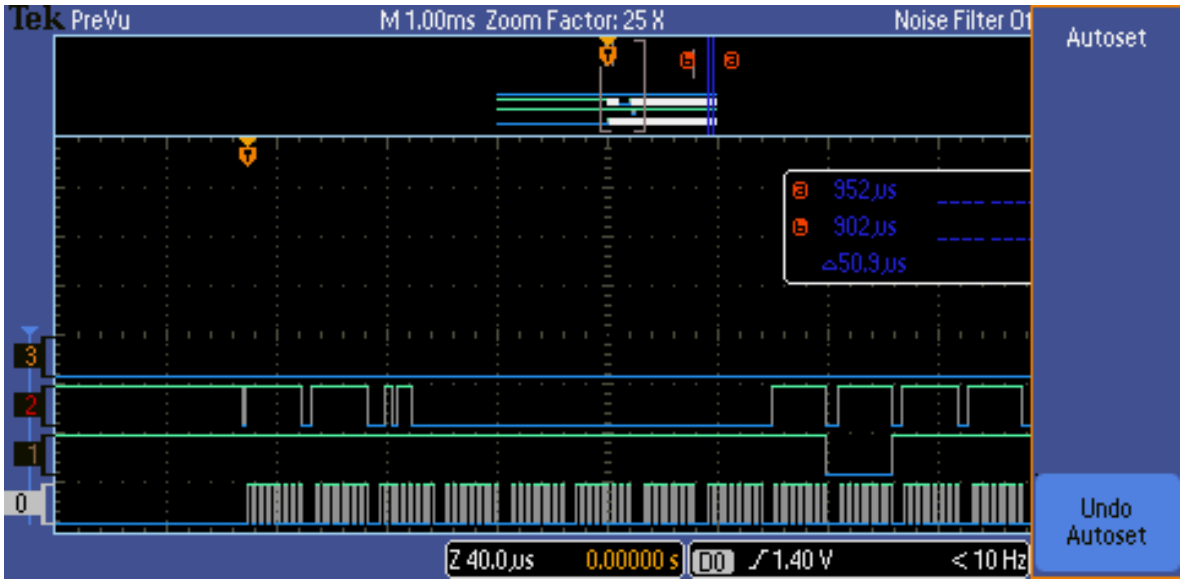
```
		} else if(!strcmp(input_str,"CAT")){
				printf("\n\rFile to View: ");
				for(i=0;input_str[i]!=0;i++){input_str[i]=0;}		//Flush the
input_str
				UART_InString(input_str,30);
				//Opens a file for reading
				if(eFile_ROpen(input_str))
				{
					printf("\n\rError or File does not exist");
					continue;
				}
				//reads file contents
				while(!eFile_ReadNext(&ch)){
					printf("%c",ch);
				}
				//close file
				eFile_RClose();

		} else if(!strcmp(input_str,"RM")){
			printf("\n\rFile to Delete: ");
			for(i=0;input_str[i]!=0;i++){input_str[i]=0;}		//Flush the input_str
			UART_InString(input_str,30);
			//Delete file
			if(eFile_Delete(input_str)){
				printf("\n\rError or File does not exist");
			}
		} else if(!strcmp(input_str,"TOUCH")){
			printf("\n\rFile to Create: ");
			for(i=0;input_str[i]!=0;i++){input_str[i]=0;}		//Flush the input_str
			UART_InString(input_str,30);
			//create file
			if(eFile_Create(input_str)){
				printf("\n\rError or No room left");
			}
		} else if(!strcmp(input_str, "INIT")){
			eFile_Init();	//initialize file system
		} else if(!strcmp(input_str, "WRT")){
			printf("\n\rFile to Write: ");
			for(i=0;input_str[i]!=0;i++){input_str[i]=0;}		//Flush the input_str
			UART_InString(input_str,30);
```

```
                    eFile_RedirectToFile(input_str);
                    for(i=0;input_str[i]!=0;i++){input_str[i]=0;}          //Flush the input_str
                    UART_InString(input_str,30);
                    printf("%s", input_str);
                    eFile_EndRedirectToFile();
                }
                else{
                    printf("\n\rInvalid Command. Try Again\n\r");
                }
        }
}
#endif
```

D) Measurement Data

1) SD card read bandwidth and write bandwidth (procedure 1)



Line 1 is the SSI CLK line

Line 2 is the SSI RX line

Line 3 is the SSI TX line

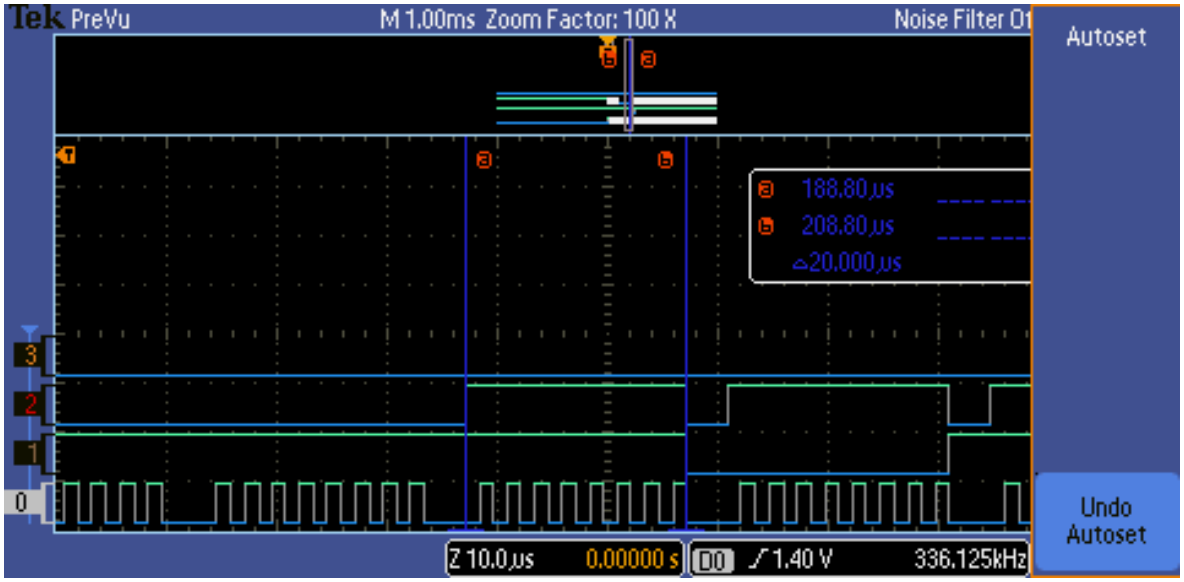
SD card write bandwidth:

one block / 2.84 ms * 1 block/ 512 bytes * 1000 ms/ 1 s = 190.760 kbytes/sec

SD card read bandwidth:

one block/ 1.436 ms * 1 block/512 bytes * 1000 ms/ 1 s = 356.545 kbytes/sec

2) SPI clock rate (procedure 1)



We ran the SPI clock at a 1/25th (400 kbps) the normal rate (10 MHz) to observe it on the oscilloscope:
Data Transmit Rate: 1 byte/20 us
Bandwidth: 1 byte/20 us * 1000000 s/us * 25 = 1.25 Mbytes/sec
This bandwidth is much higher than the read or write bandwidth because SPI must transmit and receive extra bytes involved in error correction and protocol.

3) Two SPI packets (procedure 1)

E) Analysis and Discussion (2 page maximum). In particular, answer these questions
1) Does your implementation have external fragmentation? Explain with a one sentence answer.
There is no external fragmentation because the file allocation table allows for files to be allocated anywhere on disk because the free blocks are all linked together and do not have to be contiguous.

2) If your disk has ten files, and the number of bytes in each file is a random number, what is the expected amount of wasted storage due to internal fragmentation? Explain with a one sentence answer.
The expected amount of internal fragmentation is half the block size for each block, so for 10 files, 10*256 = 2560 bytes of internal fragmentation.

3) Assume you replaced the flash memory in the SD card with a high speed battery-backed RAM and kept all other hardware/software the same. What read/write bandwidth could you expect to achieve? Explain with a one sentence answer.

The read/write bandwidth would be limited by the speed of load/store instructions to/from RAM. Loading or storing four bytes per one cycle at 80 MHz gives 4 bytes * 80 Mhz = 320 Mbytes/sec.

4) How many files can you store on your disk? Briefly explain how you could increase this number (do not do it, just explain how it could have been done).
Our file system can store 42 files on disk (see above calculations). This could be increased by either expanding the file system (expanding the directory and block number size) or by using compression algorithms on the files to reduce their size and or directory information.

5) Does your system allow for two threads to simultaneously stream debugging data onto one file? If yes, briefly explain how you handled the thread synchronization. If not, explain in detail how it could have been done. Do not do it, just give 4 or 5 sentences and some C code explaining how to handle the synchronization
No, in the interpreter for a file write we use file redirection to print to files which isn't thread safe. One solution to make it thread safe is to add semaphores on the file redirection that has a mutex on the file so that one thread writes while the other blocks. This would require a larger FIFO since there could be several writes that are blocked while the current thread with access to the file releases its mutex. Another solution would be to write it directly to the disc with arrays/FIFOs and open/close the file after every write event, this would prevent a race condition with the two threads