

Lab 1f Solid-State Disk

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, 2nd edition, by Jonathan W. Valvano, published by Thomson-Engineering Publishers, copyright © 2006.

- Goals**
- Interface a serial EEPROM chip (128 kibibyte) to the 9S12 SPI port,
 - Address translation from logical to physical address
 - Implement a FAT disk storage protocol,
 - Write a disk device driver,
 - Develop a simple directory system,
 - Use a command line interpreter to test and evaluate the solid-state disk.

- Review**
- Chapter 7 on SCI and SPI interfacing,
 - Chapter 9 on timing diagrams,
 - Data sheets on the Microchip 25LC1024.

- Starter files**
- **SPI.c**, **SPI.h** in the **tester** project
 - **SCIA** project

Background

The overall goal is to develop a solid-state disk. Your system will be able to create files, append data to the end of a file, 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. Basically, you will interface a 25LC1024 serial EEPROM using the SPI port, then write a series of software functions that make it appear as a disk. Solid-state disks can be also made from battery-backed RAM or flash EEPROM. A personal computer uses disks made with magnetic storage media and moving parts. While this hard disk technology is acceptable for the personal computer because of its large size (>gibibyte) and low cost (<\$100 OEM), it is not appropriate for an embedded system, because of its physical dimensions, electrical power requirements, noise, sensitivity to motion (maximum acceleration), and weight. Embedded applications that might require disk storage include data acquisition, data base systems, and signal generation. The personal desk accessory (PDA) devices currently employ solid-state disks because of their small physical size, and low power requirements. Unfortunately, solid-state disks have smaller sizes and higher cost/bit than the traditional magnetic storage disk. In particular, you will implement a 128 KiB disk that costs about \$3. The cost/bit is therefore about \$24/MiB. Compare this cost to a 1 TiB hard drive that costs about \$100. This cost/bit is only \$0.0001/MiB. Figure 1.1 shows the data flow graph and Figure 1.2 shows the call graph.

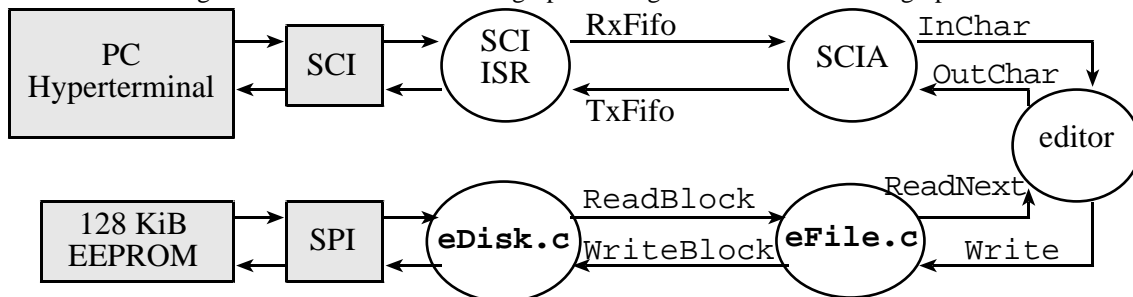


Figure 1.1. Data flow graph of the solid-state disk system.

There is a great deal of flexibility in the implementation of this lab, but the following requirements must be satisfied. Every byte of the serial EEPROM must be accessible. There must be a low-level device driver, and this software alone can directly access the serial EEPROM. 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 6 files. The files are dynamically created and can grow in size (shrinking is easy to do but not necessary in this lab.) There must be three software layers including a high-level command interpreter (e.g., a **main.c** file with the editor program), a middle-level logical file system (e.g., **file.h** and **file.c**), and a low-level memory-access system (e.g., **disk.h** and **disk.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 serial EEPROM. The EEPROM is nonvolatile. This means if the microcontroller were to loose power, all information would be accessible when power is restored back to the microcontroller.

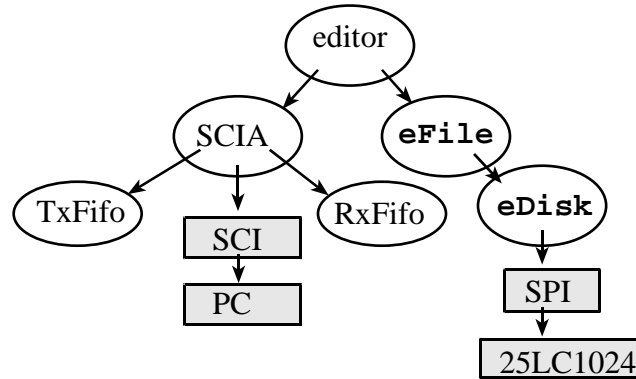


Figure 1.2. Call graph of the solid-state disk system.

The specific function prototypes are included for illustration purposes. 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 is a 256-byte block write function built into the EEPROM. You are free to adjust the specific syntax of the interpreter, as long as similar functions are available. You may implement any disk allocation method, as long as files can grow in size.

The first step is to interface the 128 kibibyte serial EEPROM to the microcontroller using the SPI interface. The second step is to develop a low-level device driver for the disk. Disks are partitioned into fixed-size blocks. The function of the low-level is to allow read/write access to the solid-state disk (i.e., the serial EEPROM.) There should be separate `eDisk.c` and `eDisk.h` files containing software that implements the low-level disk operations. The following prototypes illustrate these operations. The `eDisk_Open` command should enable the SPI interface to your serial EEPROM. All routines return a 1 if successful and a 0 on failure. You might not need routines that read and write individual bytes. I.e., you might be able to solve this lab with just `eDisk_Open`, `eDisk_ReadBlock` and `eDisk_WriteBlock`.

```

int eDisk_Open(void); // initialize disk interface
int eDisk_ReadByte(unsigned char *bytePt, // result returned by reference
                  unsigned short blockNum, // which block to read from
                  unsigned short byteNum); // which byte within this block
int eDisk_ReadBlock(unsigned char *pt, // result returned by reference
                   unsigned short blockNum); // which block to read
int eDisk_WriteByte(unsigned char byte, // value to be saved
                   unsigned short blockNum, // which block to write into
                   unsigned short byteNum); // which byte within this block
int eDisk_WriteBlock(unsigned char *pt, // values to be saved
                    unsigned short blockNum); // which block to write

```

The second step is to develop a file system. There should be separate `eFile.c` and `eFile.h` files containing software that implements the file system. There are three components of the file system: directory, allocation, and free-space management. The first component is the **directory**, as shown in Figure 1.3. The directory contains a mapping between the symbolic filename and the physical address of the data. Specific information contained in the directory might include the file name, the number of the first block containing data, and the total number of bytes stored in the file. Other information that one often finds in a directory entry is a pointer to the first block of the file, access rights, date of creation, date of last modification, and file type. One possible implementation places the directory in block 0 (again, you are free to develop your own method). In this simple system, all files are listed in this one directory (there are no subdirectories). There is one fixed-size directory entry for each file. Each entry in Figure 1.3 contains the file name (single ASCII character), the block number of the first block containing data (0 means there is no first block), and the total number of bytes stored in the file. A null-character (0) means no file. Since the directory itself is located in block 0, 0 can be used as a null-block pointer. Since the entire directory must fit into the block 0, the maximum number of files can be calculated by dividing the block size by the number of bytes used for each directory entry.

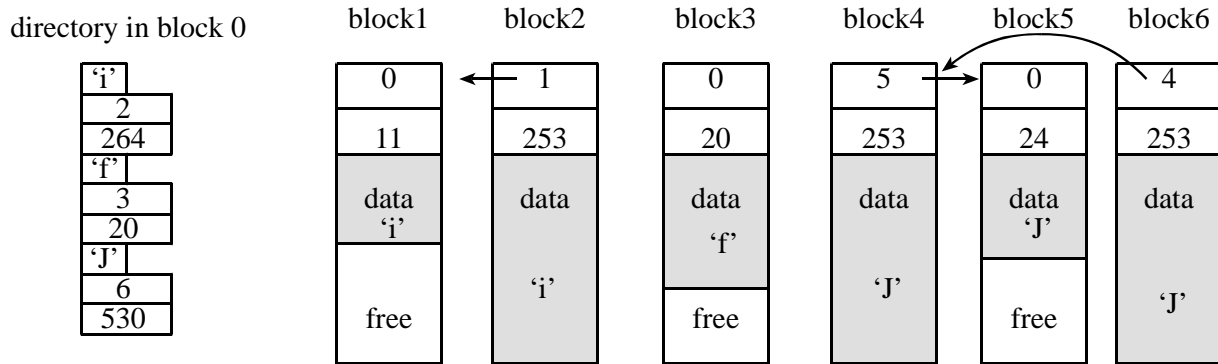


Figure 1.3. Linked file allocation with 256-byte blocks.

The second component of the file system is the **logical to physical address translation**. Logically, the data in the file are addressed in a simple linear fashion. The logical address ranges from the first to the last. There are many algorithms one could use to keep track of where all the data for a file belongs. One simple mechanism is called **linked allocation**, which is also illustrated in Figure 1.3. Recall that the directory contains the block number of the first block containing data for the file. The start of every block contains a link (the block number) of the next block, and a byte count (the number of data bytes in this block). If the link is zero, this is last block of the file. If the byte count is zero, this block is empty (contains no data). Once the block is full, the file must request a free block (empty and not used by another file) to store more data. Linked allocation is effective for systems like this that employ sequential access. Sequential read access involves two functions similar to a magnetic tape: rewind (start at beginning), and read the next data. Sequential write access simply involves appending data to the end of the file. The Figure 1.3 assumes the block size is 256 bytes and the filename has only one character. Since there are 512 blocks, the block pointers need at least 9 bits of storage. Thus, a 16-bit entry will be used for block pointers. Since each data block has a 2-byte link and a 1-byte counter, each block can store up to 253 bytes of data. The maximum file size is 511×253 , which is 129,283 bytes. You are allowed to limit the maximum file size to 65535 bytes, so a 16-bit file size parameter can be used. E.g., the size of file 'J' is 528 bytes and is stored as a 16-bit number. The file 'i' has 264 bytes, 253 of them in block 2 and the rest in block 1. All 20 bytes of file 'f' are stored in block 3.

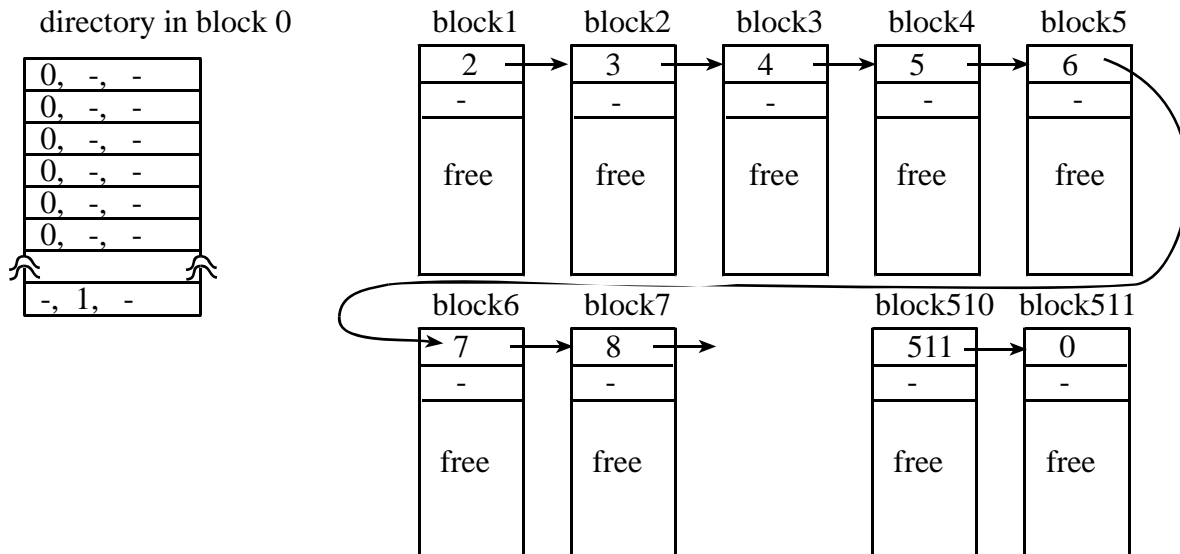


Figure 1.4. An empty disk.


```

void main(void){ unsigned char data; int ok;
  SCI_Init(115200);
  if(eFile_Init()){
    if(eFile_ROpen('J'){
      do{
        if(ok=eFile_ReadNext(&data)){ //ok=0 on end of file
          SCI_OutChar(data);
        }
      }
      while(ok);
    }
  }
}

```

eFile_Format should return an error if the call to the low-level eDisk_Open returns an error. eFile_Create should return an error if the directory is full, or if the file already exists. eFile_Open should return an error if another file is already open (only one file can be open at a time in this simple system), or if the file doesn't exist. eFile_Write should return an error if the disk is full, or if no file is open. eFile_Close should return an error if no file is open. eFile_Print and eFile_Delete should return an error if the file doesn't exist.

The third step is to develop a simple interpreter/editor that illustrates the features of your file system. This software includes the main program, which first initializes, then implements the interpreter/editor. The following commands illustrate the types of features you need to develop:

f	format the disk, erasing all data and all files
d	display the directory, including names and sizes of the files
c m	create a new empty file called "m"
p y	display the contents of file "y"
a h	open the file "h", and append characters subsequently typed characters are added to the file close the file and return to the interpreter when an <esc> is typed (\$1B)
b h	open the file "h", and add 1000 ASCII characters to the file, then close the file
c h	open the file "h", and add 10000 ASCII characters to the file, then close the file
e h	erase file "h"

Preparation (do this before your lab period)

- 1: You will draw the circuit that interfaces your memory to the 9S12. Draw a logic diagram for the memory system including pin numbers for all IC's. Show all connections to the 9S12 SPI port.
- 2: Study the memory timing diagram to choose the fastest SPI clock rate, and the setting for CPOL, CPHA.
- 3: Next, write the low-level **eDisk.c** and **eDisk.h** files. You should include debugging software, **eDisk_Test**, to test the entire 128 KiB memory. First fill the memory with zeros, then check each location for zero. Next, fill the memory with ones, then check each location for \$FF. For a third test, fill the entire memory with values equal to the low 8-bits of the address. Then, read and verify each data matches its low address.
- 4: Write a simple main program, which uses **eDisk.c**, to test the external memory and all the functions in **eDisk.c**. Report any errors to the screen.
- 5: Next, write the prototype for the file system **eFile.h**. The implementation file, **eFile.c**, is not required as part of the preparation. Instead, **eFile.c** will be developed as part of the procedure.
- 6: Next, write the high-level interpreter. Don't be fancy here. The purpose of the interpreter is to test the file system.

Procedure (do this during your lab period)

- 1: Connect the memory system to the 9S12. Perform one memory write cycle over and over (including a fixed time delay for the command to complete) and observe the serial clock and serial data on the two channel scope. Similarly, perform one memory read function over and over and observe the serial clock and serial data on the two channel scope. Either draw on paper or print from the machine the timing of an actual write and read an actual operation.
- 2: Next, debug your test program and use it to verify your EEPROM hardware is operational. After this step, you should be confident you memory system is operational.

3: Now that you have the knowledge of how the memory works, implement the file system. Develop and debug the **eFile.c** and **eFile.h** files.

4: Lastly, test the entire system. 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.

Deliverables (exact components of the lab report)

- A) Objectives (1/2 page maximum)
- B) Hardware Design
 - 1) Circuit Diagram of the 9S12/EEPROM interface with pin numbers
 - 2) Timing analysis to choose the correct clock frequency
- C) Software Design (printout of these software components)
 - 1) Low level RAM interface (**eDisk.c** and **eDisk.h** files)
 - 2) Pictures illustrating the file system protocol, showing:
 - free space management;
 - the directory; and
 - file allocation scheme
 - 3) Middle level file system (**eFile.c** and **eFile.h** files)
 - 4) High level software system (the interpreter)
- D) Measurement Data (none for this lab)
- E) Analysis and Discussion (1 page maximum)

Checkout (show this to the TA)

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.

Hints

- 1) Please don't return broken EEPROM chips back to checkout. Mark bad chips as *bad*. Bring them to my office for testing and exchange.
- 2) There is a 256-byte block write command on the EEPROM.
- 3) These routines will be used for sequential read/write access in Lab 2. It may be helpful for you to review how the file system will be used as you design and test the system. Many students report the hardest thing about this lab is all the time they had to spend during the next lab removing the bugs from their file system.
- 4) A write sequence includes an automatic, self timed erase cycle. It is not required to erase any portion of the memory prior to issuing a write command. It is however necessary to perform a WREN operation before each write command.

A great deal of confusion exists over the abbreviations we use for large numbers. In 1998 the International Electrotechnical Commission (IEC) defined a new set of abbreviations for the powers of 2, as shown in Table 1.1. These new terms are endorsed by the Institute of Electrical and Electronics Engineers (IEEE) and International Committee for Weights and Measures (CIPM) in situations where the use of a binary prefix is appropriate. The confusion arises over the fact that the mainstream computer industry, such as Microsoft, Apple, and Dell, continues to the old terminology. According to the companies that market to consumers, a 1 GHz is 1,000,000,000 Hz but 1 Gbyte of memory is 1,073,741,824 bytes. The correct terminology is to use the SI-decimal abbreviations to represent powers of 10, and the IEC-binary abbreviations to represent powers of 2. The scientific meaning of 2 kilovolts is 2000 volts, but 2 kibibytes is the proper way to specify 2048 bytes. The term **kibibyte** is a contraction of kilo binary byte and is a unit of information or computer storage, abbreviated KiB.

$$1 \text{ KiB} = 2^{10} \text{ bytes} = 1024 \text{ bytes}$$

$$1 \text{ MiB} = 2^{20} \text{ bytes} = 1,048,576 \text{ bytes}$$

$$1 \text{ GiB} = 2^{30} \text{ bytes} = 1,073,741,824 \text{ bytes}$$

These abbreviations can also be used to specify the number of binary bits. The term **kibibit** is a contraction of kilo binary bit, and is a unit of information or computer storage, abbreviated Kibit.

$$1 \text{ Kibit} = 2^{10} \text{ bits} = 1024 \text{ bits}$$

$$1 \text{ Mibit} = 2^{20} \text{ bits} = 1,048,576 \text{ bits}$$

$$1 \text{ Gibit} = 2^{30} \text{ bits} = 1,073,741,824 \text{ bits}$$

A **mebibyte** (1 MiB is 1,048,576 bytes) is approximately equal to a megabyte (1 MB is 1,000,000 bytes), but mistaking the two has nonetheless led to confusion and even legal disputes. In the engineering community, it is appropriate to use terms that have a clear and unambiguous meaning.

Value	SI	Decimal	Value	IEC	Binary
1000^1	k	kilo-	1024^1	Ki	kibi-
1000^2	M	mega-	1024^2	Mi	mebi-
1000^3	G	giga-	1024^3	Gi	gibi-
1000^4	T	tera-	1024^4	Ti	tebi-
1000^5	P	peta-	1024^5	Pi	pebi-
1000^6	E	exa-	1024^6	Ei	exbi-
1000^7	Z	zetta-	1024^7	Zi	zebi-
1000^8	Y	yotta-	1024^8	Yi	yobi-

Table 1.1. Common abbreviations for large numbers.