Lab 4 Solid-State Disk and File System

Goals
- Interface a SD card to the TM4C123 SPI port,
- Address translation from logical to physical address,
- Write a file-system driver, implementing a disk storage protocol,
- Develop a simple directory system,
- Stream debugging information onto the disk,
- Add commands to your interpreter to test and evaluate the solid-state disk.

Review
- Chapter 8 in the book,
- TM4C123 data sheet explaining SPI,
- SD_Physical_Layer_Spec.pdf explaining the SD protocol.

Starter files
- **`RTOS_Lab4_FileSystem`** (starter project for Lab 4)
  - **`eDisk.c`** and **`eDisk.h`** containing the low-level SDC card disk driver (in **RTOS_Labs_common** folder, modified from **`SDC_4C123`** starter project)
  - **`eFile.c`** and **`eFile.h`** containing prototypes for the file system (in **RTOS_Labs_common** folder, not a requirement)
  - **`Lab4.c`** (from **RTOS_Lab4_FileSystem**)

Background

The overall goal is to develop a solid-state disk. You will use the low-level **eDisk** driver that reads and writes blocks, and are required to write your own high-level file system on top of **eDisk** for this lab. Chapter 8 describes a number of simple file systems that you can use as a basis, but you must design and implement your own file system. Your system will be able to create files, stream ASCII data to the end of a file using **fputc**, printout the entire contents of a file, and delete files. In addition, your system will be able to list the names and sizes of the available files. The **`eDisk.c`** and **`eDisk.h`** starter files interface a SD card using the SPI port. You will write a series of software functions that make it appear as a disk. In addition, you will add interpreter commands to manage the disk:

- Format the SD card
- Display the directory
- Printout the contents of a file as ASCII characters via the serial port
- Delete a file

The robot thread (see **`Lab4.c`**) will stream debugging data onto the disk in the following manner

1) Create a new empty file (or open an existing file)
2) Redirect the serial output stream to write data onto the disk instead of the serial port, see **fputc** in **`OS.c`**
3) The robot thread generates debugging output using standard command, **printf**
4) At the end of the run, the serial output stream is redirected to the serial port
5) The file is closed

There is a great deal of flexibility in the implementation of this lab, but the following requirements must be satisfied. You do not have to use every byte of the SD card, but the file system should be at least 1 mebibyte. There must be a low-level device driver (**`eDisk.h`** and **`eDisk.c`**), and only this software alone can directly access the SD card. A driver means there are separate header and code files. Space is allocated to a file in fixed-size blocks (e.g., two files do not store data into the same block.) The high-level structure should include a directory that supports multiple logical files. Your system must support at least 10 files. The files are dynamically created and can grow in size (shrinking is easy to do but not necessary in this lab.)

Figure 4.1 shows the call graph illustrating the physical layer (**eDisk**), the logical layer (**eFile**) and the user layer (interpreter and robot). There must be three software layers, including:

1) two foreground threads (robot and interpreter) that share access to the disk,
2) a middle-level logical file system (e.g., **`eFile.h`** and **`eFile.c`**), and
3) a low-level SD card access system (e.g., **`eDisk.h`** and **`eDisk.c`**).

Each layer should have its own code file, and careful thought should go into deciding which components are private and which are public. All information (directory, linking and data) must be stored on the SD card. The SD card is

nonvolatile. This means if the microcontroller were to loose power, all information would be accessible when power is restored back to the microcontroller. You do not have to handle brown-out or sudden power failure in this lab. I.e., you do not have to write a power failure interrupt service routine that backs up the RAM-based data onto the disk. However, you should be able to close all files, turn the power off, turn the power back on, and have all files intact.

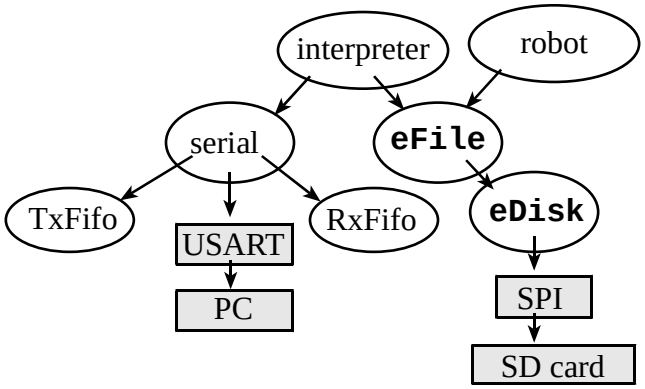

Figure 4.1. Call graph of the solid-state disk system.

The specific function prototypes (in **`eFile.h`**) are included for illustration purposes. I.e., **`eFile.h`** is given to you not as a detailed lab specification, but rather as a way to specify the level of complexity we expect you to achieve in this lab. You are free to change function names and parameters, as long as the basic operations are supported. You may choose any size for the fixed-size disk blocks, but there are 512-byte block read/write functions available in the **eDisk** starter code. You may implement any disk allocation method, as long as files can grow in size. It is necessary to store copies of pointers, counters and data into regular memory when a file is open, but when a file is closed, you should leave the disk in a consistent state (ready for a power loss). It is ok to support multiple files being open at the same time, but there is no requirement to do so. Likewise, you can support subdirectories and hierarchical folders, but you are not required to do so. You will have to handle mutual exclusion as both the interpreter and robot threads can access the disk. However, you may restrict the use of data streaming via **fputc** to just one thread if you wish.

Notice in Figure 4.1 that the file system performs no UART input/output. It is good style not to perform serial I/O within the **eFile** driver, because to do so would couple the UART and **eFile** modules unnecessarily. There are a number of good solutions that support the display of disk information out the UART port. The first method uses a shared buffer passed between modules via call by reference. Assume the function **`UART_OutString`** is an UART function to output an ASCII string. For example, to display a directory, the interpreter could execute:

```
void PrintDirectory(void) { char buffer[100];
  eFile_PrintDirectory(buffer);
  UART_OutString(buffer);
}
```

The problem with this solution is the proper sizing of the buffer in case of large directories. Such solutions can easily lead to buffer overflow problems if maximum sizes are not carefully determined or known. In addition, the formatting of the directory printout is determined and fixed in the **eFile** driver, but should really be up to the caller (interpreter).

In the second method, the interpreter passes a pointer to a callback function for printing to the **eFile** function instead. Assume the function **`UART_OutChar`** is a serial function to output one ASCII character to the PC:

```
void PrintDirectory(void) {
  eFile_PrintDirectory(&UART_OutChar);
}
```

This still suffers from the restriction of the formatting being determined by the **eFile** diver. In addition, the fixed callback signature creates further unnecessary dependencies and restrictions.

The third method is the one realized by the provided **eFile** driver prototype for both directories and files out of the box. It requires processing data entry by entry (in case of directories) or byte by byte (in case of files), where the driver provides routines to return the required information in its native form, and where the caller is in charge of using and formatting the data as desired. For example, to print a directory, the interpreter executes:

```
void PrintDirectory(void) { char *name; unsigned long size;
  if(eFile_DOpen(“”)) return;       // directory or disk does not exist
  while(!eFile_DirNext(&name, &size)) {
    UART_OutString(name);
    UART_OutString(“  (“);
    UART_OutUDec(size);
    UART_OutString(“)\n\r”);
  }
  eFile_DClose();
}
```

Note that the **eFile_DirNext** function returns the file **size** and a *pointer* to the file **name** via call by reference. Specifically, for the file name, a char pointer is returned via an **eFile_DirNext** argument of **char**** type (pointer/reference to a char/string pointer). The returned pointer must point to a static string buffer that is maintained internally in the **eFile** driver and that is alive and valid even after **eFile_DirNext** returns (i.e. can’t be allocated as a local variable on the **eFile_DirNext** stack). Alternatively, the buffer can be allocated in the caller (interpreter) and passed by reference as in the first method above. In contrast to complete directories, the maximum length of file names in a file system is typically limited, allowing the caller to determine proper buffer sizes.

Preparation

1) Write main test programs to measure the bandwidth of your SD card using the **eDisk** driver. Develop instrumentation to measure the maximum rate at which you can continuously write to the SD card. Develop a similar program to measure the maximum rate at which you can continuously read from the SD card. Your method should include output to unused port pins connected to a logic analyzer. During testing and checkout, you will also be connecting the SPI pins (CS SCLK MOSI MISO) to a logic analyzer.

2) Design and implement your file system. There should be separate **eFile.c** and **eFile.h** files containing software that implements the file system.

3) Write simple main programs with which you can test your **eFile.c**. You should some write "bad" routines, which use the file system improperly: opening two files, closing a file that is not open, writing to a file that doesn't exist, etc.

4) Edit the **Lab4.c** starter file to be compatible with your file system.

5) Add interpreter commands to manage the file system (format, print directory, create file, write to file, print file, delete file).

Procedure

1) First, use *Testmain1* to verify your microcontroller can read/write blocks to your SD card. Next, run your main programs from Preparation 1) measuring the read bandwidth and write bandwidth of the SD card. In addition to the measurement output pin(s), connect debugging output pins to the logic analyzer. Capture SPI signals as the software reads one block. Make two printouts for your report. First, capture and print one command frame (like Lecture 7, Slide 7, Figure 6.13). Zoom in close enough that you can determine which command is being issued (e.g., CMD17). Second, capture and print one data packet (like Lecture 7, Slide 12). Measure the SPI clock rate, and compare the SPI bandwidth to the disk bandwidth.

2) Next, debug your file system using software written in Preparations 2) and 3). Use *Testmain2* to verify your microcontroller can read/write files to your SD card.

3) Test the entire system using the **Lab4.c** main program. Make sure to also test your interpreter.

4) Finally, make sure your files are persistent across power offs. You need to create some files, power off the board, power the board back on and print the contents of the directory. You should see the files you created before powering off the board.

Checkout (show this to the TA)

Connect up the logic analyzer to the four SPI pins and explain the communication to your TA during a disk access. You should be able to demonstrate to the TA at least 3 files, some of them having more than one block, and at least one of them having 3 blocks. Demonstrate each of the interpreter commands.

Deliverables (exact components of the lab report and lab submission)

A) Objectives (1/2 page maximum)
B) Hardware Design (none)
C) Software Design (documentation and code)
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)
3) High level software system (the new interpreter commands)

D) Measurement Data
1) SD card read bandwidth and write bandwidth (Procedure 1)
2) SPI clock rate (Procedure 1)
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.
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.
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.
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).
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.

Hints

1) The display and SD card share the same SSI port with separate chip select (CS) pins. If you end up with multiple threads accessing display and SD card at the same time, you need to protect simultaneous SSI port accesses, e.g. by reusing the **`LCDFree`** semaphore to mutually exclude both display and file system. It is highly recommended to integrate the SDC and LCD in this manner. This way you will be able to use both the LCD and SDC for your Lab 5, Lab 6 and Lab 7 systems without any issues.

2) If you ever want to run the original **`SDC_4C123`** starter project on the sensor board, you need to specify PB0 as the chip select for the SDC (in **`eDisk.c`**)

```
#define SDC_CS_PB0 1
#define SDC_CS_PD7 0
```

For fun, you can also run the **`SDCFile_4C123`** project on the sensor board, again by specifying PB0 as the chip select for the SDC. ***However****: students cannot use this FAT16 or any file system written by others. They must implement their own high level file system.*

3) The *realmain* demonstrates use of **stdio** redirection to redirect **`printf`** output to a file in the robot thread. This relies on corresponding support in **`OS.c`** as provided in the starter code. If you modified the I/O redirection code, make sure that redirection to a file works as intended.

4) Everyone's OS runs a little different. Please edit **`Lab4.c`** so that the ST7735R is initialized before it is used.

5) The **eDisk** includes a **disk_timerproc** function that must be hooked up to a period timer interrupt for the driver to work properly. The driver includes commented out sample code to connect *Timer5A* to **disk_timerproc**, but you need to integrate that with your specific OS setup. Please make sure the **disk_timerproc** periodic background task is running at least every 10ms before your call **`eDisk_Init(0)`** or **`eFile_Init()`**. There are multiple possible solutions. In the starter project, **disk_timerproc** is run as a periodic background thread and a dedicated **`Init`** task is included to perform user-level file system initializations right after the OS kernel (and hence the periodic background thread) has been launched.

6) Make sure **`eDisk_Init(0)`** is called before calling any **eDisk** functions, and **`eFile_Init`** is called before calling **eFile** functions. Furthermore, **`eDisk_Mount`** is required to be called before accessing any file in order to mount the disk and load any file system metadata, such as file system tables, etc.

7) I had to suspend the scheduler (`NVIC_ST_CTRL_R=6`) during **`eFile_Format`** because I was getting timeout errors in **`eDisk_WriteBlock`** (I probably could have run **`eFile_Format`** at high priority). Remember to resume the scheduler before leaving **`eFile_Format`**:

```
unsigned long OS_LockScheduler(void){
  unsigned long old = NVIC_ST_CTRL_R;
  NVIC_ST_CTRL_R = NVIC_ST_CTRL_ENABLE+NVIC_ST_CTRL_CLK_SRC;
  return old;
}

void OS_UnLockScheduler(unsigned long previous){
  NVIC_ST_CTRL_R = previous;
}
```