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
- Section 6.6 and Chapter 7 in the book,
- TM4C123 data sheet explaining SPI,
- SD_Physical_Layer_Spec.pdf explaining the SD protocol.

Starter files
- **eDisk.c**, **eDisk.h** containing the low-level software driver,
- **eFile.h** containing prototypes for the file system (not a requirement),
- **Lab4.c** containing a few test programs.

Background

The overall goal is to develop a solid-state disk. Chapter 7 describes a number of simple file systems, but you are free to 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**, **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**
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 5.1 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.
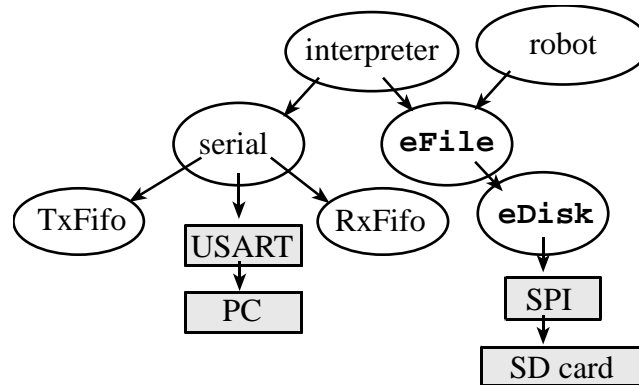
Figure 5.1. Call graph of the solid-state disk system.

The specific function prototypes (**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. 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 5.1 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 call by reference. Assume the function **Serial_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_Directory(buffer);
  Serial_OutString(buffer);
}
```

Assume the function **Serial_OutChar** is an USART function to output one ASCII character. In the second method, the interpreter passes a function pointer to the **eFile** function:

```
void PrintDirectory(void){
  eFile_Directory(&Serial_OutChar);
}
```

The third method streams data byte by byte. For example, to print a file, the interpreter could execute:

```
void PrintFile(char name[]){ char data;
  if(eFile_ROpen(name)) return;        // file or disk does not exist
  do{
    if(eFile_ReadNext(&data)) return; // done
    Serial_OutChar(data);}
  while(1);
}
```

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.

J. W. Valvano, A. Gerstlauer   3/2/2016

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, 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.

3) Lastly, test the entire system using the **Lab4.c** main program.

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) We suggest you remove the LCD code from the system because the SDC and LCD share the same SSI0. Most students do not use the SDC for their Lab 6 and Lab 7 systems.

2) I used this code to stream data onto the file. There should have been a way to use **fprintf** and pass a parameter to **fputc**, but I couldn't quite make the compiler happy. If you get a better solution, post it on the discussion forums.

```
#include "eFile.h"

int StreamToFile=0;                    // 0=UART, 1=stream to file

int eFile_RedirectToFile(char *name){
  eFile_Create(name);                  // ignore error if file already exists
  if(eFile_WOpen(name)) return 1;  // cannot open file
  StreamToFile = 1;
  return 0;
}

int eFile_EndRedirectToFile(void){
  StreamToFile = 0;
  if(eFile_WClose()) return 1;     // cannot close file
  return 0;
}

int fputc (int ch, FILE *f) {
  if(StreamToFile){
    if(eFile_Write(ch)){           // close file on error
        eFile_EndRedirectToFile(); // cannot write to file
        return 1;                  // failure
    }
    return 0; // success writing
  }

   // regular UART output
  UART_OutChar(ch)'
  return 0;
}

int fgetc (FILE *f){
  return (Serial_InChar());
}
```