

```
1 // os345fat.c - file management system
2 // ****
3 // **  DISCLAIMER **  DISCLAIMER **  DISCLAIMER **  DISCLAIMER **  DISCLAIMER  **
4 // **
5 // ** The code given here is the basis for the CS345 projects.
6 // ** It comes "as is" and "unwarranted." As such, when you use part
7 // ** or all of the code, it becomes "yours" and you are responsible to
8 // ** understand any algorithm or method presented. Likewise, any
9 // ** errors or problems become your responsibility to fix.
10 // **
11 // ** NOTES:
12 // ** -Comments beginning with "// ??" may require some implementation.
13 // ** -Tab stops are set at every 3 spaces.
14 // ** -The function API's in "OS345.h" should not be altered.
15 // **
16 // **  DISCLAIMER **  DISCLAIMER **  DISCLAIMER **  DISCLAIMER **  DISCLAIMER  **
17 // ****
18 #include <stdio.h>
19 #include <stdlib.h>
20 #include <string.h>
21 #include <ctype.h>
22 #include <setjmp.h>
23 #include <time.h>
24 #include <assert.h>
25 #include "fat.h"
26 #include "fat_tasks.h"
27 #include "kernel.h"
28
29 // ****
30 // fms variables
31 char dirPath[128]; // directory path
32 extern bool diskMounted; // disk has been mounted
33 extern unsigned char RAMDisk[]; // ram disk
34 extern unsigned char FAT1[]; // current fat table
35 extern unsigned char FAT2[]; // secondary fat table
36 extern FDEntry OFTable[]; // open files
37
38
39 FMSError FMSErrors[NUM_ERRORS] = {
40     // Invalid File Name
41     // Invalid File Type
42     RIPTOR_MSG}, // Invalid File Descriptor
43     {E_INVALID_SECTOR_NUMBER, E_INVALID_SECTOR_NUMB
44     ER_MSG}, // Invalid Sector Number
45     {E_INVALID_FAT_CHAIN, E_INVALID_FAT_CHAIN_MSG},
46     // Invalid FAT Chain
47     {E_INVALID_DIRECTORY, E_INVALID_DIRECTORY_MSG},
48     // Invalid Directory
49     {E_FILE_ALREADY_DEFINED, E_FILE_ALREADY_DEFINED
50     _MSG}, // File Already Defined
51     {E_FILE_NOT_DEFINED, E_FILE_NOT_DEFINED_MSG},
52     // File Not Defined
53     {E_FILE_ALREADY_OPEN, E_FILE_ALREADY_OPEN_MSG},
54     // File Already Open
```

```

50      {E_FILE_NOT_OPEN, E_FILE_NOT_OPEN_MSG}, // Fil
51      e Not Open
52      SG}, // File Directory Full
53      File Space Full
54      {E_END_OF_FILE, E_END_OF_FILE_MSG}, // End-Of-
55      File
56      {E_END_OF_DIRECTORY, E_END_OF_DIRECTORY_MSG},
57      // End-Of-Directory
58      {E_DIRECTORY_NOT_FOUND, E_DIRECTORY_NOT_FOUND_M
59      SG}, // Directory Not Found
60      {E_CAN_NOT_DELETE, E_CAN_NOT_DELETE_MSG}, // C
61      an Not Delete
62      {E_TOO_MANY_FILES_OPEN, E_TOO_MANY_FILES_OPEN_M
63      SG}, // Too Many Files Open
64      {E_NOT_ENOUGH_CONTINUOUS_SPACE, E_NOT_ENOUGH_CON
65      TINOUS_SPACE_MSG}, // Not Enough Contiguous Space
66      {E_DISK_NOT_MOUNTED, E_DISK_NOT_MOUNTED_MSG},
67      // Disk Not Mounted
68      {E_FILE_SEEK_ERROR, E_FILE_SEEK_ERROR_MSG}, //
69      File Seek Error
70      {E_FILE_LOCKED, E_FILE_LOCKED_MSG}, // File Lo
71      cked
72      {E_FILE_DELETE_PROTECTED, E_FILE_DELETE_PROTECT
73      ED_MSG}, // File Delete Protected
74      {E_FILE_WRITE_PROTECTED, E_FILE_WRITE_PROTECTED
75      _MSG}, // File Write Protected
76      {E_READ_ONLY_FILE, E_READ_ONLY_FILE_MSG}, // R
77      ead Only File
78      {E_ILLEGAL_ACCESS, E_ILLEGAL_ACCESS_MSG} // I
79      llegal Access
80      };
81
82  /**
83   * fmsChangeDir - changes the current directory
84   * @fileName: the name of the subdirectory
85   * @return: 0 for success, error number otherwise
86   *
87   * This function changes the current directory to the subdirectory
88   * specified by the argument fileName. You will only need to handle
89   * moving up a directory or moving down a subdirectory. Verify that
90   * fileName is a valid directory name in the current directory.
91   */
92  int fmsChangeDir(char* fileName)
93  {
94      // Variables
95      int error;
96      char mask[4] = " *.*";
97      int index = 0;
98      DirEntry dirEntry;
99      unsigned char dirEntryName[9];
100     unsigned char dirEntryExtension[4];
101     char dirName[9], dirExtension[4];
102     TCB* tcb = getTCB();
103     int curTask = gettid();

```

```

92
93
94     memset(dirEntryName,0,9);
95     memset(dirEntryExtension,0,4);
96     memset(dirExtension,0,4);
97     memset(dirName,0,9);
98     // Validate and parse dirName
99     if (!validateAndParse(FALSE,fileName,dirName,dirExtension))
100     {
101         return E_INVALID_DIRECTORY;
102     }
103
104     // Convert to uppercase
105     convertToUpperCase(dirName);
106     convertToUpperCase(dirExtension);
107
108     // Go through all directory entries
109     while (1)
110     {
111         // Get the next directory entry
112         if ((error = fmsGetNextDirEntry(&index, mask, &dirEntry, CDIR)))
113         {
114             if (error != E_END_OF_DIRECTORY) fmsError(error);
115             break;
116         }
117
118         // Check to see if it is a directory
119         if (dirEntry.attributes & DIRECTORY)
120         {
121             // Need to make some function to get the name and extension from
a dirEntry
122             getEntryStrings(&dirEntry,dirEntryName,dirEntryExtension);
123
124             // Check to see if it matches the directory name
125             if (!(strcmp(dirName,(char*)dirEntryName,8)))
126             {
127                 // Check to see if it matches the directory extension
128
129                 // Replace the cDir with this cluster
130                 tcb[0].cdir = dirEntry.startCluster;
131                 CDIR = dirEntry.startCluster;
132
133                 // See if this is a step down
134                 if (!strcmp(fileName, "."))
135                 {
136                     // No need to change dirPath
137                 }
138                 else if (!strcmp(fileName, ".."))
139                 {
140                     // Need to remove last folder
141                     removeFolderName(dirPath);
142                 }
143                 else
144                 {
145                     // Convert fileName to upper case
146                     convertToUpperCase(fileName);
147
148                     // Change dirPath

```

```

149         strcat(dirPath, fileName);
150
151         // Add a slash
152         strcat(dirPath, "\\");
153     }
154
155     // Return success
156     return 0;
157 }
158 }
159 }
160 return E_DIRECTORY_NOT_FOUND;
161 }
162
163 /**
164  * fmsGetNextDirEntry - get the next directory entry
165  * @dirNum: a pointer to the number of entries already returned
166  * @mask: a mask to select the next entry
167  * @dirEntry: a pointer to the DirEntry to return
168  * @dir: the directory number
169  * @return: 0 for success, error number otherwise
170  *
171  * This function returns the next directory entry of the current directory.
172  * The dirNum parameter is set to 0 for the first entry and is subsequently
173  * updated for each additional call. The next directory entry is returned
174  * in the 32 byte directory structure dirEntry. The parameter mask is a
175  * selection string. If null, return next directory entry. Otherwise, use
176  * the mask string to select the next directory entry.
177  *     A '*' is a wild card for any length string.
178  *     A '?' is a wild card for any single character.
179  *     Any other character must match exactly.
180  * NOTE:
181  *     *.*           all files
182  *     *             all files w/o extension
183  *     a*.txt        all files beginning with the character 'a' and with a
184  *     .txt extension
185  */
186 int fmsGetNextDirEntry(int *dirNum, char* mask, DirEntry* dirEntry, int dir)
187 {
188     // Variables
189     int dirCluster;           /* The cluster number */
190     int dirSector;           /* The sector number */
191     int dirIndex;            /* The index into which entry in the sector */
192     char buffer[BUFSIZE];    /* The sector data */
193     char maskCopy[32];
194
195     strcpy(maskCopy, mask);
196
197     // Get the starting cluster
198     dirCluster = getEndCluster(dir, *dirNum);
199
200     while(1) // find next matching directory entry
201     {
202         // Check to see if this is the end of the directory
203
204         if (!dir && ((dirCluster + BEG_ROOT_SECTOR) == BEG_DATA_SECTOR))
205         {
206             // Ran out of room in the root directory

```

```

206         return E_FILE_SPACE_FULL;
207     }
208
209     if (dir && (dirCluster == FAT_EOC))
210     {
211         // I'm at the end of a directory, not in the root
212         return E_END_OF_DIRECTORY;
213     }
214
215     // Convert the cluster to a sector number
216     dirSector = dirCluster + (dir ? (BEG_DATA_SECTOR - 2) : BEG_ROOT_SECT
OR);
217
218     // Get the index to know which entry in the sector
219     dirIndex = *dirNum % ENTRIES_PER_SECTOR;
220
221     // Get the data from the sector
222     fmsReadSector(buffer, dirSector);
223
224     if (FATDEBUG) printf("I'm looking at sector %d and index %d\n", dirSe
ctor, dirIndex);
225
226     // Read the entry at the indexed byte value;
227     memcpy(dirEntry, &buffer[dirIndex*sizeof(DirEntry)], sizeof(DirEntry)
);
228
229     // Update dirNum
230     ((*dirNum)++);
231
232     if (FATDEBUG) printf("First Byte: %x\n", dirEntry->name[0]);
233
234     // Check to see if this file has been deleted or is empty
235     if ((dirEntry->name[0] == 0x00) || (dirEntry->name[0] == 0xf6))
236     {
237         // Return that every entry after this is invalid
238         return E_END_OF_DIRECTORY;
239     }
240     // Update dirNum
241     (*dirNum)++;
242     if (dirEntry->name[0] != 0xe5)
243     {
244         // Check to see if this is a file or directory
245         //if ((dirEntry->attributes & (DIRECTORY | ARCHIVE)) && !(dirEntr
y->attributes & HIDDEN))
246         if (!(dirEntry->attributes & HIDDEN))
247         {
248             if (FATDEBUG) printf("This is a file/directory\n");
249
250             // Check to see if this is a valid entry
251             if (fmsMask(maskCopy, dirEntry->name, dirEntry->extension))
252             {
253                 break;
254             }
255         }
256     }
257
258     // Check to see if you've reached the end of the sector
259     if ((*dirNum % ENTRIES_PER_SECTOR) == 0)

```

```

260     {
261         // Get the next cluster
262         if (dir) dirCluster = getFatEntry(dirCluster, FAT1);
263         else ++dirCluster;
264     }
265 }
266
267 return 0;
268 }
269
270 // *****
271 // This function gets the DirEntry of a given fileName
272 // Return 0 for success, otherwise, return error number.
273 //
274 int fmsGetDirEntry(char* fileName, DirEntry* dirEntry)
275 {
276     // Variables
277     int error, index = 0;
278     TCB* tcb = getTCB();
279     int curTask = gettid();
280
281     // Get the entry with the fileName as a mask
282     error = fmsGetNextDirEntry(&index, fileName, dirEntry, CDIR);
283
284     // If it is not found, return not found instead of end of directory
285     return (error ? ((error == E_END_OF_DIRECTORY) ? E_FILE_NOT_DEFINED : err
or) : 0);
286 } // end fmsGetDirEntry
287
288 // *****
289 // This function checks a name and extension to see if it matches a mask.
290 // Return 1 for success, otherwise, return 0.
291 //
292 int fmsMask(char* mask, unsigned char* name, unsigned char* extension)
293 {
294     // Variables
295     int maskIndex; /* This is the current character in the mas
k */
296     int nameIndex = 0; /* This is the current character in the nam
e */
297     int extensionIndex = 0; /* This is the current character in the ex
tension */
298     bool checkingName = TRUE; /* This is to see if I'm comparing to name o
r extension */
299
300     // Convert the mask to upper case
301     convertToUpperCase(mask);
302
303     // First iterate through mask
304     for (maskIndex = 0; maskIndex < (int)strlen(mask); maskIndex++)
305     {
306         char currentChar = mask[maskIndex];
307
308         //printf("checkingName: %d & maskIndex: %d currentChar: %c & nameInde
x: %d nameChar: %c ", checkingName, maskIndex, currentChar, nameIndex, name[na
meIndex]);

```

```

309         //printf("& extensionIndex: %d extensionChar: %c\n",extensionIndex,e
xtension[extensionIndex]);
310
311         // If the mask is longer than the face and extension, it does not mat
ch
312         if (checkingName)
313         {
314             if ((name[nameIndex] == 0x20) && ((currentChar != '*') && (curren
tChar != '.'))) return 0;
315         }
316         else
317         {
318             if ((extension[extensionIndex] == 0x20) && (currentChar != '*'))
return 0;
319         }
320
321         // Check to see if the currentChar is an asterisk
322         if (currentChar == '*')
323         {
324             // If this is the last character in the mask, return success
325             if (maskIndex == strlen(mask)-1)
326             {
327                 // If you have already checked the name, return true
328                 if (!checkingName) return 1;
329
330                 // If you are checking the name, only return true if there is
no extension
331                 if (extension[0] == 0x20) return 1;
332                 else return 0;
333             }
334             else
335             {
336                 // Check to see if you have already for a name
337                 if (!checkingName)
338                 {
339                     // Every extension is valid, return success
340                     return 1;
341                 }
342
343                 // Check to see if the next character is a period
344                 if (mask[maskIndex+1] == '.')
345                 {
346                     // If the next character is a '.', skip to extension
347                     checkingName = FALSE;
348
349                     // Increment maskIndex because you don't care about the p
eriod
350                     maskIndex++;
351                 }
352                 else
353                 {
354                     // Else, advance name or extension to the next character
in the mask
355                     if (checkingName)
356                     {
357                         // Check to see if that character is in the name
358                         if (!(nameIndex = advanceString(name, nameIndex, mask
[maskIndex+1])))

```

```

359         {
360             break;
361         }
362     }
363     else
364     {
365         // Check to see if that character is in the extension
366         if (!(extensionIndex = advanceString(extension, extensionIndex, mask[maskIndex+1])))
367         {
368             continue;
369         }
370     }
371
372     // Check return value of advancing to know if it was a success or not
373     }
374 }
375
376 else if (currentChar == '?')
377 {
378     // Skip character
379     if (checkingName) nameIndex++;
380     else extensionIndex++;
381 }
382 else if ((currentChar == '.') && (strcmp(mask, ".") && (strcmp(mask, ".."))))
383 {
384     // Assign checkingName to false
385     checkingName = FALSE;
386
387     // Check to see if you are at the end of the comparing file
388     if ((nameIndex < 7) && (name[nameIndex] != 0x20)) return 0;
389     else continue;
390 }
391 else
392 {
393     // Check the indexed character to make sure it matches
394     if (checkingName)
395     {
396         if (name[nameIndex] != currentChar) return 0;
397         else nameIndex++;
398     }
399     else
400     {
401         if (extension[extensionIndex] != currentChar) return 0;
402         else extensionIndex++;
403     }
404 }
405 }
406
407 // The mask matches at least part of the name
408 if (checkingName)
409 {
410     // If the next character isn't a space
411     if ((nameIndex < 8) && (name[nameIndex] != 0x20)) return 0;
412
413     // The names match, but make sure there is no extension

```



```
414         if (extension[0] != 0x20) return 0;
415     }
416     else
417     {
418         // At least part of the extension matches, but make sure all of it
419         if ((extensionIndex < 3) && (extension[extensionIndex] != 0x20)) return 0;
420     }
421     return 1;
422 }
423
424 // *****
425 // This function is used to advance the string to the passed character
426 // return the new index if successful, return 0 if it doesn't contain
427 // that character
428 //
429 int advanceString(unsigned char* string, int curIndex, char compare)
430 {
431     return 0;
432 }
433
434 // *****
435 // This function converts a passed string to upper case
436 //
437 void convertToUpperCase(char * str)
438 {
439     int ch, i;
440     for(i = 0; i < (int)strlen(str); i++)
441     {
442         ch = toupper(str[i]);
443         str[i] = ch;
444     }
445 }
446
447 // *****
448 // This function helps in removal of folder names
449 //
450 void removeFolderName(char* name)
451 {
452     // Variables
453     int index = strlen(name) - 1;
454
455     // Remove last slash
456     name[index] = 0;
457
458     // Begin traversal
459     for (index -= 1; index > 0; index--)
460     {
461         if (name[index] == '\\')
462         {
463             break;
464         }
465     }
466     else
```

```

468     {
469         name[index] = 0;
470     }
471 }
472 }
473
474 // *****
475 // This function is used to get null-terminated versions of the name and exte
nsion
476 // strings in a DirEntry
477 //
478 void getEntryStrings(DirEntry* dirEntry, unsigned char* dirEntryName, unsigne
d char* dirEntryExtension)
479 {
480     // Variables
481     int i;
482
483     // Begin copy of the name array
484     for (i = 0; i < 8; i++)
485     {
486         // Check to see the next character
487         if (dirEntry->name[i] == 0x20)
488         {
489             // If it is a space, it is fully copied
490             dirEntryName[i] = 0;
491             break;
492         }
493         else
494         {
495             // If it is not a space, copy it
496             dirEntryName[i] = dirEntry->name[i];
497         }
498     }
499
500     // Begin copy of the extension array
501     for (i = 0; i < 3; i++)
502     {
503         // Check to see the next character
504         if (dirEntry->extension[i] == 0x20)
505         {
506             // If it is a space, it is fully copied
507             dirEntryExtension[i] = 0;
508             break;
509         }
510         else
511         {
512             // If it is not a space, copy it
513             dirEntryExtension[i] = dirEntry->extension[i];
514         }
515     }
516 }
517
518 // *****
519 // This function is used to get null-terminated versions of the name and exte
nsion
520 // strings in a DirEntry

```

```

521 //
522 void getFDEntryStrings(FDEntry* dirEntry, unsigned char* dirEntryName, unsigned char* dirEntryExtension)
523 {
524     // Variables
525     int i;
526
527     // Begin copy of the name array
528     for (i = 0; i < 8; i++)
529     {
530         // Check to see the next character
531         if (dirEntry->name[i] == 0x20)
532         {
533             // If it is a space, it is fully copied
534             dirEntryName[i] = 0;
535             break;
536         }
537         else
538         {
539             // If it is not a space, copy it
540             dirEntryName[i] = dirEntry->name[i];
541         }
542     }
543
544     // Write a 0
545     dirEntryName[i] = 0;
546
547     // Begin copy of the extension array
548     for (i = 0; i < 3; i++)
549     {
550         // Check to see the next character
551         if (dirEntry->extension[i] == 0x20)
552         {
553             // If it is a space, it is fully copied
554             dirEntryExtension[i] = 0;
555             break;
556         }
557         else
558         {
559             // If it is not a space, copy it
560             dirEntryExtension[i] = dirEntry->extension[i];
561         }
562     }
563
564     // Write a 0
565     dirEntryExtension[i] = 0;
566 }
567
568 // *****
569 // This function checks to see if a file name is valid
570 //   bool maskable lets you pass '*'s and '?'s
571 //   char* name is the string you are checking
572 //   Return 1 for success, otherwise, return 0.
573 //
574 int validateAndParse(bool maskable, char* fileName, char* name, char* extension)
575 {

```

```

576 // Variables
577 int nameIndex, extensionIndex;
578 int nameInc = 0, extensionInc = 0;
579 char currentChar;
580 bool hasQuotes = FALSE;
581 name[0] = 0;
582 extension[0] = 0;
583
584 // Check to see if it is '.' or '..'
585 if (!(strcmp(fileName, ".") || !(strcmp(fileName, ".."))))
586 {
587     // Assign the name
588     for (nameIndex = 0; nameIndex < (int)strlen(fileName); nameIndex++)
589     {
590         name[nameIndex] = '.';
591     }
592
593     // Assign the null termination
594     name[nameIndex] = 0;
595     extension[0] = 0;
596
597     // Return success
598     return 1;
599 }
600
601 // Parse the name
602 for (nameIndex = 0; nameIndex < (8+nameInc); nameIndex++)
603 {
604     // Get the current char
605     currentChar = fileName[nameIndex];
606
607     // Check for bad characters
608     if ((currentChar == ';') || (currentChar == ':') || (currentChar == '
609 \\'') || (currentChar == '\\')) return 0;
610     if ((currentChar == '*') || (currentChar == '\\') || (currentChar ==
611 '<') || (currentChar == '>')) return 0;
612     if ((currentChar == ' ') || (currentChar == '|') || (currentChar == '
613 ?') || (currentChar == '+')) return 0;
614     if ((currentChar == '=') || (currentChar == '[') || (currentChar == '
615 ]') || (currentChar == '/')) return 0;
616     if ((currentChar == ',') return 0;
617
618     // Check if this is a period
619     if (currentChar == '.')
620     {
621         // If this is the first character, return
622         if (nameIndex == 0) return 0;
623
624         // Terminate the string
625         name[nameIndex-nameInc] = 0;
626
627         // Go do the extension
628         break;
629     }
630     else if (currentChar == '\\')
631     {
632         // Ignore the quotes
633         nameInc++;

```

```

630         hasQuotes = (hasQuotes ? FALSE : TRUE);
631     }
632     else
633     {
634         // Copy the character
635         name[nameIndex+nameInc] = currentChar;
636
637         // If this is the last character, return
638         if (currentChar == 0) return 1;
639     }
640
641     // This is the last character
642     if (nameIndex == (8+nameInc-1))
643     {
644         // If the next character is not a period or null, return false
645         if ((fileName[nameIndex+1] != '.') && (fileName[nameIndex+1] != 0
646
647         {
648             // This name is too long
649             return 0;
650         }
651         else if (fileName[nameIndex+nameInc+1] == 0)
652         {
653             // Terminate and break
654             name[nameIndex+nameInc+1] = 0;
655             break;
656         }
657         else
658         {
659             // Terminate and increase nameIndex
660             name[nameIndex+nameInc+1] = 0;
661         }
662     }
663
664     // Parse the extension
665     for (extensionIndex = 0; extensionIndex < (3+extensionInc); extensionIndex++)
666     {
667         // Get the current char
668         currentChar = fileName[nameIndex + extensionIndex + 1];
669
670         // Check for bad characters
671         if ((currentChar == ';') || (currentChar == ':') || (currentChar == '
672         \'"') || (currentChar == '\\')) return 0;
673         if ((currentChar == '*') || (currentChar == '\\') || (currentChar ==
674         '<') || (currentChar == '>')) return 0;
675         if ((currentChar == ' ') || (currentChar == '|') || (currentChar == '
676         ?') || (currentChar == '+')) return 0;
677         if ((currentChar == '=') || (currentChar == '[') || (currentChar == '
678         ]') || (currentChar == '/')) return 0;
679         if ((currentChar == '.') || (currentChar == ',')) return 0;
680
681         if (currentChar == '\\')
682         {
683             // Ignore the quotes
684             extensionInc++;
685             hasQuotes = (hasQuotes ? FALSE : TRUE);

```

```
682     }
683     else
684     {
685         // Copy the character
686         extension[extensionIndex-extensionInc] = currentChar;
687
688         // If this is the last character, return
689         if (currentChar == 0) return 1;
690     }
691
692     // If the index is 2 and the next character is not a null, return false;
693     if (extensionIndex == (3+extensionInc-1))
694     {
695         // Check to see if this name is too long
696         if ((strlen(fileName) - nameIndex - 1) > 3)
697         {
698             // This name is too long
699             return 0;
700         }
701         else
702         {
703             // Add the null character
704             extension[extensionIndex-extensionInc+1] = 0;
705         }
706     }
707 }
708
709 // If the quotes haven't ended, error
710 if (hasQuotes) return 0;
711
712 // Return success
713 return 1;
714 }
715
716
717 // *****
718 // This function returns the starting cluster
719 //
720 int getEndCluster(int dir, int dirNum)
721 {
722     // Variables
723     int dirCluster = 0;
724     int i = 0;
725
726     // Check where you are
727     if (dir)
728     {
729         // If you are not looking at the root directory, traverse
730         dirCluster = dir;
731         for (i = 0; i < (dirNum / ENTRIES_PER_SECTOR); i++)
732         {
733             dirCluster = getFatEntry(dirCluster, FAT1);
734         }
735     }
736     else
737     {
```

```

738         // Root directory
739         dirCluster = (dirNum / ENTRIES_PER_SECTOR);
740     }
741
742     return dirCluster;
743 }
744
745 /**
746  * fmsCloseFile - close an open file
747  * @fileDescriptor: the id of the open file to close
748  * @return: 0 for success, error number otherwise
749  *
750  * This function closes the open file specified by fileDescriptor.
751  * The fileDescriptor was returned by fmsOpenFile and is an index
752  * into the open file table.
753  */
754 int fmsCloseFile(int fileDescriptor)
755 {
756     if (!diskMounted)
757         return E_DISK_NOT_MOUNTED;
758     if (fileDescriptor < 0)
759         return E_INVALID_FILE_DESCRIPTOR;
760
761     FDEntry* fdEntry = &OFTable[fileDescriptor];
762
763     if (fdEntry->name[0] == 0)
764         return E_FILE_NOT_OPEN;
765
766     FDEntry* fdEntry = &OFTable[fileDescriptor];
767     char fileName[12];
768     for (i = 0; i < 12; i++) {
769         fileName[i] = fdEntry->name[i];
770     }
771
772     DirEntry* dirEntry;
773     int error;
774     if (error = fmsGetDirEntry(fileName, dirEntry))
775         return error;
776
777     if (fdEntry->flags == FILE_ALTERED) {
778         setDirTimeDate(dirEntry);
779         dirEntry->fileSize = fdEntry->fileSize;
780     }
781
782     // Flush buffer
783     fmsWriteSector(fdEntry->buffer, C_2_S(fdEntry->currentCluster));
784
785     return 0; // return success
786 }
787
788 /**
789  * fmsDefineFile - creates a new file in the current directory
790  * @fileName: the name of the file to create
791  * @attribute: the type of file to create
792  *
793  * If attribute=DIRECTORY, this function creates a new directory fileName
794  * in the current directory. The directory entries "." and ".." are also
795  * defined. It is an error to try and create a directory that already exists.

```

```

796 *
797 * Else, this function creates a new file fileName in the current directory.
798 * It is an error to try and create a file that already exists.
799 * The start cluster field should be initialized to cluster 0. In FAT-12,
800 * files of size 0 should point to cluster 0 (otherwise chkdsk should report
801 * an error). Remember to change the start cluster field from 0 to a free
802 * cluster when writing to the file.
803 */
804 int fmsDefineFile(char* fileName, int attribute)
805 {
806     return 0;
807 }
808
809 /**
810 *fmsDeleteFile - deletes fileName from the current directory
811 * @fileName: the name of the file to delete
812 * @return: 0 for success, error number otherwise
813 *
814 * This function deletes the file fileName from the current directory. The
815 * file name should be marked with an "E5" as the first character and the
816 * chained clusters in FAT 1 reallocated (cleared to 0).
817 */
818 int fmsDeleteFile(char* fileName)
819 {
820     return 0;
821 }
822
823 /**
824 * fmsOpenFile - opens a file with specified access mode
825 * @fileName: the name of the file to open
826 * @rwMode: the mode of the open file
827 * @return: If successful, return file descriptor (index into open file table
828 * ),
829 * error number otherwise
830 *
831 * This function opens the file fileName for access as specified by rwMode.
832 * It is an error to try to open a file that does not exist.
833 * The open mode rwMode is defined as follows:
834 * 0 - Read access only.
835 *     The file pointer is initialized to the beginning of the file.
836 *     Writing to this file is not allowed.
837 * 1 - Write access only.
838 *     The file pointer is initialized to the beginning of the file.
839 *     Reading from this file is not allowed.
840 * 2 - Append access.
841 *     The file pointer is moved to the end of the file.
842 *     Reading from this file is not allowed.
843 * 3 - Read/Write access.
844 *     The file pointer is initialized to the beginning of the file.
845 *     Both read and writing to the file is allowed.
846 * A maximum of 32 files may be open at any one time.
847 */
848 int fmsOpenFile(char* fileName, int rwMode)
849 {
850     // Check permission
851     // Spew errors ("Invalid File Name", "File Not Defined",
852     // "File Already open", "Too Many Files Open", "File Space Full"

```



```

853
854     int error, i, j, fd = 0;
855     DirEntry dirEntry;
856     FDEntry* fdEntry;
857     TCB* tcb = getTCB(); // For use by CDIR ma
cro
858     int curTask = gettid(); // For use by CDIR
macro
859
860     if (error = fmsGetDirEntry(fileName, &dirEntry)) // Returned dir entr
y stored in dirEntry
861         return error;
862     if (dirEntry->fileName[0] == 0)
863         return E_INVALID_FILE_NAME;
864     if (dirEntry.attributes == READ_ONLY && rwMode)
865         return E_READ_ONLY_FILE;
866
867     for (i = 0, fd = -1; i < NFILES; i++) { // Look through en
tire open file table
868         fdEntry = &OFTable[i];
869
870         if (fdEntry->name[0] == 0) // Open slot (not t
oo many files
871             fd = i; // already open),
so save that location
872
873 // But we still need
to iterate over
874 // 12 because name and extension are contig. // rest of entries in
OFTable to
875     for (j = 0; j < 12; j++) { // compare the name
of file in the file
876         if (fdEntry->name[j] != dirEntry.name[j]) // descriptor we're
checking right now
877             break; //inner for, check next entry // to name of file
in directory entry,
878             if (j == 12) // to make sure that
the
879                 return E_FILE_ALREADY_OPEN; // file isn't alre
ady open
880     }
881 }
882 if (fd == -1) // fd never got assi
gned to an open slot
883     return E_TOO_MANY_FILES_OPEN; // so, too many file
s open
884
885     memcpy(fdEntry->name, dirEntry.name, 8); // Create new file d
escriptor entry
886     memcpy(fdEntry->extension, dirEntry.extension, 3); // TODO check about
null-termination
887     fdEntry->attributes = dirEntry.attributes;
888     fdEntry->directoryCluster = CDIR;
889     fdEntry->startCluster = dirEntry.startCluster;
890     fdEntry->currentCluster = !rwMode ? fdEntry->startCluster : 0; // At l
east for part 1
891     fdEntry->fileSize = (rwMode == 1) ? 0 : dirEntry.fileSize; // If writing
to file, 0 size

```

```

892     fdEntry->pid = curTask;                                     // curTask
893     fdEntry->mode = rwMode;
894     fdEntry->flags = 0;
895     fdEntry->fileIndex = (rwMode != 2) ? 0 : dirEntry.fileSize; // If appendi
ng file, go to end
896     memset(fdEntry->buffer, -2, BUFSIZE);
897
898     if (rwMode == 2) {                                           // Appending, fill
buff with last cluster
899         fdEntry->currentCluster = fdEntry->startCluster;
900         unsigned short nextCluster = 0;
901         while ((nextCluster = getFatEntry(fdEntry->currentCluster, FAT1)) !=
FAT_EOC)
902             fdEntry->currentCluster = nextCluster;
903         if ((error = fmsReadSector(fdEntry->buffer, C_2_S(fdEntry->currentClu
ster))))
904             return error;
905     }
906
907     return fd;
908 }
909
910 /**
911  * fmsReadFile - read a specified number of bytes from a file
912  * @fileDescriptor: the file descriptor of the open file
913  * @buffer: the buffer to load the read data into
914  * @nBytes: the number of bytes to read
915  * @return: the number of bytes read, error number otherwise
916  *
917  * This function reads nBytes bytes from the open file specified by
918  * fileDescriptor into memory pointed to by buffer. The fileDescriptor was
919  * returned by fmsOpenFile and is an index into the open file table. After
920  * each read, the file pointer is advanced.
921  */
922 int fmsReadFile(int fileDescriptor, char* buffer, int nBytes)
923 {
924     if (!diskMounted)
925         return E_DISK_NOT_MOUNTED;
926     if (fileDescriptor < 0)
927         return E_INVALID_FILE_DESCRIPTOR;
928
929     FdEntry* fdEntry = &OFTable[fileDescriptor];
930
931     if (fdEntry->name[0] == 0)
932         return E_FILE_NOT_OPEN;
933
934     unsigned short nextCluster = 0;
935     int error;
936
937     // If the buffer for this fd is empty (either we're entering
938     // a new sector and just cleared the buffer in a previous
939     // iteration of this loop, or it's never been filled), fill it
940     if (fdEntry->buffer[0] == -2) {
941         if (error = fmsReadSector(fdEntry->buffer, C_2_S(fdEntry->currentClus
ter)));
942             return error;
943         // So we're at the next cluster next time for next time
944         nextCluster = getFatEntry(fdEntry->currentCluster, FAT1);

```

```

945     fdEntry->currentCluster = nextCluster;
946 }
947
948 memcpy(buffer, fdEntry->buffer + (fdEntry->fileIndex % BYTES_PER_SECTOR),
nBytes);
949 fdEntry->fileIndex = fdEntry->fileIndex + nBytes;
950
951 if (fdEntry->fileIndex % BYTES_PER_SECTOR == 0) {
952     memset(fdEntry->buffer, -2, BUFSIZE);
953 }
954
955 if (fdEntry->fileIndex >= fdEntry->fileSize)
956     return E_END_OF_FILE;
957
958 return nBytes;
959 }
960
961 /**
962  * fmsSeekFile - change the current file pointer of an open file
963  * @fileDescriptor: the file descriptor of the open file
964  * @index: the new file position
965  * @return: the new position in the file, error number otherwise
966  *
967  * This function changes the current file pointer of the open file specified
968  * by fileDescriptor to the new file position specified by index. The
969  * fileDescriptor was returned by fmsOpenFile and is an index into the open
970  * file table. The file position may not be positioned beyond the end of the
971  * file.
972  */
973 int fmsSeekFile(int fileDescriptor, int index)
974 {
975     if (!diskMounted)
976         return E_DISK_NOT_MOUNTED;
977     if (fileDescriptor < 0)
978         return E_INVALID_FILE_DESCRIPTOR;
979
980     FDEntry* fdEntry = &OFTable[fileDescriptor];
981
982     // This fdEntry is still free/hasn't been allocated
983     if (fdEntry->name[0] == 0)
984         return E_FILE_NOT_OPEN; // Could also be E_FILE_NOT_DEFINE
985
986     if (index >= fdEntry->fileSize)
987         return E_FILE_SEEK_ERROR;
988
989     fdEntry->fileIndex = index;
990
991     // Get to the right cluster, put part of cluster up to index into fd's bu
992     ffer
993     char buf[BUFSIZE];
994     int i;
995     fdEntry->currentCluster = fdEntry->startCluster;
996     for (i = 0; i < (index / BYTES_PER_SECTOR); i++) {
997         unsigned int nextCluster = getFatEntry(fdEntry->currentCluster, FAT1)
998     ;
999     fdEntry->currentCluster = nextCluster;
1000 }
1001 if ((error = fmsReadSector(buf, C_2_S(fdEntry->currentCluster))))

```

```

999         return error;
1000
1001     memcpy(fdEntry->buffer, buff, (index % BYTES_PER_SECTOR));
1002
1003     return index;
1004 }
1005
1006 /**
1007  * fmsWriteFile - write to an open file
1008  * @fileDescriptor: the file descriptor of the open file
1009  * @buffer: the data to write to a file
1010  * @nBytes: the number of bytes to write
1011  * @return: the number of bytes written, error number otherwise
1012  *
1013  * This function writes nBytes bytes to the open file specified by
1014  * fileDescriptor from memory pointed to by buffer. The fileDescriptor was
1015  * returned by fmsOpenFile and is an index into the open file table.
1016  * Writing is always "overwriting" not "inserting" in the file and always
1017  * writes forward from the current file pointer position.
1018  */
1019 int fmsWriteFile(int fileDescriptor, char* buffer, int nBytes)
1020 {
1021     return 0;
1022 }
1023
1024
1025 // *****
1026 // *****
1027 // *****
1028 // *****
1029 // *****
1030
1031
1032 // *****
1033 // Take a FAT table index and return an unsigned short containing the 12-bit
1034 // FAT entry code
1035 // *****
1036 // Take a FAT table index and return an unsigned short containing the 12-bit
1037 // FAT entry code
1038 unsigned short getFatEntry(int FATindex, unsigned char* FATtable)
1039 {
1040     unsigned short FATEntryCode;           /* The return value */
1041     int FatOffset = ((FATindex * 3) / 2);  /* Calculate the offset of the u
1042 nsigned short to get */
1043     if ((FATindex % 2) == 1)               /* If the index is odd */
1044     {
1045         // Pull out a unsigned short from a unsigned char array
1046         FATEntryCode = *((unsigned short *)&FATtable[FatOffset]);
1047         FATEntryCode = SWAP_BYTES(FATEntryCode);
1048         FATEntryCode >=> 4;                /* Extract the high-order 1
1049 2 bits */

```

```

1046     }
1047     else                                     /* If the index is ev
en */
1048     {
1049         // Pull out a unsigned short from a unsigned char array
1050         FATEntryCode = *((unsigned short *)&FATtable[FatOffset]);
1051         FATEntryCode = SWAP_BYTES(FATEntryCode);
1052         FATEntryCode &= 0x0fff;               /* Extract the low-order 12 bits
*/
1053     }
1054     return FATEntryCode;
1055 } // end GetFatEntry
1056
1057
1058
1059 // *****
1060 // *****
1061 // Replace the 12-bit FAT entry code in the unsigned char FAT table at index
1062 void setFatEntry(int FATindex, unsigned short FAT12ClusEntryVal)
1063 {
1064     int FATOffset = ((FATindex * 3) / 2);    /* Calculate the offset */
1065     int FATData = *((unsigned short *)&FAT1[FATOffset]);
1066     FATData = SWAP_BYTES(FATData);
1067     if (FATindex % 2 == 0)                   /* If the index is even */
1068     {                                         /* mask to 12 bits */
1069         FAT12ClusEntryVal &= 0x0FFF;        /* mask complement */
1070     }
1071     else                                     /* Index is odd */
1072     {                                         /* move 12-bits high */
1073         FATData &= 0x000F;                 /* mask complement */
1074     }
1075     // Update FAT entry value in the FAT table
1076     FATData = SWAP_BYTES(FATData);
1077     *((unsigned short *)&FAT1[FATOffset]) = FATData | FAT12ClusEntryVal;
1078 } // End SetFatEntry
1079
1080
1081 // *****
1082 //      setDirTimeDate
1083 //
1084 //      struct tm
1085 //      {
1086 //          int tm_sec;           // 0 to 60
1087 //          int tm_min;           // 0 to 59
1088 //          int tm_hour;          // 0 to 23
1089 //          int tm_mday;          // 1 to 31
1090 //          int tm_mon;           // 0 to 11
1091 //          int tm_year;          // year - 1900
1092 //          int tm_wday;          // Sunday = 0
1093 //          int tm_yday;          // 0 to 365
1094 //          int tm_isdst;         // >0 if Daylight Savings Time,
1095 //                                // 0 if Standard,
1096 //                                // <0 if unknown
1097 //          char *tm_zone;        // time zone name
1098 //          long tm_gmtoff;       // offset from GMT

```

```

1099 //      };
1100 //
1101 void setDirTimeDate(DirEntry* dir)
1102 {
1103     time_t a;
1104     struct tm *b;
1105
1106     time(&a);
1107     b = localtime(&a);
1108     dir->date.year = b->tm_year + 1900 - 1980;
1109     dir->date.month = b->tm_mon;
1110     dir->date.day = b->tm_mday;
1111
1112     dir->time.hour = b->tm_hour;
1113     dir->time.min = b->tm_min;
1114     dir->time.sec = b->tm_sec / 2; // FAT16 time resolution is 2 seconds (only
    5 bits allocated to seconds);
1115     return;
1116 } // end setDirTimeDate
1117
1118
1119
1120 // *****
    *****
1121 // Error processor
1122 void fmsError(int error)
1123 {
1124     int i;
1125
1126     for (i=0; i<NUM_ERRORS; i++)
1127     {
1128         if (FMSErrors[i].error == error)
1129         {
1130             printf("%s\n", FMSErrors[i].error_msg);
1131             return;
1132         }
1133     }
1134     printf("%s %d\n", E_UNDEFINED_MSG, error);
1135     return;
1136 } // end fmsError
1137
1138
1139
1140 // *****
    *****
1141 int fmsMount(char* fileName, void* ramDisk)
1142 //      Called by mount command.
1143 //      This function loads a RAM disk image from a file.
1144 //      The parameter fileName is the file path name of the disk image.
1145 //      The parameter ramDisk is a pointer to a character array whose
1146 //      size is equal to a 1.4 mb floppy disk (2849 ` 512 bytes).
1147 //      Return 0 for success, otherwise, return the error number
1148 {
1149     FILE* fp;
1150     fp = fopen(fileName, "rb");
1151     if (fp)
1152     {
1153         fread(ramDisk, sizeof(char), SECTORS_PER_DISK * BYTES_PER_SECTOR, fp);

```

```

1154     }
1155     else return -1;
1156     fclose(fp);
1157     // copy FAT table to memory
1158     memcpy(FAT1, &RAMDisk[1 * BYTES_PER_SECTOR], NUM_FAT_SECTORS * BYTES_PER_
SECTOR);
1159     memcpy(FAT2, &RAMDisk[10 * BYTES_PER_SECTOR], NUM_FAT_SECTORS * BYTES_PER
_SECTOR);
1160     diskMounted = 1;                                /* disk has been mounted */
1161     //@DISABLE_SWAPS
1162     strcpy(dirPath, fileName);
1163     strcat(dirPath, "\\");
1164     return 0;
1165 } // end fmsMount
1166 //@ENABLE_SWAPS
1167
1168
1169
1170 // *****
1171 int fmsUnMount(char* fileName, void* ramDisk)
1172 // Called by the unmount command.
1173 // This function unloads your Project 5 RAM disk image to file computer file.
1174 // The parameter fileName is the file path name of the disk image.
1175 // The pointer parameter ramDisk points to a character array whose size is eq
ual to a 1.4
1176 // mb floppy disk (2849 ` 512 bytes).
1177 // Return 0 for success; otherwise, return the error number.
1178 {
1179     diskMounted = 0;                                /* unmount disk */
1180     return -1;
1181 } // end
1182
1183
1184
1185 // *****
1186 int fmsReadSector(void* buffer, int sectorNumber)
1187 // Read into buffer RAM disk sector number sectorNumber.
1188 // Sectors are 512 bytes.
1189 // Return 0 for success; otherwise, return an error number.
1190 {
1191     memcpy(buffer, &RAMDisk[sectorNumber * BYTES_PER_SECTOR], BYTES_PER_SECTO
R);
1192     return 0;
1193 } // end fmsReadSector
1194
1195 // *****
1196 int fmsWriteSector(void* buffer, int sectorNumber)
1197 // Write 512 bytes from memory pointed to by buffer to RAM disk sector sector
Number.
1198 // Return 0 for success; otherwise, return an error number.
1199 {
1200     memcpy(&RAMDisk[sectorNumber * BYTES_PER_SECTOR], buffer, BYTES_PER_SECTO
R);
1201     return 0;
1202 } // end fmsWriteSector

```