```c
// os345fat.c - file management system
// ***********************************************************************
// **   DISCLAMER ** DISCLAMER ** DISCLAMER ** DISCLAMER ** DISCLAMER   **
// **                                                                   **
// ** The code given here is the basis for the CS345 projects.          **
// ** It comes "as is" and "unwarranted."  As such, when you use part   **
// ** or all of the code, it becomes "yours" and you are responsible to **
// ** understand any algorithm or method presented.  Likewise, any      **
// ** errors or problems become your responsibility to fix.             **
// **                                                                   **
// ** NOTES:                                                            **
// ** -Comments beginning with "// ??" may require some implementation. **
// ** -Tab stops are set at every 3 spaces.                             **
// ** -The function API's in "OS345.h" should not be altered.           **
// **                                                                   **
// **   DISCLAMER ** DISCLAMER ** DISCLAMER ** DISCLAMER ** DISCLAMER   **
// ***********************************************************************
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <setjmp.h>
#include <time.h>
#include <assert.h>
#include "fat.h"
#include "fat_tasks.h"
#include "kernel.h"

// ***********************************************************************
// fms variables
char dirPath[128];                                  // directory path
extern bool diskMounted;                          // disk has been mounted
extern unsigned char RAMDisk[];                  // ram disk
extern unsigned char FAT1[];                      // current fat table
extern unsigned char FAT2[];                      // secondary fat table
extern FDEntry OFTable[];                        // open files


FMSERROR FMSErrors[NUM_ERRORS]   = {
                              {E_INVALID_FILE_NAME, E_INVALID_FILE_NAME_MSG},  // Invalid File Name
                              {E_INVALID_FILE_TYPE, E_INVALID_FILE_TYPE_MSG},  // Invalid File Type
                              {E_INVALID_FILE_DESCRIPTOR, E_INVALID_FILE_DESCRIPTOR_MSG},  // Invalid File Descriptor
                              {E_INVALID_SECTOR_NUMBER, E_INVALID_SECTOR_NUMBER_MSG},  // Invalid Sector Number
                              {E_INVALID_FAT_CHAIN, E_INVALID_FAT_CHAIN_MSG},  // Invalid FAT Chain
                              {E_INVALID_DIRECTORY, E_INVALID_DIRECTORY_MSG},  // Invalid Directory

                              {E_FILE_ALREADY_DEFINED, E_FILE_ALREADY_DEFINED_MSG},  // File Already Defined
                              {E_FILE_NOT_DEFINED, E_FILE_NOT_DEFINED_MSG}, // File Not Defined
                              {E_FILE_ALREADY_OPEN, E_FILE_ALREADY_OPEN_MSG},  // File Already Open
                              {E_FILE_NOT_OPEN, E_FILE_NOT_OPEN_MSG},  // File Not Open
                              {E_FILE_DIRECTORY_FULL, E_FILE_DIRECTORY_FULL_MSG},  // File Directory Full
                              {E_FILE_SPACE_FULL, E_FILE_SPACE_FULL_MSG},  // File Space Full
                              {E_END_OF_FILE, E_END_OF_FILE_MSG},  // End-Of-File
                              {E_END_OF_DIRECTORY, E_END_OF_DIRECTORY_MSG},  // End-Of-Directory
                              {E_DIRECTORY_NOT_FOUND, E_DIRECTORY_NOT_FOUND_MSG},  // Directory Not Found
                              {E_CAN_NOT_DELETE, E_CAN_NOT_DELETE_MSG},  // Can Not Delete

                              {E_TOO_MANY_FILES_OPEN, E_TOO_MANY_FILES_OPEN_MSG},  // Too Many Files Open
                              {E_NOT_ENOUGH_CONTINOUS_SPACE, E_NOT_ENOUGH_CONTINOUS_SPACE_MSG},  // Not Enough Contiguous Space
                              {E_DISK_NOT_MOUNTED, E_DISK_NOT_MOUNTED_MSG},  // Disk Not Mounted

                              {E_FILE_SEEK_ERROR, E_FILE_SEEK_ERROR_MSG},  // File Seek Error
                              {E_FILE_LOCKED, E_FILE_LOCKED_MSG},  // File Locked
                              {E_FILE_DELETE_PROTECTED, E_FILE_DELETE_PROTECTED_MSG},  // File Delete Protected
                              {E_FILE_WRITE_PROTECTED, E_FILE_WRITE_PROTECTED_MSG},  // File Write Protected
                              {E_READ_ONLY_FILE, E_READ_ONLY_FILE_MSG},  // Read Only File
                              {E_ILLEGAL_ACCESS, E_ILLEGAL_ACCESS_MSG}   // Illegal Access
                   };

/**
 * fmsChangeDir - changes the current directory
 * @fileName: the name of the subdirectory
 * @return: 0 for success, error number otherwise
 *
 * This function changes the current directory to the subdirectory
 * specified by the argument fileName. You will only need to handle
 * moving up a directory or moving down a subdirectory. Verify that
 * fileName is a valid directory name in the current directory.
 */
int fmsChangeDir(char* fileName)
{
    // Variables
    int error;
    char mask[4] = "*.*";
    int index = 0;
    DirEntry dirEntry;
    unsigned char dirEntryName[9];
    unsigned char dirEntryExtension[4];
    char dirName[9], dirExtension[4];
    TCB* tcb = getTCB();
    int curTask = gettid();


    memset(dirEntryName,0,9);
    memset(dirEntryExtension,0,4);
    memset(dirExtension,0,4);
    memset(dirName,0,9);
    // Validate and parse dirName
    if (!validateAndParse(FALSE,fileName,dirName,dirExtension))
    {
        return E_INVALID_DIRECTORY;
    }

    // Convert to uppercase
    convertToUpperCase(dirName);
    convertToUpperCase(dirExtension);

    // Go through all directory entries
    while (1)
    {
        // Get the next directory entry
        if ((error = fmsGetNextDirEntry(&index, mask, &dirEntry, CDIR)))
        {
            if (error != E_END_OF_DIRECTORY) fmsError(error);
            break;
        }

        // Check to see if it is a directory
        if (dirEntry.attributes & DIRECTORY)
        {
            // Need to make some function to get the name and extension from a dirEntry
            getEntryStrings(&dirEntry,dirEntryName,dirEntryExtension);

            // Check to see if it matches the directory name
            if (!(strncmp(dirName,(char*)dirEntryName,8)))
            {
                // Check to see if it matches the directory extension

                // Replace the cDir with this cluster
                tcb[0].cdir = dirEntry.startCluster;
                CDIR = dirEntry.startCluster;

                // See if this is a step down
                if (!strcmp(fileName,"."))
                {
                    // No need to change dirPath
                }
                else if (!strcmp(fileName,".."))
                {
                    // Need to remove last folder
                    removeFolderName(dirPath);
                }
                else
                {
                    // Convert fileName to upper case
                    convertToUpperCase(fileName);

                    // Change dirPath
```

```c
                     strcat(dirPath, fileName);

                     // Add a slash
                     strcat(dirPath, "\\");
                 }

                 // Return success
                 return 0;
             }
         }
     }
     return E_DIRECTORY_NOT_FOUND;
}

/**
 * fmsGetNextDirEntry - get the next directory entry
 * @dirNum: a pointer to the number of entries already returned
 * @mask: a mask to select the next entry
 * @dirEntry: a pointer to the DirEntry to return
 * @dir: the directory number
 * @return: 0 for success, error number otherwise
 *
 * This function returns the next directory entry of the current directory.
 * The dirNum parameter is set to 0 for the first entry and is subsequently
 * updated for each additional call. The next directory entry is returned
 * in the 32 byte directory structure dirEntry. The parameter mask is a
 * selection string.  If null, return next directory entry. Otherwise, use
 * the mask string to select the next directory entry.
 *    A '*' is a wild card for any length string.
 *    A '?' is a wild card for any single character.
 *    Any other character must match exactly.
 * NOTE:
 *        *.*            all files
 *        *             all files w/o extension
 *        a*.txt         all files beginning with the character 'a' and with a
 .txt extension
 */
int fmsGetNextDirEntry(int *dirNum, char* mask, DirEntry* dirEntry, int dir)
{
    // Variables
    int dirCluster;             /* The cluster number */
    int dirSector;             /* The sector number */
    int dirIndex;             /* The index into which entry in the sector */
    char buffer[BUFSIZE];     /* The sector data */
    char maskCopy[32];

    strcpy(maskCopy,mask);

    // Get the starting cluster
    dirCluster = getEndCluster(dir,*dirNum);

    while(1) // find next matching directory entry
    {
        // Check to see if this is the end of the directory

        if (!dir && ((dirCluster + BEG_ROOT_SECTOR) == BEG_DATA_SECTOR))
        {
            // Ran out of room in the root directory
            return E_FILE_SPACE_FULL;
        }

        if (dir && (dirCluster == FAT_EOC))
        {
            // I'm at the end of a directory, not in the root
            return E_END_OF_DIRECTORY;
        }

        // Convert the cluster to a sector number
        dirSector = dirCluster + (dir ? (BEG_DATA_SECTOR - 2) : BEG_ROOT_SECTOR);

        // Get the index to know which entry in the sector
        dirIndex = *dirNum % ENTRIES_PER_SECTOR;

        // Get the data from the sector
        fmsReadSector(buffer,dirSector);

        if (FATDEBUG) printf("I'm looking at sector %d and index %d\n", dirSector, dirIndex);

        // Read the entry at the indexed byte value;
        memcpy(dirEntry, &buffer[dirIndex*sizeof(DirEntry)], sizeof(DirEntry));

        // Update dirNum
        //(*dirNum)++;

        if (FATDEBUG) printf("First Byte: %x\n",dirEntry->name[0]);

        // Check to see if this file has been deleted or is empty
        if ((dirEntry->name[0] == 0x00) || (dirEntry->name[0] == 0xf6))
        {
            // Return that every entry after this is invalid
            return E_END_OF_DIRECTORY;
        }
        // Update dirNum
        (*dirNum)++;
        if (dirEntry->name[0] != 0xe5)
        {
            // Check to see if this is a file or directory
            //if ((dirEntry->attributes & (DIRECTORY | ARCHIVE)) && !(dirEntry->attributes & HIDDEN))
            if (!(dirEntry->attributes & HIDDEN))
            {
                if (FATDEBUG) printf("This is a file/directory\n");

                // Check to see if this is a valid entry
                if (fmsMask(maskCopy, dirEntry->name, dirEntry->extension))
                {
                    break;
                }
            }
        }

        // Check to see if you've reached the end of the sector
        if ((*dirNum % ENTRIES_PER_SECTOR) == 0)
        {
            // Get the next cluster
            if (dir) dirCluster = getFatEntry(dirCluster, FAT1);
            else ++dirCluster;
        }
    }

    return 0;
}

// *****************************************************************************
//  This function gets the DirEntry of a given fileName
//    Return 0 for success, otherwise, return error number.
//
int fmsGetDirEntry(char* fileName, DirEntry* dirEntry)
{
    // Variables
    int error, index = 0;
    TCB* tcb = getTCB();
    int curTask = gettid();

    // Get the entry with the fileName as a mask
    error = fmsGetNextDirEntry(&index, fileName, dirEntry, CDIR);

    // If it is not found, return not found instead of end of directory
    return (error ? ((error == E_END_OF_DIRECTORY) ? E_FILE_NOT_DEFINED : error) : 0);
} // end fmsGetDirEntry

// *****************************************************************************
//  This function checks a name and extension to see if it matches a mask.
//    Return 1 for success, otherwise, return 0.
//
int fmsMask(char* mask, unsigned char* name, unsigned char* extension)
{
    // Variables
    int maskIndex;                /* This is the current character in the mask */
    int nameIndex = 0;            /* This is the current character in the name */
    int extensionIndex = 0;        /* This is the current character in the extension */
    bool checkingName = TRUE;    /* This is to see if I'm comparing to name or extension */

    // Convert the mask to upper case
    convertToUpperCase(mask);

    // First iterate through mask
    for (maskIndex = 0; maskIndex < (int)strlen(mask); maskIndex++)
    {
        char currentChar = mask[maskIndex];

        //printf("checkingName: %d & maskIndex: %d currentChar: %c & nameIndex: %d nameChar: %c ", checkingName,maskIndex,currentChar,nameIndex, name[nameIndex]);
        //printf ("& extensionIndex: %d extensionChar: %c\n",extensionIndex,extension[extensionIndex]);

        // If the mask is longer than the face and extension, it does not match
        if (checkingName)
        {
            if ((name[nameIndex] == 0x20) && ((currentChar != '*') && (currentChar != '.'))) return 0;
        }
        else
        {
            if ((extension[extensionIndex] == 0x20) && (currentChar != '*')) return 0;
        }

        // Check to see if the currentChar is an asterisk
        if (currentChar == '*')
        {
            // If this is the last character in the mask, return success
            if (maskIndex == strlen(mask)-1)
            {
                // If you have already checked the name, return true
                if (!checkingName) return 1;

                // If you are checking the name, only return true if there is no extension
                if (extension[0] == 0x20) return 1;
                else return 0;
            }
            else
            {
                // Check to see if you have already for a name
                if (!checkingName)
                {
                    // Every extension is valid, return success
                    return 1;
                }

                // Check to see if the next character is a period
                if (mask[maskIndex+1] == '.')
                {
                    // If the next character is a '.', skip to extension
                    checkingName = FALSE;

                    // Increment maskIndex because you don't care about the period
                    maskIndex++;
                }
                else
                {
                    // Else, advance name or extension to the next character in the mask
                    if (checkingName)
                    {
                        // Check to see if that character is in the name
                        if (!(nameIndex = advanceString(name, nameIndex, mask[maskIndex+1])))
                        {
                            break;
                        }
                    }
                    else
                    {
                        // Check to see if that character is in the extension
                        if (!(extensionIndex = advanceString(extension, extensionIndex, mask[maskIndex+1])))
                        {
                            continue;
                        }
                    }

                    // Check return value of advancing to know if it was a success or not
                }
            }
        }
        else if (currentChar == '?')
        {
            // Skip character
            if (checkingName) nameIndex++;
            else extensionIndex++;
        }
        else if ((currentChar == '.') && (strcmp(mask,".")) && (strcmp(mask,"..")))
        {
            // Assign checkingName to false
            checkingName = FALSE;

            // Check to see if you are at the end of the comparing file
            if ((nameIndex < 7) && (name[nameIndex] != 0x20)) return 0;
            else continue;
        }
        else
        {
            // Check the indexed character to make sure it matches
            if (checkingName)
            {
                if (name[nameIndex] != currentChar) return 0;
                else nameIndex++;
            }
            else
            {
                if (extension[extensionIndex] != currentChar) return 0;
                else extensionIndex++;
            }
        }
    }

    // The mask matches at least part of the name
    if (checkingName)
    {
        // If the next character isn't a space
        if ((nameIndex < 8) && (name[nameIndex] != 0x20)) return 0;

        // The names match, but make sure there is no extension
        if (extension[0] != 0x20) return 0;
    }
    else
    {
        // At least part of the extension matches, but make sure all of it
        if ((extensionIndex < 3) && (extension[extensionIndex] != 0x20)) return 0;
    }

    return 1;
}

// *****************************************************************************
// This function is used to advance the string to the passed character
// return the new index if successful, return 0 if it doesn't contain
// that character
//
int advanceString(unsigned char* string, int curIndex, char compare)
{
    return 0;
}

// *****************************************************************************
// This function converts a passed string to upper case
//
void convertToUpperCase(char * str)
{
    int ch, i;

    for(i = 0; i < (int)strlen(str); i++)
    {
        ch = toupper(str[i]);
        str[i] = ch;
    }
}

// *****************************************************************************
// This function helps in removal of folder names
//
void removeFolderName(char* name)
{
    // Variables
    int index = strlen(name) - 1;

    // Remove last slash
    name[index] = 0;

    // Begin traversal
    for (index -= 1; index > 0; index--)
    {
        if (name[index] == '\\')
        {
            break;
        }
        else
        {
            name[index] = 0;
        }
    }
}

// *****************************************************************************
// This function is used to get null-terminated versions of the name and extension
// strings in a DirEntry
//
void getEntryStrings(DirEntry* dirEntry, unsigned char* dirEntryName, unsigned char* dirEntryExtension)
{
    // Variables
    int i;

    // Begin copy of the name array
    for (i = 0; i < 8; i++)
    {
        // Check to see the next character
        if (dirEntry->name[i] == 0x20)
        {
            // If it is a space, it is fully copied
            dirEntryName[i] = 0;
            break;
        }
        else
        {
            // If it is not a space, copy it
            dirEntryName[i] = dirEntry->name[i];
        }
    }

    // Begin copy of the extension array
    for (i = 0; i < 3; i++)
    {
        // Check to see the next character
        if (dirEntry->extension[i] == 0x20)
        {
            // If it is a space, it is fully copied
            dirEntryExtension[i] = 0;
            break;
        }
        else
        {
            // If it is not a space, copy it
            dirEntryExtension[i] = dirEntry->extension[i];
        }
    }
}

// *****************************************************************************
// This function is used to get null-terminated versions of the name and extension
// strings in a DirEntry
```

```c
//
void getFDEntryStrings(FDEntry* dirEntry, unsigned char* dirEntryName, unsigned char* dirEntryExtension)
{
    // Variables
    int i;

    // Begin copy of the name array
    for (i = 0; i < 8; i++)
    {
        // Check to see the next character
        if (dirEntry->name[i] == 0x20)
        {
            // If it is a space, it is fully copied
            dirEntryName[i] = 0;
            break;
        }
        else
        {
            // If it is not a space, copy it
            dirEntryName[i] = dirEntry->name[i];
        }
    }

    // Write a 0
    dirEntryName[i] = 0;

    // Begin copy of the extension array
    for (i = 0; i < 3; i++)
    {
        // Check to see the next character
        if (dirEntry->extension[i] == 0x20)
        {
            // If it is a space, it is fully copied
            dirEntryExtension[i] = 0;
            break;
        }
        else
        {
            // If it is not a space, copy it
            dirEntryExtension[i] = dirEntry->extension[i];
        }
    }

    // Write a 0
    dirEntryExtension[i] = 0;
}

// *****************************************************************************************
// This function checks to see if a file name is valid
//   bool maskable lets you pass *'s and ?'s
//   char* name is the string you are checking
//     Return 1 for success, otherwise, return 0.
//
int validateAndParse(bool maskable, char* fileName, char* name, char* extension)
{
    // Variables
    int nameIndex, extensionIndex;
    int nameInc = 0, extensionInc = 0;
    char currentChar;
    bool hasQuotes = FALSE;
    name[0] = 0;
    extension[0] = 0;

    // Check to see if it is '.' or '..'
    if (!(strcmp(fileName,".")) || !(strcmp(fileName,"..")))
    {
        // Assign the name
        for (nameIndex = 0; nameIndex < (int)strlen(fileName); nameIndex++)
        {
            name[nameIndex] = '.';
        }

        // Assign the null termination
        name[nameIndex] = 0;
        extension[0] = 0;

        // Return success
        return 1;
    }

    // Parse the name
    for (nameIndex = 0; nameIndex < (8+nameInc); nameIndex++)
    {
        // Get the current char
        currentChar = fileName[nameIndex];

        // Check for bad characters
        if ((currentChar == ';') || (currentChar == ':') || (currentChar == '\"') || (currentChar == '\'')) return 0;
        if ((currentChar == '*') || (currentChar == '\\') || (currentChar == '<') || (currentChar == '>')) return 0;
        if ((currentChar == ' ') || (currentChar == '|') || (currentChar == '?') || (currentChar == '+')) return 0;
        if ((currentChar == '=') || (currentChar == '[') || (currentChar == ']') || (currentChar == '/')) return 0;
        if ((currentChar == ',')) return 0;

        // Check if this is a period
        if (currentChar == '.')
        {
            // If this is the first character, return
            if (nameIndex == 0) return 0;

            // Terminate the string
            name[nameIndex-nameInc] = 0;

            // Go do the extension
            break;
        }
        else if (currentChar == '\"')
        {
            // Ignore the quotes
            nameInc++;
```

```
                hasQuotes = (hasQuotes ? FALSE : TRUE);
            }
            else
            {
                // Copy the character
                name[nameIndex-nameInc] = currentChar;

                // If this is the last character, return
                if (currentChar == 0) return 1;
            }

            // This is the last character
            if (nameIndex == (8+nameInc-1))
            {
                // If the next character is not a period or null, return false
                if ((fileName[nameIndex+1] != '.') && (fileName[nameIndex+1] != 0))
                {
                    // This name is too long
                    return 0;
                }
                else if (fileName[nameIndex+nameInc+1] == 0)
                {
                    // Terminate and break
                    name[nameIndex+nameInc+1] = 0;
                    break;
                }
                else
                {
                    // Terminate and increase nameIndex
                    name[nameIndex+nameInc+1] = 0;
                }
            }
        }

        // Parse the extension
        for (extensionIndex = 0; extensionIndex < (3+extensionInc); extensionIndex++)
        {
            // Get the current char
            currentChar = fileName[nameIndex + extensionIndex + 1];

            // Check for bad characters
            if ((currentChar == ';') || (currentChar == ':') || (currentChar == '\"') || (currentChar == '\'')) return 0;
            if ((currentChar == '*') || (currentChar == '\\') || (currentChar == '<') || (currentChar == '>')) return 0;
            if ((currentChar == ' ') || (currentChar == '|') || (currentChar == '?') || (currentChar == '+')) return 0;
            if ((currentChar == '=') || (currentChar == '[') || (currentChar == ']') || (currentChar == '/')) return 0;
            if ((currentChar == '.') || (currentChar == ',')) return 0;

            if (currentChar == '\"')
            {
                // Ignore the quotes
                extensionInc++;
                hasQuotes = (hasQuotes ? FALSE : TRUE);
            }
            else
            {
                // Copy the character
                extension[extensionIndex-extensionInc] = currentChar;

                // If this is the last character, return
                if (currentChar == 0) return 1;
            }

            // If the index is 2 and the next character is not a null, return fal
    se;
            if (extensionIndex == (3+extensionInc-1))
            {
                // Check to see if this name is too long
                if ((strlen(fileName) - nameIndex -1) > 3)
                {
                    // This name is too long
                    return 0;
                }
                else
                {
                    // Add the null character
                    extension[extensionIndex-extensionInc+1] = 0;
                }
            }
        }

        // If the quotes haven't ended, error
        if (hasQuotes) return 0;

        // Return success
        return 1;
    }


    // ***********************************************************************
    *************
    // This function returns the starting cluster
    //
    int getEndCluster(int dir, int dirNum)
    {
        // Variables
        int dirCluster = 0;
        int i = 0;

        // Check where you are
        if (dir)
        {
            // If you are not looking at the root directory, traverse
            dirCluster = dir;
            for (i = 0; i < (dirNum / ENTRIES_PER_SECTOR); i++)
            {
                dirCluster = getFatEntry(dirCluster, FAT1);
            }
        }
        else
        {
            // Root directory
            dirCluster = (dirNum / ENTRIES_PER_SECTOR);
        }

        return dirCluster;
    }

    /**
     * fmsCloseFile - close an open file
     * @fileDescriptor: the id of the open file to close
     * @return: 0 for success, error number otherwise
     *
     * This function closes the open file specified by fileDescriptor.
     * The fileDescriptor was returned by fmsOpenFile and is an index
     * into the open file table.
     */
    int fmsCloseFile(int fileDescriptor)
    {
        if (!diskMounted)
            return E_DISK_NOT_MOUNTED;
        if (fileDescriptor < 0)
            return E_INVALID_FILE_DESCRIPTOR;

        FDEntry* fdEntry = &OFTable[fileDescriptor];

        if (fdEntry->name[0] == 0)
            return E_FILE_NOT_OPEN;

        FDEntry* fdEntry = &OFTable[fileDescriptor];
        char fileName[12];
        for (i = 0; i < 12; i++) {
            fileName[i] = fdEntry->name[i];
        }

        DirEntry* dirEntry;
        int error;
        if (error = fmsGetDirEntry(fileName, dirEntry))
            return error;

        if (fdEntry->flags == FILE_ALTERED) {
            setDirTimeDate(dirEntry);
            dirEntry->fileSize = fdEntry->fileSize;
        }

        // Flush buffer
        fmsWriteSector(fdEntry->buffer, C_2_S(fdEntry->currentCluster);

        return 0; // return success
    }

    /**
     * fmsDefineFile - creates a new file in the current directory
     * @fileName: the name of the file to create
     * @attribute: the type of file to create
     *
     * If attribute=DIRECTORY, this function creates a new directory fileName
     * in the current directory. The directory entries "." and ".." are also
     * defined. It is an error to try and create a directory that already exists.
```

```c
 *
 * Else, this function creates a new file fileName in the current directory.
 * It is an error to try and create a file that already exists.
 * The start cluster field should be initialized to cluster 0.  In FAT-12,
 * files of size 0 should point to cluster 0 (otherwise chkdsk should report
 * an error). Remember to change the start cluster field from 0 to a free
 * cluster when writing to the file.
 */
int fmsDefineFile(char* fileName, int attribute)
{
    return 0;
}

/**
 *fmsDeleteFile - deletes fileName from the current directory
 * @fileName: the name of the file to delete
 * @return: 0 for success, error number otherwise
 *
 * This function deletes the file fileName from the current directory. The
 * file name should be marked with an "E5" as the first character and the
 * chained clusters in FAT 1 reallocated (cleared to 0).
 */
int fmsDeleteFile(char* fileName)
{
    return 0;
}

/**
 * fmsOpenFile - opens a file with specified access mode
 * @fileName: the name of the file to open
 * @rwMode: the mode of the open file
 * @return: If successful, return file descriptor (index into open file table
),
 *              error number otherwise
 *
 * This function opens the file fileName for access as specified by rwMode.
 * It is an error to try to open a file that does not exist.
 * The open mode rwMode is defined as follows:
 *   0 - Read access only.
 *      The file pointer is initialized to the beginning of the file.
 *      Writing to this file is not allowed.
 *   1 - Write access only.
 *      The file pointer is initialized to the beginning of the file.
 *      Reading from this file is not allowed.
 *   2 - Append access.
 *      The file pointer is moved to the end of the file.
 *      Reading from this file is not allowed.
 *   3 - Read/Write access.
 *      The file pointer is initialized to the beginning of the file.
 *       Both read and writing to the file is allowed.
 *
 * A maximum of 32 files may be open at any one time.
 */
int fmsOpenFile(char* fileName, int rwMode)
{
    // Check permission
    // Spew errors ("Invalid File Name", "File Not Defined",
    // "File Already open", "Too Many Files Open", "File Space Full"

    int error, i, j, fd = 0;
    DirEntry dirEntry;
    FDEntry* fdEntry;
    TCB* tcb = getTCB();                                // For use by CDIR macro
    int curTask = gettid();                                // For use by CDIR macro

    if (error = fmsGetDirEntry(fileName, &dirEntry))     // Returned dir entry stored in dirEntry
        return error;
    if (dirEntry->fileName[0] == 0)
        return E_INVALID_FILE_NAME;
    if (dirEntry.attributes == READ_ONLY && rwMode)
        return E_READ_ONLY_FILE;

    for (i = 0, fd = -1; i < NFILES; i++) {                // Look through entire open file table
        fdEntry = &OFTable[i];

        if (fdEntry->name[0] == 0)                        // Open slot (not too many files
            fd = i;                                        // already open), so save that location

                                                        // But we still need to iterate over
        // 12 because name and extension are contig.    // rest of entries in OFTable to
        for (j = 0; j < 12; j++) {                        // compare the name of file in the file
            if (fdEntry->name[j] != dirEntry.name[j])    // descriptor we're checking right now
                break;    //inner for, check next entry     // to name of file in directory entry,
            if (j == 12)                                // to make sure that the
                return E_FILE_ALREADY_OPEN;                // file isn't already open
        }
    }
    if (fd == -1)                                        // fd never got assigned to an open slot
        return E_TOO_MANY_FILES_OPEN;                    // so, too many files open

    memcpy(fdEntry->name, dirEntry.name, 8);             // Create new file descriptor entry
    memcpy(fdEntry->extension, dirEntry.extension, 3);    // TODO check about null-termination
    fdEntry->attributes = dirEntry.attributes;
    fdEntry->directoryCluster = CDIR;
    fdEntry->startCluster = dirEntry.startCluster;
    fdEntry->currentCluster = !rwMode ? fdEntry->startCluster : 0;     // At least for part 1
    fdEntry->fileSize = (rwMode == 1) ? 0 : dirEntry.fileSize; // If writing to file, 0 size
    fdEntry->pid = curTask;                                         // curTask
    fdEntry->mode = rwMode;
    fdEntry->flags = 0;
    fdEntry->fileIndex = (rwMode != 2) ? 0 : dirEntry.fileSize; // If appending file, go to end
    memset(fdEntry->buffer, -2, BUFSIZE);

    if (rwMode == 2) {                                              // Appending, fill buff with last cluster
        fdEntry->currentCluster = fdEntry->startCluster;
        unsigned short nextCluster = 0;
        while ((nextCluster = getFatEntry(fdEntry->currentCluster, FAT1)) != FAT_EOC)
            fdEntry->currentCluster = nextCluster;
        if ((error = fmsReadSector(fdEntry->buffer, C_2_S(fdEntry->currentCluster))))
            return error;
    }

    return fd;
}

/**
 * fmsReadFile - read a specified number of bytes from a file
 * @fileDescriptor: the file descriptor of the open file
 * @buffer: the buffer to load the read data into
 * @nBytes: the number of bytes to read
 * @return: the number of bytes read, error number otherwise
 *
 * This function reads nBytes bytes from the open file specified by
 * fileDescriptor into memory pointed to by buffer. The fileDescriptor was
 * returned by fmsOpenFile and is an index into the open file table. After
 * each read, the file pointer is advanced.
 */
int fmsReadFile(int fileDescriptor, char* buffer, int nBytes)
{
    if (!diskMounted)
        return E_DISK_NOT_MOUNTED;
    if (fileDescriptor < 0)
        return E_INVALID_FILE_DESCRIPTOR;

    FDEntry* fdEntry = &OFTable[fileDescriptor];

    if (fdEntry->name[0] == 0)
        return E_FILE_NOT_OPEN;

    unsigned short nextCluster = 0;
    int error;

    // If the buffer for this fd is empty (either we're entering
    // a new sector and just cleared the buffer in a previous
    // iteration of this loop, or it's never been filled), fill it
    if (fdEntry->buffer[0] == -2) {
        if (error = fmsReadSector(fdEntry->buffer, C_2_S(fdEntry->currentCluster)));
            return error;
        // So we're at the next cluster next time for next time
        nextCluster = getFatEntry(fdEntry->currentCluster, FAT1);
        fdEntry->currentCluster = nextCluster;
    }

    memcpy(buffer, fdEntry->buffer + (fdEntry->fileIndex % BYTES_PER_SECTOR), nBytes);
    fdEntry->fileIndex = fdEntry->fileIndex + nBytes;

    if (fdEntry->fileIndex % BYTES_PER_SECTOR == 0) {
        memset(fdEntry->buffer, -2, BUFSIZE);
    }

    if (fdEntry->fileIndex >= fdEntry->fileSize)
        return E_END_OF_FILE;

    return nBytes;
}

/**
 * fmsSeekFile - change the current file pointer of an open file
 * @fileDescriptor: the file descriptor of the open file
 * @index: the new file position
 * @return: the new position in the file, error number otherwise
 *
 * This function changes the current file pointer of the open file specified
 * by fileDescriptor to the new file position specified by index. The
 * fileDescriptor was returned by fmsOpenFile and is an index into the open
 * file table. The file position may not be positioned beyond the end of the
 * file.
 */
int fmsSeekFile(int fileDescriptor, int index)
{
    if (!diskMounted)
        return E_DISK_NOT_MOUNTED;
    if (fileDescriptor < 0)
        return E_INVALID_FILE_DESCRIPTOR;

    FDEntry* fdEntry = &OFTable[fileDescriptor];

    // This fdEntry is still free/hasn't been allocated
    if (fdEntry->name[0] == 0)
        return E_FILE_NOT_OPEN;             // Could also be E_FILE_NOT_DEFINED...?
    if (index >= fdEntry->fileSize)
        return E_FILE_SEEK_ERROR;

    fdEntry->fileIndex = index;

    // Get to the right cluster, put part of cluster up to index into fd's buffer
    char buf[BUFSIZE];
    int i;
    fdEntry->currentCluster = fdEntry->startCluster;
    for (i = 0; i < (index / BYTES_PER_SECTOR); i++) {
        unsigned int nextCluster = getFatEntry(fdEntry->currentCluster, FAT1);
        fdEntry->currentCluster = nextCluster;
    }
    if ((error = fmsReadSector(buf, C_2_S(fdEntry->currentCluster))))
```

```c
        return error;

    memcpy(fdEntry->buffer, buff, (index % BYTES_PER_SECTOR));

    return index;
}

/**
 * fmsWriteFile - write to an open file
 * @fileDescriptor: the file descriptor of the open file
 * @buffer: the data to write to a file
 * @nBytes: the number of bytes to write
 * @return: the number of bytes written, error number otherwise
 *
 * This function writes nBytes bytes to the open file specified by
 * fileDescriptor from memory pointed to by buffer. The fileDescriptor was
 * returned by fmsOpenFile and is an index into the open file table.
 * Writing is always "overwriting" not "inserting" in the file and always
 * writes forward from the current file pointer position.
 */
int fmsWriteFile(int fileDescriptor, char* buffer, int nBytes)
{
    return 0;
}


// *************************************************************************************
// *************************************************************************************
// *************************************************************************************
// *************************************************************************************
// *************************************************************************************


// *************************************************************************************
// Take a FAT table index and return an unsigned short containing the 12-bit FAT entry code
// *************************************************************************************
// Take a FAT table index and return an unsigned short containing the 12-bit FAT entry code
unsigned short getFatEntry(int FATindex, unsigned char* FATtable)
{
    unsigned short FATEntryCode;                /* The return value */
    int FatOffset = ((FATindex * 3) / 2);    /* Calculate the offset of the unsigned short to get */
    if ((FATindex % 2) == 1)                    /* If the index is odd */
    {
        // Pull out a unsigned short from a unsigned char array
        FATEntryCode = *((unsigned short *)&FATtable[FatOffset]);
      FATEntryCode = SWAP_BYTES(FATEntryCode);
        FATEntryCode >>= 4;                       /* Extract the high-order 12 bits */
    }
    else                                                /* If the index is even */
    {
        // Pull out a unsigned short from a unsigned char array
        FATEntryCode = *((unsigned short *)&FATtable[FatOffset]);
      FATEntryCode = SWAP_BYTES(FATEntryCode);
      FATEntryCode &= 0x0fff;                /* Extract the low-order 12 bits */
    }
    return FATEntryCode;
} // end GetFatEntry



// *****************************************************************************
// *****************************************************************************
// Replace the 12-bit FAT entry code in the unsigned char FAT table at index
void setFatEntry(int FATindex, unsigned short FAT12ClusEntryVal)
{
    int FATOffset = ((FATindex * 3) / 2);    /* Calculate the offset */
    int FATData = *((unsigned short*)&FAT1[FATOffset]);
    FATData = SWAP_BYTES(FATData);
    if (FATindex % 2 == 0)                        /* If the index is even */
    {    FAT12ClusEntryVal &= 0x0FFF;            /* mask to 12 bits */
    FATData &= 0xF000;                            /* mask complement */
    }
    else                                                /* Index is odd */
    {    FAT12ClusEntryVal <<= 4;                 /* move 12-bits high */
        FATData &= 0x000F;                        /* mask complement */
    }
    // Update FAT entry value in the FAT table
    FATData = SWAP_BYTES(FATData);
    *((unsigned short *)&FAT1[FATOffset]) = FATData | FAT12ClusEntryVal;
} // End SetFatEntry


// *****************************************************************************
//    setDirTimeDate
//
//        struct tm
//        {
//            int tm_sec;     // 0 to 60
//            int tm_min;     // 0 to 59
//            int tm_hour;    // 0 to 23
//            int tm_mday;    // 1 to 31
//            int tm_mon;     // 0 to 11
//            int tm_year;    // year - 1900
//            int tm_wday;    // Sunday = 0
//            int tm_yday;    // 0 to 365
//            int tm_isdst;   // >0 if Daylight Savings Time,
//                            //  0 if Standard,
//                            // <0 if unknown
//            char *tm_zone;  // time zone name
//            long tm_gmtoff; // offset from GMT
//        };
//
void setDirTimeDate(DirEntry* dir)
{
   time_t a;
   struct tm *b;

   time(&a);
   b = localtime(&a);
   dir->date.year = b->tm_year + 1900 - 1980;
   dir->date.month = b->tm_mon;
   dir->date.day = b->tm_mday;

   dir->time.hour = b->tm_hour;
   dir->time.min = b->tm_min;
   dir->time.sec = b->tm_sec / 2; // FAT16 time resolution is 2 seconds (only
 5 bits allocated to seconds);
   return;
} // end setDirTimeDate



// ****************************************************************************
*************
// Error processor
void fmsError(int error)
{
   int i;

   for (i=0; i<NUM_ERRORS; i++)
   {
      if (FMSErrors[i].error == error)
      {
         printf("%s\n", FMSErrors[i].error_msg);
         return;
      }
   }
   printf("%s %d\n", E_UNDEFINED_MSG, error);
   return;
} // end fmsError



// ****************************************************************************
*************
int fmsMount(char* fileName, void* ramDisk)
//    Called by mount command.
// This function loads a RAM disk image from a file.
//    The parameter fileName is the file path name of the disk image.
//    The parameter ramDisk is a pointer to a character array whose
//    size is equal to a 1.4 mb floppy disk (2849 ´ 512 bytes).
//    Return 0 for success, otherwise, return the error number
{
   FILE* fp;
   fp = fopen(fileName, "rb");
   if (fp)
   {
      fread(ramDisk, sizeof(char), SECTORS_PER_DISK * BYTES_PER_SECTOR, fp);
   }
   else return -1;
   fclose(fp);
    // copy FAT table to memory
    memcpy(FAT1, &RAMDisk[1 * BYTES_PER_SECTOR], NUM_FAT_SECTORS * BYTES_PER_SECTOR);
    memcpy(FAT2, &RAMDisk[10 * BYTES_PER_SECTOR], NUM_FAT_SECTORS * BYTES_PER_SECTOR);
    diskMounted = 1;                  /* disk has been mounted */
    //@DISABLE_SWAPS
    strcpy(dirPath, fileName);
    strcat(dirPath, ":\\");
    return 0;
} // end fmsMount
//@ENABLE_SWAPS



// *****************************************************************************
int fmsUnMount(char* fileName, void* ramDisk)
// Called by the unmount command.
// This function unloads your Project 5 RAM disk image to file computer file.
// The parameter fileName is the file path name of the disk image.
// The pointer parameter ramDisk points to a character array whose size is equal to a 1.4
// mb floppy disk (2849 ´ 512 bytes).
// Return 0 for success; otherwise, return the error number.
{
    diskMounted = 0;                              /* unmount disk */
    return -1;
} // end



// *****************************************************************************
int fmsReadSector(void* buffer, int sectorNumber)
//    Read into buffer RAM disk sector number sectorNumber.
// Sectors are 512 bytes.
//    Return 0 for success; otherwise, return an error number.
{
    memcpy(buffer, &RAMDisk[sectorNumber * BYTES_PER_SECTOR], BYTES_PER_SECTOR);
   return 0;
} // end fmsReadSector

// *****************************************************************************
int fmsWriteSector(void* buffer, int sectorNumber)
// Write 512 bytes from memory pointed to by buffer to RAM disk sector sectorNumber.
// Return 0 for success; otherwise, return an error number.
{
    memcpy(&RAMDisk[sectorNumber * BYTES_PER_SECTOR], buffer, BYTES_PER_SECTOR);
    return 0;
} // end fmsWriteSector
```