Lab 4: Solid-State Disk and File System

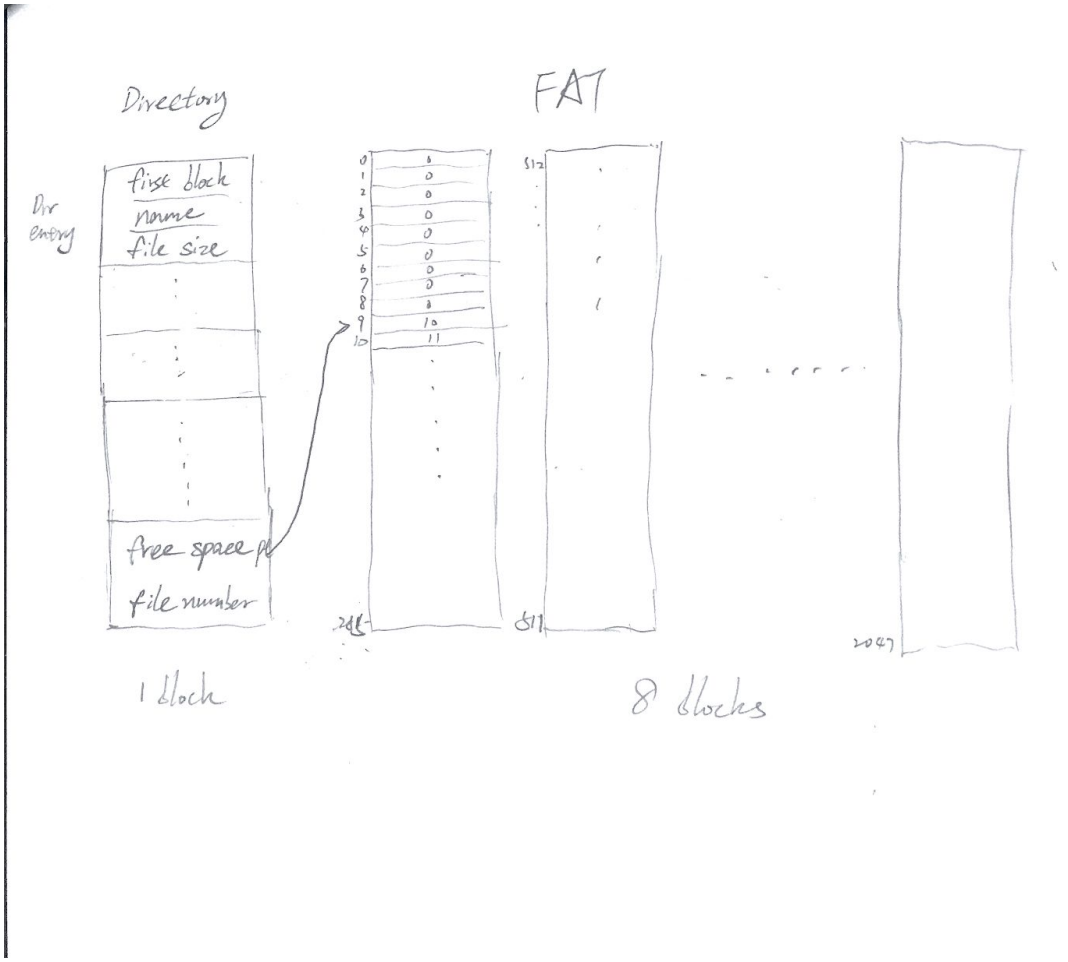
Xinyuan (Allen) Pan & Qinyu (Jack) Diao

Objective

The objective of this lab was to interface an SD card to the TM4C123 SPI port. We write a file-system driver, implement a disk storage protocol, develop a simple directory system, and implement the address translation from logical to physical address. The debugging information streams onto the disk. We also add commands to the interpreter to test and evaluate the solid-state disk.

Software Design

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



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

```
#define BLOCK_SIZE                  512         // in byte
#define FS_SIZE                     2048        // in number of blocks
#define BLOCK_NUMBER_BYTES 2
#define DIR_SIZE                    1                   // in number of blocks
#define DIR_ENTRY_SIZE              32          // in byte
#define FAT_SIZE                    (BLOCK_NUMBER_BYTES * FS_SIZE / BLOCK_SIZE)  // in number of
blocks
#define FAT_START_BLOCK             DIR_SIZE
#define FAT_END_BLOCK               FAT_SIZE
#define FIRST_FILE_BLOCK        (DIR_SIZE + FAT_SIZE)
#define MAX_FILE_NUM                12

/* The directory entry stored in disk */
typedef struct {
        block_t  firstBlock;
        BYTE  name[15];
        file_len_t fileSize;    // in byte
} DirEntry;

/* The file object created after opened*/
typedef struct {
        DirEntry *entry;
        BYTE data[BLOCK_SIZE];  // one block in memory at most
        block_t curBlock;               // currently opened block, if any
        file_len_t pos;         // current cursor position (the next to read/write)
        int rw;                         // 0 for read, 1 for write
} File;


struct __directory {
        DirEntry entries[MAX_FILE_NUM];
        block_t freeSpace;  // first block in free space link
        BYTE fileNum;
};

struct __directory directory;   // have to make sure this is under DIR_SIZE * BLOCK_SIZE

BYTE fat[FS_SIZE];  // zero mean no next block
File openedFile;  // only one opened file allowed
int fileOpened = 0;


typedef enum {
        EFILE_SUCCESS = 0,
        EFILE_FAILURE = 1
} FRESULT;

/**
```

```
 * @details This function must be called first, before calling any of the other eFile functions
 * @param  none
 * @return 0 if successful and 1 on failure (already initialized)
 * @brief  Activate the file system, without formating
 */
FRESULT eFile_Init(void); // initialize file system


/**
 * @details Erase all files, create blank directory, initialize free space manager
 * @param  none
 * @return 0 if successful and 1 on failure (e.g., trouble writing to flash)
 * @brief  Format the disk
 */
FRESULT eFile_Format(void); // erase disk, add format

/**
 * @details Create a new, empty file with one allocated block
 * @param  name file name is an ASCII string up to seven characters
 * @return 0 if successful and 1 on failure (e.g., already exists)
 * @brief  Create a new file
 */
FRESULT eFile_Create(char name[]);  // create new file, make it empty


/**
 * @details Open the file for writing, read into RAM last block
 * @param  name file name is an ASCII string up to seven characters
 * @return 0 if successful and 1 on failure (e.g., trouble reading from flash)
 * @brief  Open an existing file for writing
 */
FRESULT eFile_WOpen(char name[]);      // open a file for writing


/**
 * @details Save one byte at end of the open file
 * @param  data byte to be saved on the disk
 * @return 0 if successful and 1 on failure (e.g., trouble writing to flash)
 * @brief  Format the disk
 */
FRESULT eFile_Write(char data);

/**
 * @details Deactivate the file system. One can reactive the file system with eFile_Init.
 * @param  none
 * @return 0 if successful and 1 on failure (e.g., trouble writing to flash)
 * @brief  Close the disk
 */
FRESULT eFile_Close(void);
```

```
/**
 * @details Close the file, leave disk in a state power can be removed.
 * This function will flush all RAM buffers to the disk.
 * @param  none
 * @return 0 if successful and 1 on failure (e.g., trouble writing to flash)
 * @brief  Close the file that was being written
 */
FRESULT eFile_WClose(void); // close the file for writing

/**
 * @details Open the file for reading, read first block into RAM
 * @param  name file name is an ASCII string up to seven characters
 * @return 0 if successful and 1 on failure (e.g., trouble reading from flash)
 * @brief  Open an existing file for reading
 */
FRESULT eFile_ROpen(char name[]);      // open a file for reading


/**
 * @details Read one byte from disk into RAM
 * @param  pt call by reference pointer to place to save data
 * @return 0 if successful and 1 on failure (e.g., trouble reading from flash)
 * @brief  Retreive data from open file
 */
FRESULT eFile_ReadNext(char *pt);       // get next byte

/**
 * @details Close the file, leave disk in a state power can be removed.
 * @param  none
 * @return 0 if successful and 1 on failure (e.g., wasn't open)
 * @brief  Close the file that was being read
 */
FRESULT eFile_RClose(void); // close the file for reading


/**
 * @details Display the directory with filenames and sizes
 * @param  printf pointer to a function that outputs ASCII characters to display
 * @return none
 * @brief  Show directory
 */
void eFile_Directory(void printf(const char *, ...));

/**
 * @details Delete the file with this name, recover blocks so they can be used by another file
 * @param  name file name is an ASCII string up to seven characters
 * @return 0 if successful and 1 on failure (e.g., file doesn't exist)
 * @brief  delete this file
 */
```

```
FRESULT eFile_Delete(char name[]);  // remove this file

/**
 * @details open the file for writing, redirect stream I/O (printf) to this file
 * @note if the file exists it will append to the end<br>
 If the file doesn't exist, it will create a new file with the name
 * @param  name file name is an ASCII string up to seven characters
 * @return 0 if successful and 1 on failure (e.g., can't open)
 * @brief  redirect printf output into this file
 */
FRESULT eFile_RedirectToFile(char *name);


/**
 * @details close the file for writing, redirect stream I/O (printf) back to the UART
 * @param  none
 * @return 0 if successful and 1 on failure (e.g., trouble writing)
 * @brief  Stop streaming printf to file
 */
FRESULT eFile_EndRedirectToFile(void);
```

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

```
static void parse_fsinit (char cmd[][20], int len) {
        if (eFile_Init()) {
                printf("fs: file system init failed");
        }
}
static void parse_ls(char cmd[][20], int len) {
        eFile_Directory(printf);
}
static void parse_format(char cmd[][20], int len) {
        if (eFile_Format()) {
                printf("format: failed.\n\r");
        }
}
static void parse_cat(char cmd[][20], int len) {
        if (len == 1) {
                printf("cat: no file name.\n\r");
                return;
        }
        if (eFile_ROpen(cmd[1])) {
                printf("cat: open failed.\n\r");
                return;
        }

        char ch;
        while (eFile_ReadNext(&ch) == 0) {
                printf("%c", ch);
```

```
        }
        if (eFile_RClose()) {
                printf("cat: close failed.\n\r");
                return;
        }
}
static void parse_rm(char cmd[][20], int len) {
        if (len == 1) {
                printf("rm: no file name.\n\r");
                return;
        }

        if (eFile_Delete(cmd[1])) {
                printf("rm: delete failed.\n\r");
                return;
        }
}
```

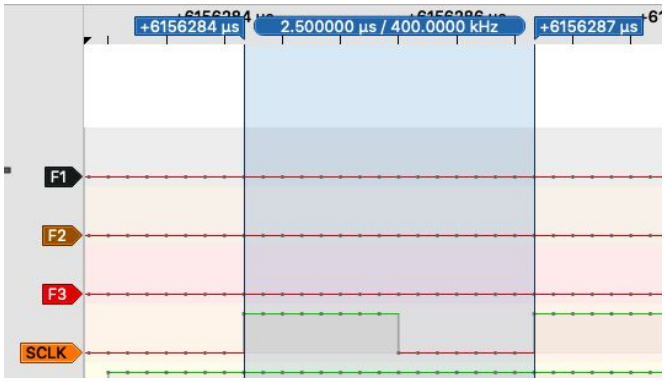
Measurement Data

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

Write Bandwidth: 3.67ms/block (7.17μs/byte)

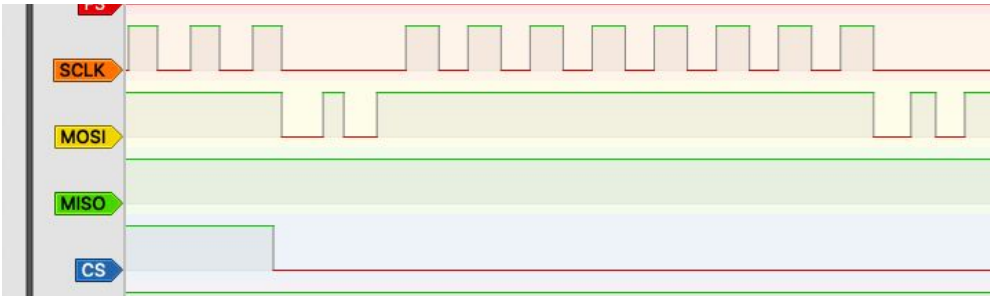
Read Bandwidth: 2.32ms/block (4.53μs/byte)

2) SPI clock rate (Procedure 1)



Frequency: 400kHz

3) Two SPI packets (Procedure 1)



Analysis and Discussion

1) Does your implementation have external fragmentation? Explain with a one sentence answer.

No, there's no external fragmentation because the system uses FAT for allocation, allocating one block at a time.

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.

On average, each file wastes ½ * size bytes of the allocation resolution (here it's 512 byte), therefore the expected amount of wasted storage due to internal fragmentation is 10 * ½ * 512B = 2.5KiB.

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 same bandwidth since the SPI clock is generated by the microcontroller, which does not change if we do not change software.

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).

$(2^{11} - 9)$ files, because the file system size is 1MB (with 9 of the blocks not available for files) and block size is 512B and each file is at least allocated one block of space. To increase this number, either increase the file system size or decrease the block size.

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, our system does not allow for two threads to simultaneously stream data onto one file.

We will need to add mutex around the write operation so that each write is operation is reentrant. Furthermore, when attempting to open a file, the thread needs to check whether it’s already opened or not then proceed (currently, trying to open an opened file just returns an error).