Lab 25 Solid-State Disk

This laboratory assignment accompanies the book, Embedded Microcomputer Systems: Real Time Interfacing, by Jonathan W. Valvano, published by Brooks-Cole, copyright © 2000.

Goals
- Interface a static RAM chip (128K by 8-bit) to the 6812,
- Address translation using extended and expanded modes on the MC68HC812A4
- 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 9 on interfacing to the MC68HC812A4 in extended mode,
- Motorola MC68HC812A4 Technical Reference on extended and expanded memory
- Motorola MC68HC812A4 Electrical Specifications on memory bus timing
- Data sheets on the 128K by 8 bit static RAM.

Starter files
- MEMTEST.C

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 large static RAM to the 6812, 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 (>gigabyte) and low cost (<$100 OEM), it is not appropriate for an embedded system, because it 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 Kbyte disk that costs $1 to $4. The cost/bit is about $30/Mbyte. Compare that to a 20-Gbyte hard drive that cost about $100. This cost/bit is $0.005/Mbyte.

There is a great deal of flexibility in the implementation of this lab, but the following requirements must be satisfied. You need to use MC68HC812A4 extended mode address translation using CSD. Every byte of the external 128K RAM must be accessible. The logical block size used by your software must be different from the physical 4K data page size used by the 6812. There must be a low-level device driver, and this software alone can directly access the external RAM. 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.) There must be three software layers including a high-level command interpreter (e.g., a `Lab25.c` file with the main 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 external RAM. Assuming the external RAM had battery-backup power, and the 6812 were to loose power, then all information would be accessible when power is restored to the 6812. You do not need to implement battery backup hardware. If you were to need battery backup, you would simply use a battery-backup RAM chip and your exact system would be the same. If you are interested search the Dallas Semiconductor web site.

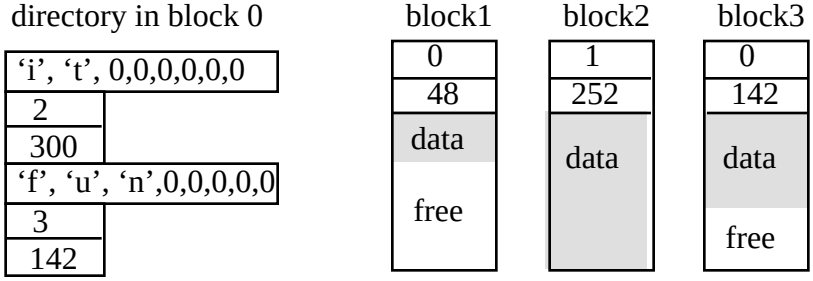
Some of the issues that you can select are as follows. You are free to change the specific names and parameter formats for the software routines. The specific function prototypes are included for illustration purposes. You are free to change and function names and parameters, as long as the basic operations are supported. You can place the functions in whichever module you think is appropriate, as long as there are three layers. You may choose any size for the fixed-size disk blocks, except 4096. You may implement any directory structure. 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 128K by 8-bit static RAM to the 6812 using extended mode addressing. The built-in CSD decoder and extended mode logic will greatly simplify interfacing the external memory to the 6812.

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 external memory.) There should be separate **Disk.c** and **Disk.h** files containing software that implements the low-level disk operations. The following prototypes illustrate these operations. The `Disk_Open` command should test your external memory, making sure all bytes are operational. All routines return a 1 if successful and a 0 on failure.

```
int Disk_Open(void);                        // initialize solid-state disk
int Disk_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 Disk_ReadWord(unsigned short *wordPt, // result returned by reference
  unsigned short blockNum,                  // which block to read from
  unsigned short wordNum);                  // which word within this block
int Disk_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 Disk_WriteWord(unsigned short word,   // value to be saved
  unsigned short blockNum,                  // which block to write into
  unsigned short wordNum);                  // which word within this block
```

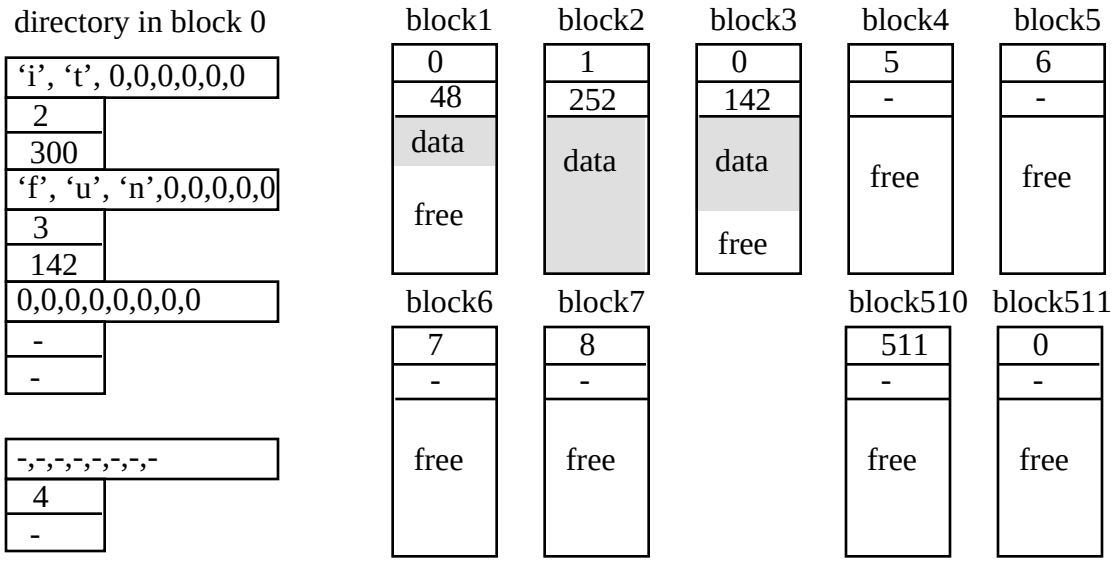
The third step is to develop a file system. There should be separate **File.c** and **File.h** files containing software that implements the file system. There are three components of the file system: directory, allocation, and free-space management. The first component is the **directory**. The directory 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 last 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 contains the file name, the block number of the first block containing data (0 means there is no first block), the block number of the last block containing data (0 means there is no last block), and the total number of bytes stored in the file. A filename is stored as a null-terminated ASCII string in a fixed-size array. A null-string (first byte 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.



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** (again, you are free to develop your own method). 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 following figure assumes the block size is 256 bytes and the filename has a maximum of 7 characters. All counters and pointers have 16-bit precision. Since each data block has a 2-byte link and a 2-byte counter, each block can store up to 252 bytes of data. The file "it" has 300 bytes, 252 of them in block 2 and the rest in block 1. All of the 142 bytes for file "fun" are stored in block 3.

The third component of the file system is **free space management**. Initially all blocks except the one used for the directory are free, available for files to store data. To store data into a file, blocks must be allocated to the file. When a file is deleted, its blocks must be made available again. One simple free space management technique uses **linked allocation**, similar to the way data is stored. You could assign the last directory entry for free space management. This entry is hidden from the user. E.g., this free space file can't be opened, printed, or deleted. It doesn't use any of the byte count fields, but does use the links to access all the free blocks. Initially, all the blocks (except the directory itself) are linked together, with the special directory entry pointing to the first one and the last one having a null pointer. When a file requests a block, it is unlinked from the free space and linked to the file. When a file is deleted, all its blocks are linked to the free space again.



The following prototypes illustrate the operations to be performed by the file system. It is OK to make copies of pointers and counters into regular memory while a command is being executed. But after each operation, all counters and pointers should be written back onto the disk. Similarly it is OK for the `Open` command to make copies of pointers and counters to be used by the `Write` command, but the `Close` command should leave the disk in a consistent state (ready for a power loss). All routines return a 1 if successful and a 0 on failure.

```
int File_Format(void);                    // initialize file system
int File_Create(unsigned char name[8]);   // create new file, make it empty
int File_Open(unsigned char name[8]);     // open a file for appending
int File_Write(unsigned char byte);       // save at end of the open file
int File_Close(void);                     // close the file
int File_Print(unsigned char name[8]);    // print entire contents
int File_Directory(void);                 // print directory contents
int File_Delete(unsigned char name[8]);   // delete this file
```

`File_Format` should return an error if the call to the low-level `Disk_Open` returns an error. `File_Create` should return an error if the directory is full, or if the file already exists. `File_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. `File_Write` should return an error if the disk is full, or if no file is open. `File_Close` should return an error if no file is open. `File_Print` and `File_Delete` should return an error if the file doesn't exist.

The fourth step is to develop a simple interpreter that illustrates the features of your file system. This software includes the main program, which first initializes, then implements the interpreter. 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 mine`	create a new empty file called "`mine`"
`p yours`	display the contents of file "`yours`"
`a hers`	open the file "`hers`", and add characters subsequently typed characters are added to the file close the file and return to the interpreter when an <esc> is typed ($1B)
`e his`	erase file "`his`"

Preparation (do this before your lab period)

1: You will draw the circuit that interfaces your memory to the 6812. You should use the CSD, which is a built-in address decoder of the MC68HC812A4. Draw a logic diagram for the memory system including pin numbers for all IC's. Show all connections to the MC68HC812A4 bus. On your circuit diagram, label both the 6812 pin name and the Adapt812 H2 connector pin number. The memory bus is available on the Adapt812 H2 connector:

- 6812 Ports A, B contain the 16-bit address A15-A0,
- 6812 Port G contains the 4-bit address A19-A16,
- 6812 Port C contains the bidirectional 8-bit data bus,
- 6812 Port E contains the E clock and R/W,
- 6812 Port F contains the built-in chips selects.

In narrow mode the data is on PORTC, but is labeled as D15-D8 on the Adapt812.

2: Choose the correct number of cycle stretches so that Read Data Available overlaps Read Data Required and that Write Data Available overlaps Write Data Required. By using the built-in chip select, CSD, you will be synchronizing the memory control signals with the E clock. Draw the combined READ timing diagram and draw the combined WRITE timing diagram, as described in Chapter 9 of the book. If you can find a data sheet for your memory chip, then calculate the correct number of cycles, if no data sheet is available, start with 3 stretches and experimentally test it once it is built.

3: Next, write the low-level **Disk.c** and **Disk.h** files. The `Disk_Open` function should test the entire 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 second 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 **Disk.c**, to test the external memory and all the functions in **Disk.c**. Report any errors to the screen.

5: Next, write the prototype for the file system **File.h**. The implementation file, **File.c**, is not required as part of the preparation. Instead, **File.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, and you only have 4K total of program space.

Procedure (do this during your lab period)

1: Connect the memory system to the microprocessor using the address and data busses and the appropriate memory control signals. In expanded narrow mode, Port C is the data bus (labeled D15-D8) on the H2 connector. The jumpers for MODA and MODB on the Adapt812 will be left at their usual settings (0,0). To download and run a program in special expanded narrow mode, follow these steps:

1) Within ICC12 edit, compile the program using the regular single chip segmentation
   - RAM at 0x0800-0xBFF (globals start at 0x0800, SP initialized to 0x0C00)
   - EEPROM at 0xF000-0xFFFF
2) Power up Adapt812/BDM, and start BDM software
3) Execute **firm**, **reset**, **b** until 6812 is halted (**r** command works) and the **status** returns **C0**
4) Verify the processor is in special mode
   `db 000b` should give **1B** for *special single chip mode*
   (your program will change it to **33** *special expanded narrow mode*)
5) Verify the EEPROM is at $F000-$FFFF
   `db 0012` should be F1
6) Download your S19 record in the usual way
7) Start your program from the debugger (do not hit the reset button[1])
   `g f000`

[1] Hitting the reset button will place the processor in normal single chip mode. You need special mode.

2: Next, test your interface with the `MemTest2` program (you could also configure the `MODE=0x3B;` to see internal memory accesses on the external bus). This program does not run in extended mode, but it should still be able to test the first 4K of your memory.
3: Test your software written in preparation parts 3 and 4. After this step, you should be confident you memory system is operational.
4: Now that you have the knowledge of how the memory works, implement the file system.
5: 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. After the file system is tested, discard these "bad" files.

Checkout (show this to the TA)

You should be able to demonstrate to the TA at least 3 files, at least of them have more than one block, and at least one of them has 3 blocks. Demonstrate each of the interpreter commands.

Hints

1) Please don't put broken RAM chips back in your bag. Mark bad chips as destroyed.
2) Smaller block sizes will be less efficient, but easier to test.
3) To save program space, start with SCI12.c instead of SCI12A.c, then make a version of SCI12.C that includes only those functions you need. Rename the reduced SCI12.c filename, so you don't mix it up with the full version.

```
/* filename  MemTest2.C  Jonathan W. Valvano,
   External 4K RAM from 0x7000 to 0x7FFF*/
#include "HC12.h"
#include "SCI12.h"
void main(void){ unsigned short addr;
unsigned short numErrors=0;
  InitSCI();
  COPCTL = 0;      // disable COP
  MODE   = 0x33;   // special exp narrow
  MODE   = 0x33;   // special exp narrow, you have to write it twice, second time works
/* bit 765 001  Special expanded narrow
   bit 4 ESTR=1  E stretch during external
   bit 3 IVIS=0  No Internal bus activity visible
   bit 1 EMD =1  Port D access as external device
   bit 0 EME =1  Port E access as external device */
  PEAR = 0x2C;
/* bit 7 ARSIE=0  PE7 is general I/O
   bit 6 PLLTE=0  no PPL test
   bit 5 PIPOE=1  enable pipe
   bit 4 NECLK=0  show E
   bit 3 LSTRE=1  low strobe enable
   bit 2 RDWE =1  R/W enable */
  CSCTL0 = 0x10;  // enable CSD
  CSCTL1 = 0;     // $7000 to $7FFF
  CSSTR0 = 0x3F;  // stretch 3 E clocks
  for(addr = 0x7000; addr <= 0x7FFF; addr++){
    *(char*)addr = (char)addr; // store low address into memory
  }
  for(addr = 0x7000; addr <= 0x7FFF; addr++){
    if((char)addr != *(char*)addr){ // match previous write?
      numErrors++;
    }
  }
  OutString("Number of errors = ");
  OutUDec(numErrors);
}
#include "SCI12.c"
// no vectors because program started from BDM
```

look at `http://www.ece.utexas.edu/~valvano/programs/k1008ce.pdf`
Search also for 1Mbit SRAM at `http://www.samsungelectronics.com/semiconductors/SRAM/`