EE445M Lab 5 Report

Yen-Kai Huang
Siavash Zanganeh Kamali

March 28, 2014

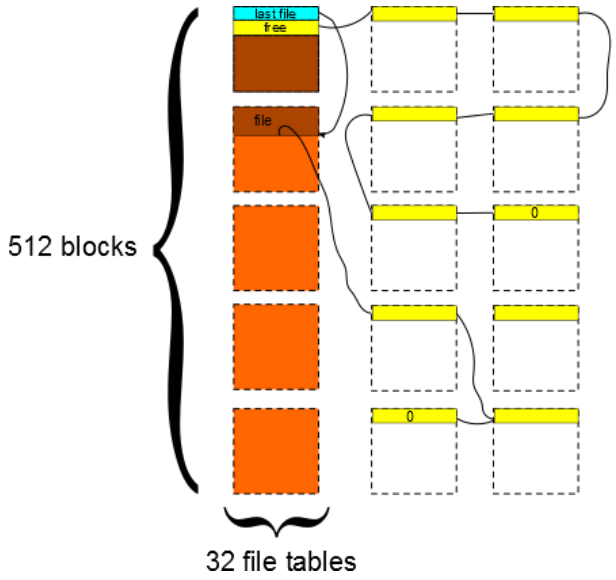
1 Objective

The goal of this lab is to build a working file system. In this lab we will use low-level library code to interface a SD card, and write the program that serves as a middle-level interface, with directory access and interpreter commands for easy use.

As an application of the file system, we will stream the debugging information of a robot onto the SD card.

2 Software Design

(a) Here is the picture showing the file system scheme we use. It is a linked-list structure with 32 blocks dedicated to store the directory information.



The *White* blocks represent file blocks that store actual data. Each of these data blocks has a header that stores the address of the next blocks in a the same file. If a block is the last in a file this value is `null`.

The *Orange* and *Brown* parts represent file tables that store the file names, start/end addresses, and size of each file. The first of these file table is special. The first `short` value is an index of first free spot or the last file among the file tables. The yellow part *free* is a directory object storing the start/end address of the free blocks.

(b) Middle-level File system code Below are the file system codes implementing the scheme shown in (a)

```
// filename ************** eFile.h ****************************
// Middle-level routines to implement a solid-state disk
// Jonathan W. Valvano 3/16/11

#define EOF    0

//---------- eFile_Init-----------------
// Activate the file system, without formating
// Input: none
// Output: 0 if successful and 1 on failure (already initialized)
// since this program initializes the disk, it must run with
//    the disk periodic task operating
int eFile_Init(void); // initialize file system

//---------- eFile_Format-----------------
// Erase all files, create blank directory, initialize free space
   manager
// Input: none
// Output: 0 if successful and 1 on failure (e.g., trouble writing to
   flash)
int eFile_Format(void); // erase disk, add format

//---------- eFile_Create-----------------
// Create a new, empty file with one allocated block
// Input: file name is an ASCII string up to seven characters
// Output: 0 if successful and 1 on failure (e.g., trouble writing to
   flash)
int eFile_Create( char name[]);  // create new file, make it empty


//---------- eFile_WOpen-----------------
// Open the file, read into RAM last block
// Input: file name is a single ASCII letter
// Output: 0 if successful and 1 on failure (e.g., trouble writing to
   flash)
int eFile_WOpen(char name[]);      // open a file for writing

//---------- eFile_Write-----------------
// save at end of the open file
// Input: data to be saved
// Output: 0 if successful and 1 on failure (e.g., trouble writing to
   flash)
int eFile_Write( char data);

//---------- eFile_Close-----------------
// Deactivate the file system
// Input: none
// Output: 0 if successful and 1 on failure (not currently open)
int eFile_Close(void);


//---------- eFile_WClose-----------------
// close the file, left disk in a state power can be removed
// Input: none
// Output: 0 if successful and 1 on failure (e.g., trouble writing to
```

```
   flash)
int eFile_WClose(void); // close the file for writing

//---------- eFile_ROpen----------------
// Open the file, read first block into RAM
// Input: file name is a single ASCII letter
// Output: 0 if successful and 1 on failure (e.g., trouble read to
   flash)
int eFile_ROpen( char name[]);      // open a file for reading

//---------- eFile_ReadNext----------------
// retreive data from open file
// Input: none
// Output: return by reference data
//         0 if successful and 1 on failure (e.g., end of file)
int eFile_ReadNext( char *pt);       // get next byte

//---------- eFile_RClose----------------
// close the reading file
// Input: none
// Output: 0 if successful and 1 on failure (e.g., wasn't open)
int eFile_RClose(void); // close the file for writing

//---------- eFile_Directory----------------
// Display the directory with filenames and sizes
// Input: pointer to a function that outputs ASCII characters to
   display
// Output: characters returned by reference
//         0 if successful and 1 on failure (e.g., trouble reading from
    flash)
int eFile_Directory(char *filelist);

//---------- eFile_Delete----------------
// delete this file
// Input: file name is a single ASCII letter
// Output: 0 if successful and 1 on failure (e.g., trouble writing to
   flash)
int eFile_Delete( char name[]);  // remove this file

//---------- eFile_RedirectToFile----------------
// open a file for writing
// Input: file name is a single ASCII letter
// stream printf data into file
// Output: 0 if successful and 1 on failure (e.g., trouble read/write
   to flash)
int eFile_RedirectToFile(char *name);

//---------- eFile_EndRedirectToFile----------------
// close the previously open file
// redirect printf data back to UART
// Output: 0 if successful and 1 on failure (e.g., wasn't open)
int eFile_EndRedirectToFile(void);
```

Code/efile.h

```
// filename ************** eFile.h ****************************
```

```
// Middle-level routines to implement a solid-state disk
// Jonathan W. Valvano 3/16/11

#include "efile.h"
#include "edisk.h"

/****************************** Global ******************************/
#define NO_FILE_OPEN 0
#define FILE_ROPEN   1
#define FILE_WOPEN   2
unsigned char FILE_OPEN = NO_FILE_OPEN;

// 1 if redirecting, 0 otherwise
unsigned char Redirect_stream;

/************************** Data Structure **********************/

#define MAXFILENAME     2 /* bytes */
typedef struct {
  char filename[MAXFILENAME];
  short startAddr; // 2 bytes
  short endAddr;   // 2 bytes
  short size;      // 2 bytes
} DirType;           // 8 bytes

// Definition of SD card space
#define MAXBLOCK            2048 /* 1 MiB */

#define MAXFILETABLE        32
#define START_OF_DATA_BLOCK MAXFILETABLE
#define END_OF_DATA_BLOCK   MAXBLOCK

/* The first table block with the size 8 * 64 = 512 bytes
 * the inititialization function will fill this with the first 64 files
    in SD card */
#define FREE_SPACE                0
#define NumDirTableEntry          64
#define NumDirTableEntry_One      (NumDirTableEntry-2)

static void longToString (char * buffer, long value);

struct ft1struct {
  short firstFree;                              /*   2 bytes */
  char __wastedspace__2[2];                     /*   2 bytes wasted */
  DirType free;                                 /*   8 bytes */
  DirType datablock[NumDirTableEntry_One];      /* 496 bytes = 62*8 */
  char __wastedspace__4[4];                     /*   4 bytes wasted */
} Filetable_One;                                  /* 512 bytes */

/* a buffer for the dir entry */
char TableNum;
char EntryNum; /* in the abstract filetable */
DirType DirBuf;

/* a buffer for the data block in RAM */
char BlockNum; /* in the physical SD card */
```

```
57 BYTE BlockBuf[512];
58 char* BlockPt = (char *) &BlockBuf[sizeof(short)];
59 #define Block_NextAddr (((short *) BlockBuf)[0])
60
61
62 /* the SD card can contain a maximum of 2^11 files, which are addressed
       in 32 file tables
63  * the first table is loaded into RAM upon initialization to accelerate
       the process */
64
65 /****************\****** Static Functions **********\***************/
66 // Function to find the named file in entries
67 // Input: name - a string of name, note that it must be filled up to 4
       characters long with spaces
68 //        wd - working directory, passed by reference to store the dir
       info
69 //        entrynum - passed by reference to store the index of the file
       in filetable
70 // Output: -1 if failed to find the file (file does not exist or fail
       to read from SD card)
71 //         otherwise returns the table at which it finds the file
72 // When terminating, the filetable is in BlockBuf or Filetable_One
73 // Only **reads** the filetables
74 char openfile(const char *name, DirType *wd, char *entrynum);
75
76 // function to save a directory into a filetable
77 // Input: wd - reference to the working directory
78 //        wd_at_Table - index of the table
79 //        entrynum - index of the file in filetable
80 // Output: 0 if successful and 1 on failure (fails to write)
81 int savefile(const DirType *wd, char wd_at_Table, char entrynum);
82
83 // returns 0 if equal, 1 if not
84 char my_strcmp2(const char *s, const char *t);
85
86 int eFile_Init(void) { // initialize file system
87   if(eDisk_Init(0)) return 1;;
88
89   eDisk_ReadBlock((BYTE *) &Filetable_One, 0);
90
91   return 0;
92 }
93
94 int eFile_Format(void) { // erase disk, add format
95   // Clear file tables
96   short i;
97   char result = 0;
98
99   for(i = 0; i < 512; i++) {
100    BlockBuf[i] = 0;
101  }
102
103  for(i = 1; i < MAXFILETABLE; i++) {
104    result |= eDisk_WriteBlock(BlockBuf, i);
105  }
106
```

```
107   // Fails to write the SD card
108   if(result) return 1;
109   /* Clear the data blocks in the first filetable */ {
110     char *name = DirBuf.filename;
111     // Clear the filename, using the right associativity of = operator
112     *name = *name++ = 0;
113 
114     DirBuf.startAddr = DirBuf.endAddr = 0;
115     DirBuf.size = 0;
116 
117     for(i = 0; i < NumDirTableEntry_One; i++) {
118       Filetable_One.datablock[i] = DirBuf;
119     }
120   }
121 
122   // The name of free space block doesn't matter
123   Filetable_One.free.startAddr = START_OF_DATA_BLOCK;
124   Filetable_One.free.endAddr = END_OF_DATA_BLOCK-1;
125 
126   Filetable_One.firstFree = 0;
127 
128   eDisk_WriteBlock((BYTE *) &Filetable_One, 0);
129 
130   // Link data blocks to each other
131   for(i = START_OF_DATA_BLOCK; i < END_OF_DATA_BLOCK; i++) {
132     Block_NextAddr = i+1; // the first short of each data block
      contains the address to the next block
133     result |= eDisk_WriteBlock(BlockBuf, i);
134   }
135 
136   FILE_OPEN = NO_FILE_OPEN;
137 
138   return result;
139 }
140 
141 int eFile_Create(char *name) {  // create new file, make it empty
142   DirType *newdir;
143   char entry_at_table = -1;
144 
145   //Checking whether the file already
146   if(openfile(name, &DirBuf, &EntryNum) != -1) {
147     return 3;
148   }
149 
150   // Find an empty entry
151   if(Filetable_One.firstFree < MAXBLOCK) {
152     TableNum = entry_at_table = Filetable_One.firstFree / 64;
153     EntryNum = Filetable_One.firstFree % 64;
154     if(entry_at_table == 0)
155       newdir = &Filetable_One.datablock[EntryNum];
156     else {
157       if(eDisk_ReadBlock(BlockBuf, entry_at_table)) return 1;
158       newdir = (DirType *) &BlockBuf[EntryNum];
159     }
160   } else { // All table are filled up
161     return 1;
```

```
162   }
163
164   /* Create a dir entry in Filetable */ {
165     unsigned char counter;
166
167     for(counter = 0; counter < 2; counter++) {
168       if(*name) {
169         newdir->filename[counter] = *name++;
170       } else {
171         newdir->filename[counter] = ' ';
172       }
173     }
174
175     BlockNum = newdir->startAddr    \
176       = newdir->endAddr \
177       = Filetable_One.free.startAddr;
178
179     newdir->size = 1;
180
181     // write back to the SD card copy
182     if(entry_at_table == 0) {
183       if(eDisk_WriteBlock((BYTE *) &Filetable_One, 0)) return 1;
184     } else {
185       if(eDisk_WriteBlock((BYTE *) BlockBuf, entry_at_table)) return 1;
186     }
187
188     DirBuf = *newdir; // a readonly copy for reference
189   }
190
191   if(eDisk_ReadBlock((BYTE *) BlockBuf, DirBuf.startAddr)) return 1;
192   // Unlink the block from free space list
193   if(Filetable_One.free.startAddr == Filetable_One.free.endAddr) return
        2; // SD card is full
194
195   Filetable_One.free.startAddr = Block_NextAddr;
196   Block_NextAddr = 0;
197
198   // Prepare for write operation
199   BlockPt = (char *) &BlockBuf[sizeof(short)];
200   *BlockPt = EOF;
201
202   // !! Commiting here is kinf od optional
203   if(eDisk_WriteBlock((BYTE *) BlockBuf, DirBuf.startAddr)) return 1;
204
205   // increment first free entry
206   // It is safe to assume contiguous allocation, when delete I will
        always fill up the space with the globally last entry.
207   // Because the first filetable is not full (only 62 blocks), for
        convenience I skip 62, 63 in the firstFree value
208   // This ensures that I can always take / 64 as the table number and %
        64 as the entry number
209   if(++Filetable_One.firstFree == NumDirTableEntry_One) Filetable_One.
        firstFree = 64;
210
211   if(eDisk_WriteBlock((BYTE *) &Filetable_One, 0)) return 1;
212
```

```
213   FILE_OPEN = FILE_WOPEN;
214
215   return 0;
216 }
217
218 int eFile_Close(void) {
219   if(!FILE_OPEN) return 1;
220
221   // save any opened files
222   // savefile may possibly overwrite BlockBuf, therefore it needs to be
        called last
223   if(eDisk_WriteBlock((BYTE *) BlockBuf, BlockNum)) return 1;
224   savefile(&DirBuf, TableNum, EntryNum);
225
226   FILE_OPEN = NO_FILE_OPEN;
227
228   return 0;
229 }
230
231 int eFile_WOpen(char name[]) {      // open a file for writing
232   if(FILE_OPEN) return 1;          // cannot open multiple files
233
234   if((TableNum = openfile(name, &DirBuf, &EntryNum)) == -1) {
235     return 1;
236   }
237
238   if(eDisk_ReadBlock((BYTE *) BlockBuf, BlockNum = DirBuf.endAddr))
       return 1;
239
240   /* Linear search to find where the EOF marker is */
241   while(*BlockPt != EOF && BlockPt < (char *) &BlockBuf[512]) ++BlockPt
       ;
242
243   if(BlockPt == (char *) &BlockBuf[512]) { // A new block needs to be
       appended
244     // Unlink a block from the free space and link into the file
245     if(Filetable_One.free.startAddr == Filetable_One.free.endAddr)
       return 1; // SD card is full
246
247     Block_NextAddr = Filetable_One.free.startAddr;
248     if(eDisk_WriteBlock((BYTE *) BlockBuf, BlockNum)) return 1;
249     if(eDisk_ReadBlock((BYTE *) BlockBuf, BlockNum = DirBuf.endAddr =
       Block_NextAddr)) return 1;
250     Filetable_One.free.startAddr = Block_NextAddr;
251     Block_NextAddr = 0;
252     DirBuf.size += 1;
253     if(eDisk_WriteBlock((BYTE *) &Filetable_One, 0)) return 1;
254
255     BlockPt = (char *) &BlockBuf[sizeof(short)];
256     *BlockPt = EOF;
257   }
258
259
260   FILE_OPEN = FILE_WOPEN;
261
262   return 0;
```

```
263 }
264
265 int eFile_Write(char data) {
266   if(FILE_OPEN != FILE_WOPEN) return 2; // Can only write in Write mode
267
268   /* We can assume that BlockPt now points at a the EOF marker, one
       place beyond all written characters
269    * the first write will overwrite the EOF marker. And when file is
       closed the EOF marker will be place
270    * at one place beyond. Therefore, when data is equal to EOF it will
       mess up the file and should be forbidden. */
271   if(data == EOF) return 3; // illegal input
272
273   if(BlockPt == (char *) &BlockBuf[512]) { // Append a new block
274     // Unlink a block from the free space and link into the file
275     if(Filetable_One.free.startAddr == Filetable_One.free.endAddr)
      return 4; // SD card is full
276
277     Block_NextAddr = Filetable_One.free.startAddr;
278     if(eDisk_WriteBlock((BYTE *) BlockBuf, BlockNum)) return 1;
279     if(eDisk_ReadBlock((BYTE *) BlockBuf, BlockNum = DirBuf.endAddr =
      Block_NextAddr)) return 1;
280     Filetable_One.free.startAddr = Block_NextAddr;
281     Block_NextAddr = 0;
282     DirBuf.size += 1;
283     if(eDisk_WriteBlock((BYTE *) &Filetable_One, 0)) return 1;
284
285     BlockPt = (char *) &BlockBuf[sizeof(short)];
286   }
287
288   *BlockPt++ = data;
289
290   return 0;
291 }
292
293 int eFile_WClose(void) { // close the file for writing
294   if(!FILE_WOPEN) return 2;
295
296   // If the BlockPt is at the end of a file then skip the EOF marker
297   // When the file is opened next time a new block should be appended
298   if(BlockPt < (char *) &BlockBuf[512]) {
299     *BlockPt = EOF;
300   }
301
302   // save any opened files
303   // savefile may possibly overwrite BlockBuf, therefore it needs to be
       called last
304   if(eDisk_WriteBlock((BYTE *) BlockBuf, BlockNum)) return 1;
305   savefile(&DirBuf, TableNum, EntryNum);
306
307   FILE_OPEN = NO_FILE_OPEN;
308
309   return 0;
310 }
311
312 int eFile_ROpen( char name[]) {      // open a file for reading
```

```
313   if(FILE_OPEN) return 2; // cannot open multiple files
314
315   if((TableNum = openfile(name, &DirBuf, &EntryNum)) == -1) {
316     return 3;
317   }
318
319   if(eDisk_ReadBlock((BYTE *) BlockBuf, BlockNum = DirBuf.startAddr))
       return 1;
320   BlockPt = (char *) &BlockBuf[sizeof(short)];  // the first (short) is
        the next pointer
321
322   FILE_OPEN = FILE_ROPEN;
323
324   return 0;
325 }
326
327 int eFile_ReadNext(char *pt) {       // get next byte
328   if(FILE_OPEN != FILE_ROPEN) {   // Can only read in read mode
329     return 2;
330   }
331
332   if(*BlockPt == EOF) {
333     return 3;
334   }
335
336   if(BlockPt == (char *) &BlockBuf[512]) {
337     // Because we always open a new block for reading, the pointer is
       beyond the end of block
338     // only we are the end of file.
339       return 3;
340   }
341
342   *pt = *BlockPt++;
343
344   if(BlockPt == (char *) &BlockBuf[512]) {
345     if(Block_NextAddr == 0) { // Reached end of file
346       return 0;
347     }
348     else {  // Read next block
349       if(eDisk_ReadBlock((BYTE *) BlockBuf, Block_NextAddr)) return 1;
350       BlockPt = (char *) &BlockBuf[sizeof(short)];
351
352       if(*BlockPt == EOF) {
353         return 3;
354       }
355     }
356   }
357
358   return 0;
359 }
360
361 int eFile_RClose(void) { // close the file for writing
362   if(FILE_OPEN != FILE_ROPEN) return 1;
363
364   FILE_OPEN = NO_FILE_OPEN;
365
```

```
366   return 0;
367 }
368
369 int eFile_Directory(char *filelist){
370   int i,j=0,k,l;
371   char buffer[8];
372   for (i=0; i<NumDirTableEntry-2; i++ ) {
373     if (!Filetable_One.datablock[i].filename[0]){
374       return 0;
375     }
376     for (k=0;k<2;k++) {
377       filelist[j++] = Filetable_One.datablock[i].filename[k];
378     }
379     longToString(buffer, (long)(Filetable_One.datablock[i].size)*512);
380     filelist[j++] = ' ';
381     for (k=0; buffer[k]; k++) {
382       filelist[j++] = buffer[k];
383     }
384     filelist[j++] = ' ';
385     filelist[j++] = ' ';
386     filelist[j++] = 'B';
387     filelist[j++] = '\r';
388     filelist[j++] = '\n';
389   }
390   for (l=1;l<MAXFILETABLE; l++ ) {
391     DirType *DirBuffer;
392     if(eDisk_WriteBlock(BlockBuf, l)) return 1;
393     DirBuffer = (DirType *) (BlockBuf);
394     for (i=0; i<NumDirTableEntry; i++ ) {
395       if (!DirBuffer[i].filename[0]){
396         return 0;
397       }
398       for (k=0;k<2;k++) {
399         filelist[j++] = DirBuffer[i].filename[k];
400       }
401       longToString(buffer, (long)(Filetable_One.datablock[i].size)*512)
    ;
402       filelist[j++] = ' ';
403       for (k=0; buffer[k]; k++) {
404         filelist[j++] = buffer[k];
405       }
406       filelist[j++] = ' ';
407       filelist[j++] = 'B';
408       filelist[j++] = '\r';
409       filelist[j++] = '\n';
410     }
411   }
412   return 0;
413 }
414
415 int eFile_Delete(char name[]) { // remove this file
416   DirType dir;
417   DirType repdir;
418   DirType *reppt;
419   char wd_at_Table, entrynum;
420   char replace_dir_at_Table;
```

```
421   /* 62 and 63 are skipped in the first table */
422   short lastDir = (Filetable_One.firstFree==64)? 61 : Filetable_One.
       firstFree-1;
423
424   // Trivial case there is no file to delete
425   if(Filetable_One.firstFree == 0) return 2;
426
427   //Close any file that is open
428   switch(FILE_OPEN) {
429     case FILE_WOPEN:
430       eFile_WClose();
431       break;
432     case FILE_ROPEN:
433       FILE_OPEN = NO_FILE_OPEN;
434       break;
435     default:
436       break;
437   }
438
439   /* find the globally last file for replacement */  {
440
441   /* Because the allocation is contiguous it can be easily found by */
442     replace_dir_at_Table = lastDir / 64;
443     if(replace_dir_at_Table == 0) {
444       reppt = &Filetable_One.datablock[lastDir % 64];
445     } else {
446       if(eDisk_ReadBlock((BYTE *) BlockBuf, replace_dir_at_Table))
       return 1;
447       reppt = (DirType *) &BlockBuf[lastDir % 64];
448     }
449
450     repdir = *reppt;
451   }
452
453   // find the file to delete, this will change the BlockBuf
454   if((wd_at_Table = openfile(name, &dir, &entrynum)) != -1) {
455   // now the information of the file is in the dir
456     if(wd_at_Table == 0) {
457       Filetable_One.datablock[entrynum] = repdir;
458       if(eDisk_WriteBlock((BYTE *) &Filetable_One, 0)) return 1;
459     } else {
460       // Then the filetable should be in the BlockBuf
461       ((DirType *) BlockBuf)[entrynum] = repdir;
462       if(eDisk_WriteBlock((BYTE *) BlockBuf, wd_at_Table)) return 1;
463     }
464
465
466   // Link all the data block of the file into the free space list
467     if(eDisk_ReadBlock((BYTE *) BlockBuf, dir.endAddr)) return 1;
468     Block_NextAddr = Filetable_One.free.startAddr;
469     if(eDisk_WriteBlock((BYTE *) BlockBuf, dir.endAddr)) return 1;
470     Filetable_One.free.startAddr = dir.startAddr;
471
472   } else return 2; // no such file was found, at this point the last
       filetable is in the BlockBuf
473
```

```
474
475   /* clear the replace file */ {
476     char *name = reppt->filename;
477
478     // Because the previous step changed the BlockBuf, it may be
      necessary to reload it
479     if(replace_dir_at_Table > 0)
480       if(eDisk_ReadBlock((BYTE *) BlockBuf, replace_dir_at_Table))
      return 1;
481     // note that even though the BlockBuf is unloaded and reloaded,
      reppt still points to the same address
482
483     // Clear the filename, using the right associativity of = operator
484     *name = *name++ = 0;
485
486     reppt->startAddr = reppt->endAddr = 0;
487
488     Filetable_One.firstFree = lastDir;
489   }
490
491   // Commit changes to the replacement and/or First table
492   if(eDisk_WriteBlock((BYTE *) &Filetable_One, 0)) return 1;
493   if(replace_dir_at_Table > 0)
494     if(eDisk_WriteBlock((BYTE *) BlockBuf, replace_dir_at_Table))
      return 1;
495
496   return 0;
497 }
498
499 int eFile_RedirectToFile(char *name) {
500   if(eFile_WOpen(name) == 0) {
501     Redirect_stream = 1;
502   } else return 1;
503
504   return 0;
505 }
506
507 int eFile_EndRedirectToFile(void) {
508   // Clear redirecting flag, it wouldn't hurt if no file is open
509   Redirect_stream = 0;
510
511   return eFile_Close();
512 }
513
514 /******************************** static
      ********************************/
515 char openfile(const char *name, DirType *wd, char *entrynum) {
516   DirType *d;
517   char lastIndex, lastPage;
518
519   // Trivial case: there is no file to open
520   if(Filetable_One.firstFree == 0) return -1;
521
522   lastIndex = Filetable_One.firstFree % 64; // one beyond
523   // However, if the lastIndex is 0, then the last page is in fact
      empty
```

```
524   lastPage = Filetable_One.firstFree / 64 - (lastIndex == 0); // last
       real one
525   // Ensure lastIndex is one beyond
526   if(lastIndex == 0) lastIndex = 64;
527
528   /* look for the file in Filetable_One */ {
529     char FirstPageLimit = (lastPage == 0)? lastIndex : 64; // one
       beyond
530
531     for(d = Filetable_One.datablock; d < &Filetable_One.datablock[
       FirstPageLimit]; d++) {
532       if(my_strcmp2(d->filename, name) == 0) {
533         *entrynum = (char) (d - Filetable_One.datablock);
534         *wd = *d;
535         return 0;
536       }
537     }
538
539     // this ensures the remainder part below doesn't do unnecessary
       work
540     if(lastPage == 0) return -1;
541   }
542
543   /* look for the file in all the other (non-empty) filetables */ {
544     char i;
545
546     // Full pages if any
547     for(i = 1; i < lastPage; i++) {
548       if(eDisk_ReadBlock((BYTE *) BlockBuf, i)) {
549         return *entrynum = -1;
550       }
551
552       for(d = (DirType *) BlockBuf; d < &((DirType *) BlockBuf)[64]; d
       ++) {
553         if(my_strcmp2(d->filename, name) == 0) {
554           *entrynum = (char) (d - (DirType *) BlockBuf);
555           *wd = *d;
556           return i;
557         }
558       }
559     }
560
561     // The remainder
562     // This will not repeat the work of when lastPage = 0
563     if(eDisk_ReadBlock((BYTE *) BlockBuf, lastPage)) {
564         return *entrynum = -1;
565     }
566     for(d = (DirType *) BlockBuf; d < &((DirType *) BlockBuf)[lastIndex
       ]; d++) {
567       if(my_strcmp2(d->filename, name) == 0) {
568         *entrynum = (char) (d - (DirType *) BlockBuf);
569         *wd = *d;
570         return i;
571       }
572     }
573
```

```
574   }
575
576   // Still cannot find
577   return *entrynum = -1;
578 }
579
580 int savefile(const DirType *wd, char wd_at_Table, char entrynum) {
581   if(wd_at_Table == 0) {
582     Filetable_One.datablock[entrynum] = *wd;
583     if(eDisk_WriteBlock((BYTE *) &Filetable_One, 0)) return 1;
584   } else {
585     if(eDisk_ReadBlock((BYTE *) BlockBuf, wd_at_Table)) return 1;
586     ((DirType *)BlockBuf)[entrynum] = *wd;
587     if(eDisk_WriteBlock((BYTE *) BlockBuf, wd_at_Table)) return 1;
588   }
589
590   return 0;
591 }
592
593 // returns 0 if equal, 1 if not
594 char my_strcmp2(const char *s, const char *t) {
595   char result = (*s++ != *t++);
596   result |= (*s != *t);
597
598   return result;
599 }
```

Code/efile.c

Below is the interpreter commands for easier control of the file system.

```
1  // interpreter.c
2  // written By Siavash Zangeneh * Nicholas Huang
3  // Functions that the interpreter uses to parse Input strings and
       execute the appropriate functions
4
5  #include "io.h"
6  #include "UART2.h"
7  #include <string.h>
8  #include "efile.h"
9
10 /************************** Private Functions
       ******************************/
11 static void Interpreter_ParseInput(char *input);
12 static char *getline(char *str, unsigned short length);
13
14 //Structure that is used to create the table that holds all the
       commands
15 typedef struct {
16   char command[10];
17   void(*functionPt) (void);
18 } commandTable;
19
20 // Commands
21 static void parseLCDCommand (void);
22 static void helpList (void);
23
```

```
24 //FileSystem commands
25 static void parseRemoveCommand(void);
26 static void parseDirectoryCommand(void);
27 static void parseFormatCommand(void);
28 static void parseNewCommand(void);
29 static void parseEditCommand(void);
30 static void parseCATCommand(void);
31
32 //function protoypes for private functions
33 static long stringToInteger (char *string);
34 static void appendtofile (void);
35
36 //Commands table, edit both the array and element numbers to update the
       table
37 #define NUM_COMMANDS 10
38 static const commandTable Table[NUM_COMMANDS]= {
39   {"lcd", &parseLCDCommand},
40   {"help", &helpList},
41
42   // File system commands
43   {"rm", &parseRemoveCommand},
44   {"pwd", &parseDirectoryCommand},
45   {"format", &parseFormatCommand},
46   {"new", &parseNewCommand},
47   {"edit", &parseEditCommand},
48   {"cat", &parseCATCommand},
49 };
50
51 void Interpreter(void) {
52   char string[50];
53   UART_Init();
54   OS_bWait(&Sema4UART);
55     printf("\r\nWelcome to OS NS ....\r\n");
56   OS_bSignal(&Sema4UART);
57   for(;;) {
58     OS_bWait(&Sema4UART);
59       printf("Enter Command -> ");
60       getline(string, 50);
61       Interpreter_ParseInput(string);
62     OS_bSignal(&Sema4UART);
63   }
64 }
65
66 //Public Function
67 //Description: this function parses the input to the interpreter and
       calls the appropriate function
68 //Input: String that holds the complete input line
69 //Output: None
70 static void Interpreter_ParseInput(char *input) {
71   char* buffer; int i;
72   buffer = strtok(input, " ,");                  // parsing the first
       token
73   for (i=0;i<NUM_COMMANDS;i++) {                 // Iterating through
       the command table to
74     if ( ! strcmp(buffer, Table[i].command)) {
```

```
75        Table[i].functionPt();                    // Calling the
    appropriate function in the table
76        return;
77      }
78    }
79    printf("command is invalid!\r\n");
80    return;
81  }
82
83  const char *fileerrormsg = "Reading/Writing to disk failed\r";
84
85  static void parseRemoveCommand(void){
86    char *buffer;
87
88    const char *msg = "Enter the file name to be deleted as the first
     argument\r";
89
90    OS_bWait(&Sema4FileSystem);
91
92    if (buffer = strtok(NULL, " ")) {
93      if (eFile_Delete(buffer) == 2) {
94        puts("File not found\r");
95      } else if (eFile_Delete(buffer) == 1) {
96        puts(fileerrormsg);
97      }
98    } else{
99      puts(msg);
100     OS_bSignal(&Sema4FileSystem);
101     return;
102   }
103
104   OS_bSignal(&Sema4FileSystem);
105 }
106 static void parseDirectoryCommand(void){
107   char buffer[100];
108
109   OS_bWait(&Sema4FileSystem);
110
111   if (eFile_Directory(buffer)){
112     puts(fileerrormsg);
113     OS_bSignal(&Sema4FileSystem);
114     return;
115   }
116
117   OS_bSignal(&Sema4FileSystem);
118
119   puts(buffer);
120 }
121
122 static void parseFormatCommand(void){
123   OS_bWait(&Sema4FileSystem);
124
125   if (eFile_Format()) {
126     puts(fileerrormsg);
127   }
128
```

```
129   OS_bSignal(&Sema4FileSystem);
130 }
131
132 static void parseNewCommand(void){
133   char *buffer;
134
135   const char *msg = "Enter the file name to be created as the first
       argument\r";
136
137   OS_bWait(&Sema4FileSystem);
138
139   if (buffer = strtok(NULL, " ")) {
140     switch (eFile_Create(buffer) ) {
141       case 1:
142         puts(fileerrormsg);
143       OS_bSignal(&Sema4FileSystem);
144       return;
145       case 2:
146         puts("The disk is full\r");
147       OS_bSignal(&Sema4FileSystem);
148       return;
149       case 3:
150         puts("File already exists\r");
151       OS_bSignal(&Sema4FileSystem);
152       return;
153     }
154   } else{
155     puts(msg);
156     OS_bSignal(&Sema4FileSystem);
157     return;
158   }
159
160
161   appendtofile();
162   OS_bSignal(&Sema4FileSystem);
163 }
164
165 static void parseCATCommand(void){
166   char *buffer;
167   int error;
168
169   const char *msg = "Enter the file name to be printed as the first
       argument\r";
170
171   OS_bWait(&Sema4FileSystem);
172
173   if (buffer = strtok(NULL, " ")) {
174     switch ( eFile_ROpen(buffer) ) {
175       case 1:
176         puts(fileerrormsg);
177         OS_bSignal(&Sema4FileSystem);
178       return;
179       case 3:
180         puts("File does not exist\r");
181         OS_bSignal(&Sema4FileSystem);
182       return;
```

```
183       }
184     } else{
185       puts(msg);
186       OS_bSignal(&Sema4FileSystem);
187       return;
188     }
189
190     do {
191       char nextchar;
192       error = eFile_ReadNext(&nextchar);
193       switch (error) {
194         case 0:
195           putchar(nextchar);
196         break;
197         case 1:
198           puts(fileerrormsg);
199           OS_bSignal(&Sema4FileSystem);
200           return;
201         case 3:
202           printf("\r\n");
203         break;
204       }
205     } while (error == 0);
206
207     if (eFile_RClose()){
208       puts(fileerrormsg);
209     }
210
211
212     OS_bSignal(&Sema4FileSystem);
213 }
214
215 static void parseEditCommand(void){
216     char *buffer;
217     int error;
218
219     const char *msg = "Enter the file name to be edited as the first
        argument\r";
220
221     OS_bWait(&Sema4FileSystem);
222
223     if (buffer = strtok(NULL, " ")) {
224       error = eFile_WOpen(buffer);
225       if (error == 3) {
226         puts("File already exists\r");
227
228         OS_bSignal(&Sema4FileSystem);
229         return;
230       } else if (error == 2) {
231         puts("The disk is full\r");
232
233         OS_bSignal(&Sema4FileSystem);
234         return;
235       } else if (error == 1) {
236         puts(fileerrormsg);
237
```

```
238         OS_bSignal(&Sema4FileSystem);
239         return;
240       }
241     } else{
242       puts(msg);
243 
244       OS_bSignal(&Sema4FileSystem);
245       return;
246     }
247 
248     appendtofile();
249     OS_bSignal(&Sema4FileSystem);
250 }
251 
252 //Description: command for parsing Profiling requests
253 // Can invokde OS_Profile_Start and, OS_Profile_Clear, OS_Profile_Dump
254 //Input: None
255 //Output: none
256 static void parseProfilingCommand(void) {
257   puts("Begin Profiling for 100 samples...\r\n");
258   OS_Profile_Start();
259 }
260 
261 //Description: command for getting charachters from interpreter and
        writing them to file
262 //Ends if charachter '^' is typed
263 //Input: None
264 //Output: none
265 static void appendtofile (void) {
266   char nextchar;
267   puts("Type to append. Enter '^' to finish:\r");
268   while ( (nextchar = getchar()) != '^' ) {
269     putchar (nextchar);
270     switch (eFile_Write(nextchar)){
271       case 1:
272         puts(fileerrormsg);
273         return;
274       case 2: case 3:
275         puts("Interpreter failed\r");
276         return;
277       case 4:
278         puts("\r\n\nDisk is full. Closing the file.");
279         break;
280     }
281     if (nextchar == '\r') {
282       putchar ('\n');
283       switch (eFile_Write('\n')){
284         case 1:
285           puts(fileerrormsg);
286           return;
287         case 2: case 3:
288           puts("Interpreter failed\r");
289           return;
290         case 4:
291           puts("\r\n\nDisk is full. Closing the file.");
292           break;
```

```
293         }
294       }
295     }
296
297     printf("\r\n");
298     if (eFile_WClose()){
299       puts(fileerrormsg);
300     }
301 }
```

Code/interpreter.c

3 Measurement

Below are the data analyzer screen captures.

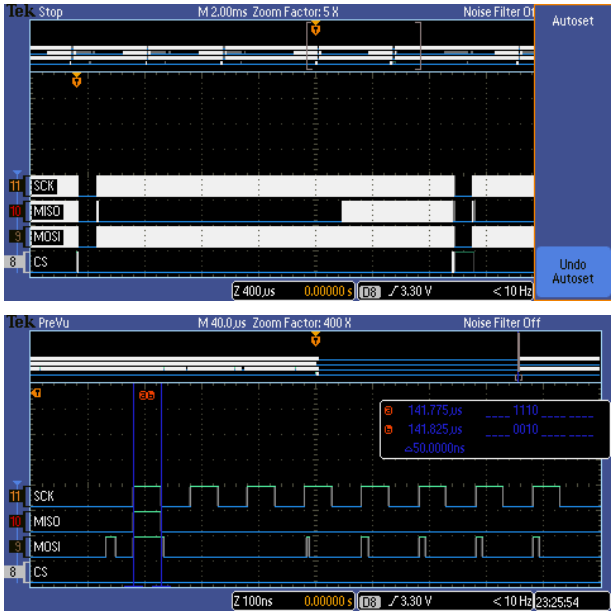


Figure 1: Data Analyzer showing packets

(1) SD card read bandwidth and write bandwidth by average 10000 blocks
Writing time = 3153 us/block
Reading time = 3169 us/block

(2) SPI clock rate The measured SPI clock rate is 100 ns
The maximum baud rate of SD card is 10 $Mbit$

The first picture shows the SPI transmission for the whole communication done for a block write to the SD Card. The second picture is a zoomed in version of showing the first command sent to write. The microcontroller kept sending empty packets after write, until it received an acknowledge from SD Card that write was done.

4 Analysis

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

No, our system uses a linked-list scheme. Therefore it can use all of the data block available and will not have external fragmentation.

2) 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.

In our file system, every data block has the first `short` used to store the address of the next block.

If the average number of bytes in the files are represented as a random variable N, the block it uses is $B = \lceil \frac{N}{512} \rceil$, and the internal fragmentation is

$$2 \times B \times 10 \text{ bytes}$$

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 maximum read/write bandwidth would be 100 Mbps. This is the clock frequency of the SPI. Note that we cannot achieve this bandwidth because some bits are used for sending commands and checksums for the SD card.

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).

The maximum number of files is limited by two factors:
1- Number of available files in the directory: by $64 \times 31 + 62 = 2046$.
2- Number of blocks formatted other than directory $= 2^11 - 32 = 2016$

In our case, since the number of files is limited by the number of formatted blocks, we can increase the number of files by formatting a larger amount of space in SD card. If the number of available files in the directory is the limiting factor, we should allocate more blocks to the directory.

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.

Our system does not allow multiple threads to stream data into one file. If a thread interrupts the `eFile_Write()` function call of another thread there could be errors.

If we wanted to do it, it could be done by adding mutex semaphores into Write function. For an easy solution, we could specify that only one thread should open the file and closes the file. The other thread should wait for the first thread to open the file.

```
int eFile_Write (char data) {
  OS_bWait (&Sema4FileWrite);

  //The actual code

  OS_Signal (&Sema4FileWrite);
}
```

To safely allow multiple calls to WOpen from each thread,

```
int eFile_WOpen (char data) {
  static char numFileOpen = 0;
  OS_bWait (&Sema4FileWOpen);
  numFileOpen ++;
  if (numFileOpen) {
    //actually open the file
```

```
  }   //Otherwise file is already opened

  OS_Signal (&Sema4FileWOpen);
}
int eFile_WClose () {
  static char numFileClose = 0;
  OS_bWait (&Sema4FileWClose);
  numFileOpen --;
  if (numFileOpen == 0) {
    //actually close the file
  }   //Otherwise some other thread is still writing

  OS_Signal (&Sema4FileWClose);
}
```