An Array Based Filesystem for a PIC Micro-controller

Alex Olson

12/03/2002

Abstract

In this document, I present an array based filesystem that can be implemented on micro-controllers with only a very minuscule amount of RAM. This filesystem was designed for an portable MP3 Player, where files are rarely changed or deleted. The burden of the filesystem maintenance is placed on the host computer, not the controller. This system also allows files to be accessed either sequentially or randomly in constant time by the micro-controller.

1 Problems with FAT32

Why FAT32 isn't great for everything

The FAT32 file system is a fairly simple system because it is based on a linked-list structure. Whether a directory or a file is stored, it is divided into clusters. Each cluster itself need not be 'near' any other cluster. There is a table (the File Allocation Table) of 32-bit entries that, given a cluster number, determines the next cluster. A cluster is a group of consecutive sectors, and on a hard-drive, a sector is usually 512 bytes. Microsoft specifies that the number of sectors per cluster must be a power of two, but no more than 128 [1]pg. 9. I have determined that my home computer uses 16 sectors per cluster, which is equivalent to 8192 bytes per cluster. Since the average mp3 audio file is approximately 5-10 megabytes, this implies that there would be at least 640 clusters for one file. Unfortunately the PIC micro-controller only has 1.5 Kbytes of RAM and it cannot be interfaced to external memory. There would be no way to store 640 32bit values on it. Worse yet, only 128 cluster entries can fit on a single sector in the File Allocation Table. If the filesystem is badly fragmented, the series of entries in the FAT for one file may not even be on the same sector. Harddrives generally have very high access times, in the range of 10-20ms. If the entire table cannot be stored in RAM, several disk accesses would be required. This causes for a serious performance penalty.

The directory structure of the FAT32 filesystem is also similar to a linked list. A directory and a file are treated no differently. The 'data'of a directory is the directory listing itself! The directory contains 32 byte entries (16 per sector) that tell things like filename, file attributes, file size, first data cluster, etc. The entries in a directory are not stored in any particular order and are not necessarily all contiguous (there may be free entries anywhere). If the file were being read, the data of the file pointed to by the first cluster entry would be read. Then the FAT would be

consulted to find the next data cluster. It would be read, then the FAT would be consulted again to find the next cluster, etc...

One problem with the FAT structure is that it is only a *singly* linked list. If the contents of a file were being read and there was a need to access an (earlier) already read portion, there would be no way to do it just from the information contained in a FAT32 volume. One would have maintain a list or stack that contained all the clusters that had been entered. But the problem with micro-controllers is that there isn't enough space to store a large enough stack to hold the all of the cluster numbers for the data of even ONE mp3 file!

Even worse, most CD & MP3 players have a 'Random Song 'capability. To seek to a random file on a FAT32 volume, the entire list of files (at least locations of their file entries) would need to be maintained in memory. The alternative of repeatedly accessing the disk to find the 'next' file is not a realistic approach if the volume contains several thousand files and the disk access time is in the tens of milliseconds.

These problems arise from the fact that FAT32 filesystem was designed for use on PCs, where files are constantly being changed and modified, and where ample memory exists. The linked-list structure allows file of varying sizes to be stored without the need of contiguous blocks of free space. For an application such as an MP3 Player, or really any device that needs to access a large amount of data without the need to modify it, the FAT32 filesystem is not necessarily ideal.

What follows in this document is an array based filesystem. It requires less than 1k of memory. Files can be accessed either sequentially or randomly in constant time. Virtually the RAM is only used to store the current sector buffer, the current directory entry, and the current file entry. No other tables need to be stored.

2 A Simpler File System

Directory Structure

The use of directories in a file system can be very useful. They provide a way to organize a large number of files into more manageable units. But for a micro-controller, they are a nuisance. Building a list of all files on a drive with directories usually requires some kind of recursive process. In my file system, there are two 'special' directories – the *TOP* directory and the *ROOT* directory. The names have slightly different meanings than the usual convention. The *TOP* directory contains a list of all directories on the drive (it cannot contain files). It always has at least on entry, the *ROOT* directory. The *ROOT* directory contains entries for *every* file on the volume, regardless in which directory(ies) it is stored. For simplicity, I do not currently have any provision of storing directories anywhere outside the *TOP* directory. If there are other directories in the volume, they will also contain entries for their files, just like the *ROOT* directory. This implies that files will often be 'cross-linked'. The entries within every directory are contiguous; there are no gaps between successive entries and the entries are located on consecutive sectors.

Table 1: Anatomy of a file entry

Bytes	Name	Purpose
1	entryType	specifies either ENTRY_FILE or ENTRY_DIRECTORY
1	nameLen	length of name (filename or directory name)
4	firstSector	specifies beginning of data
4	lastSector	specified last sector of data
4	size	# of bytes in a file, # of files in a directory
4	special	for special purposes
192	name	file/directory name

(210 Total Bytes for 1 file entry)
For the *TOP* directory entry (sector 0), this *special* field specifies the maximum number of directory entries for this volume. For all others, it is currently undefined (and should be set to zero).

Table 2: Overview of Entire Drive

Sector	Type Description
0	file_entry: contains *TOP* directory info
1-256	file_entry: content of *TOP* directory (directory entries)
257-8488	file_entry: content of *ROOT* directory (directory entries)
8489- 131072	file_entry: content of other directories (directory entries)
131073 - x1	data: content of first file
x1+1 - x2	data: content of second file
(etc)	

Anatomy of a file entry

The following table shows the structure of a file entry. This structure is used to store both file and directory entries.

Although each directory entry uses only 210 bytes, there is only one directory entry per sector. The remaining space could be used for future expansion. In the case of storing MP3 music, fields such as 'artist', 'type', etc could be added. The example above assumes that the *TOP* directory may have a maximum of 256 entries, the *ROOT* directory a maximum of 8192 entries, and total maximum of 131072 (8192*16) entries. In the case of my mp3 player, the directory entries in the *TOP* directory could be thought of as 'playlists', so 256 possible entries is more than sufficient. Above, I allocate 8192 entries for the root directory. I currently am using a 10GB drive and could realistically only fit about 2048 entries; so 8192 possible entries should be enough.

Benefits of this file system

The simplicity of this file system is what makes it so fast. A 32 bit integer may be used to keep track of the current directory entry (sector). The entire 210byte entry for the current directory could be stored in memory along with the 210byte entry for the current file. A 32bit integer could be used as pointer to refer to a specific entry within a directory. It could be incremented or decremented to move about the directory. As soon as it equals the *firstSector* has been incremented successively *s*ize times, then it would be known that the beginning or end of the directory had been reached. Seeking directly to a particular index in the directory is easy since both the directory entry count and the range of sectors are known directly from the current directory entry. Since the *ROOT* directory contains all files in the drive, searching for a particular file is simple and requires no recursion. If all entries were stored in alphabetical order, a binary search could be attempted. Likewise, seeking to a 'random'song is simple, since the location of a song can be computed directly.

Drive Formatting

To format the drive, only a few simple things need to be done. Three values need to be known, the maximum number of file entries in: the *TOP* directory, the *ROOT* directory, and the grand total for the drive. Generally the number of entries in the *TOP* directory will be small, perhaps less than 100. The number of entries in the *ROOT* directory determines how many files can be stored on the drive. The number of entries for the grand total determines how large the other directories in the *TOP* directory can be. For example to allow up to 16 directory entries in the *TOP* directory, and 1000 files in the *ROOT* directory, there should be space for a little more than 16000 entries. This would allow the root directory and 15 other directories, each also containing every file on the drive. Since the idea of a directory is to usually not include every file on the drive, the grand total number of entries can be set lower than the product of the number of entries in the *TOP* directory and the number of entries in the *ROOT* directory.

Once these values are known, all that remains is to write the 0th sector with the *TOP* directory entry. Generally, the *TOP* directory would begin at sector 1, but this is not a requirement. Then the *ROOT* directory can be created to hold the number of entries desired. The drive is then read to accept files.

Creating a directory

To create a directory (including the root directory), the *TOP* directory should be examined to see in what sector the last directory entry ends. If the *TOP* directory is empty, then sector 1 could be used. An entry is written to this sector that specifies the beginning and ending sectors that this new directory spans (determined by desired number of entries it could hold). Its *size* is set to zero to signify that there are no files contained within.

Creating a file

To create a directory, the *ROOT* directory should be examined to see what sector the last file entry ends. The sum of 1 and the *lastSector* value of the last file entry determines the value of the first

sector of the new file. If the *ROOT* directory is empty, then the *special* value contained in the 0th sector specifies the first file data sector. A free directory entry sector is found in the *ROOT* directory, using the sum of *size* and *firstSector* values.

Deleting a file

The only downside to this approach is that deleting files is not a simple task. To delete a file and recover the space used by it, all entries and file data would have to be moved and updated. A better, but more complicated algorithm could find a file near the end of the drive that is slightly smaller and move it to fill the gap, leaving most of the other files unmoved. Then, only a small amount of space would be lost (caused by the difference in file sizes) by the fragmentation of the free space that would result. In either case, the purpose of this file system is that it should be used in situations where file deletions rarely (if ever) occur.

Deleting a directory

In theory, a directory could be deleted in the same manner as a file. The only minor difference is where they store their 'data'.

3 The Code

What it Does

Below is some code that can format a drive, create a directory, add a file, read a file, and list all files. The routines in low.h match interface I wrote for my mp3 player to control the harddrive. Since I have a C compiler for the microcontroler I used, I should be able to move the code contained in fs.h (and fs.c directly to the microcontroler with no changes. All functions that return a boolean return *true* if successful and *false* if not successful. The datatype unsigned long designates a 32bit unsigned integer. The driver, main.c allows all of the implemented operations to be tested on a file. In practice, they would operate on the drive itself (after a mere change to one routine in low.c. Thus, the program listed below is capable of formating and loading a harddrive with files for use in my mp3 player.

Program Listing

```
// ======================================================
// MP3 Filesytem Implementation
// Alex Olson
// Fall 2002
// low.c
// ======================================================

#include "low.h"
#include <string.h>

unsigned char hd_sector[512];
#define FILENAME "MYFS.FS"
FILE *inp=NULL;
```

```
void HD_Init() //decl
{
        inp = fopen (FILENAME,"r+b");
        if (!inp)
        {
                printf("HD_Init: Could not open input file!\n");
                exit(1);
        }

}


void HD_WaitBusy()//decl
{
        return;
}



void HD_Read512()//decl
{
        HD_WaitBusy();
        clearerr(inp);
        memset(hd_sector,0,512);
        fread(hd_sector,512,1,inp);
        if (ferror(inp))
        {
                printf("HD: Low-level read error!\n");
        }
}

void HD_Write512()
{
}

void HD_SeekSector(const unsigned long sec)
{
        unsigned long result;
        unsigned long s;
        const unsigned long big =  2048000000/512;

        clearerr(inp);

        s=sec;
        rewind(inp);

        while (s>0)
        {
                if (s >=big)
                {
                        result = fseek(inp, 512*big,SEEK_CUR);
                        s-=big;
                }
                else
                {
                        result = fseek(inp, s*512,SEEK_CUR);
                        s=0;
                }
                if (result != 0 )
                {
```

```
                        printf("HD_SeekSector: Seek Error for sector # %ld\n",sec);
                        exit(1);
                }
        }
}


void HD_StartWriteSector(const unsigned long sec)
{
        unsigned long n;
        HD_WaitBusy();
        HD_SeekSector(sec);

        n=fwrite(hd_sector, 512, 1, inp);

        if ( n!=1 || ferror(inp))
        {
                printf("HD_StartWriteSector: Write Error!\n");
                exit(1);
        }
        fflush(inp);
        memset(hd_sector,0,512);
}

void HD_StartReadSector(const unsigned long sec)
{
        HD_WaitBusy();
        HD_SeekSector(sec);
}

void HD_Shutdown()
{
        fflush(inp);
        fclose(inp);
}

// ========================================================
// MP3 Filesytem Implementation
// Alex Olson
// Fall 2002
// low.h
// ========================================================

#ifndef LOW_H
#define LOW_H

#include <stdio.h>
#include <stdlib.h>

extern unsigned char hd_sector[512];

void HD_Init();
void HD_WaitBusy();
void HD_StartReadSector(const unsigned long sec);
void HD_Read512();

void HD_Write512();
void HD_StartWriteSector(const unsigned long sec);

void HD_Shutdown();
#endif
```

```
// ========================================================
// MP3 Filesytem Implementation
// Alex Olson
// Fall 2002
// fs.h
// ========================================================

// Header file for FSFS

#include "low.h" // for hd operations
#include <stdio.h>
#include <string.h>


#define NAME_MAX  192
// Allow for 64k entries (including files & playlists)
//#define FileDataStart  65536

#define PACKED __attribute__((packed))  // for linux


enum {ENTRY_UNKNOWN, ENTRY_FILE, ENTRY_DIRECTORY};

enum {false, true};
typedef unsigned char bool;

struct file_entry
{
        unsigned char entryType         PACKED;
        unsigned char nameLen           PACKED;
        unsigned long firstSector       PACKED;
        unsigned long lastSector        PACKED;
        unsigned long size              PACKED;
        unsigned long special           PACKED;
                 char name[ NAME_MAX ]  PACKED;
};

void FS_PrintFile();
bool FS_Init();
bool FS_Shutdown();
bool FS_FormatDrive(const unsigned short numRootEntries);
bool FS_AddFile(const char *pcFilename,         const char *mp3Filename);
bool FS_GetFile(const char *mp3Filename, const char *pcFilename);
bool FS_CreateDirectory(const char dirName[], const unsigned long numEntries);
bool FS_TopDirectory();
bool FS_RootDirectory();
bool FS_IncrementFile();
bool FS_DecrementFile();
bool FS_EnterDirectory();
unsigned long FS_FindFreeDataSector();
bool FS_LoadFileIndex(const unsigned long index);

unsigned long FS_StartReadFirstDataSector();
unsigned long FS_StartReadNextDataSector();
void          FS_FinishReadSector();

bool            FS_CurDirIsEmpty();
char  *FS_CurDirName();
unsigned char   FS_CurDirNameLength();
```

```
unsigned short  FS_CurDirNumEntries();
char  *FS_CurFileName();
unsigned char   FS_CurFileNameLength();
unsigned long   FS_CurFileSize();
unsigned char  *FS_BufferPointer();
unsigned short  FS_BufferSize();


// =====================================================
// MP3 Filesytem Implementation
// Alex Olson
// Fall 2002
// fs.c
// =====================================================

#include "fs.h"


// For accessing a directory
struct file_entry curDir;
// Points to a particular directory sector (in the top)
unsigned long curDirSector;
// points to a particular entry inside a directory
unsigned long curFileEntrySector;

// For accessing a file
struct file_entry curFile;              // points to file currently being read
unsigned long     curFileSector;        // points to a block of file data
unsigned long     curFileBytesLeft;     // For sending data



bool FS_Init()
{
        HD_Init();
        //return FS_RootDirectory();
        return true;
}

bool FS_Shutdown()
{
        HD_Shutdown();
        return true;
}

bool FS_TopDirectory()
{
        curFileEntrySector=0;
        curDir.entryType = ENTRY_DIRECTORY;
        return FS_EnterDirectory();
}

bool FS_RootDirectory()
{
        curFileEntrySector=1;
        curDir.entryType = ENTRY_DIRECTORY;
        return FS_EnterDirectory();
}

// Enters the directory pointed to by curFileEntrySector of the directory curDirEntry
bool FS_EnterDirectory()
```

```
{
        if (curDir.entryType == ENTRY_DIRECTORY )
        {
                curDirSector = curFileEntrySector;              // Remember this directory's sector
                HD_StartReadSector( curFileEntrySector );
                HD_Read512();
                memcpy( &curDir, hd_sector, sizeof(curDir) );   // Copy new directory to current one
                curFileEntrySector = curDir.firstSector;                // point pointer to first entry...

                memset(&curFile, 0, sizeof(curFile) );
                HD_StartReadSector( curFileEntrySector );               // Read First File Entry
                HD_Read512();
                memcpy( &curFile, hd_sector, sizeof(curFile) );

                return true;
        }
        else
        {
                printf("FS_EnterDirectory() That's not a directory!\n");
                return false;
        }

}

// Loads the file entry specified by index of the current folder
bool FS_LoadFileIndex(const unsigned long index)
{
        if (index>= curDir.size)
        {
                printf("FS_LoadFileIndex: index is out of range!\n");
                return false;
        }
        else
        {
                printf("FS_LoadFileIndex: Loading file entry at index %ld\n",index);
                curFileEntrySector = curDir.firstSector + index;
                HD_StartReadSector(curFileEntrySector);
                HD_Read512();
                memcpy(&curFile, hd_sector, sizeof(curFile) );
                return true;
        }
}

// Finds first free sector
// By looking at root directory
unsigned long FS_FindFreeDataSector()
{
        unsigned long f;

        FS_RootDirectory();

        if ( curDir.size == 0 )
        {
                return 0;
        }
        else
        {
                while (FS_IncrementFile());
                f = curFile.lastSector + 1;
                return f;
        }
```

```
}

bool FS_CreateDirectory(const char dirName[], const unsigned long numEntries)
{
        unsigned long freeEntry;
        unsigned long freeSector;

        struct file_entry *dir = (struct file_entry *)hd_sector;

        FS_TopDirectory();      // Everything is created in the top directory

        // Validate Directory and Name length
        if ( curDir.lastSector - curDir.firstSector + 1 == curDir.size )
        {
                printf("FS_CreateDirectory: The top directory is full..\n");
                return 0;
        }
        if ( strlen(dirName) >=NAME_MAX )
        {
                printf("FS_CreateDirectory: Directory name is too long!\n");
                return 0;
        }

        if (curDir.size == 0 ) // if top is empty
        {
                freeSector = curDir.lastSector+1;
                freeEntry  = curDir.firstSector;

        }
        else // Top non-empty, curFile pointing to
        {
                while (FS_IncrementFile());     // Go To Last Entry
                freeSector = curFile.lastSector+1;
                freeEntry  = curFileEntrySector+1;
        }

        memset(hd_sector,0,512);
        // Set New Dir Name
        strcpy( dir->name, dirName);
        dir->nameLen = strlen( dirName);
        dir->entryType   = ENTRY_DIRECTORY;
        dir->firstSector = freeSector;
        dir->lastSector  = freeSector + numEntries -1;
        dir->size        = 0;
        HD_Write512();
        HD_StartWriteSector( freeEntry );

        // Update table for parent directory....
        HD_StartReadSector(curDirSector);
        HD_Read512();
        dir->size++;    // Increment Count
        HD_Write512();
        HD_StartWriteSector(curDirSector);
        curDir = *dir;

        return true;
}


bool FS_FormatDrive(const unsigned short rootEntries)
{
```

```
        const char  rootName[]="root";
        const char   topName[]="top";
        const unsigned long   topEntries=255; // 255 Playlists
        const unsigned long   MAX_ENTRIES = rootEntries*16;
        struct file_entry *top  = (struct file_entry *)hd_sector;

        memset(hd_sector, 0, 512);
        // Write Top Entry
        strcpy( top->name, topName);
        top->nameLen     = strlen( topName);
        top->entryType   = ENTRY_DIRECTORY;
        top->firstSector = 1;
        top->lastSector  = top->firstSector + topEntries -1;
        top->size        = 0;
        top->special     = MAX_ENTRIES+top->firstSector; // data starts here
        HD_Write512();
        HD_StartWriteSector(0); // Top Sector

        printf("FS_FormatDrive: Writing Root Entry [%d max entries]\n",rootEntries);
        FS_CreateDirectory(rootName, rootEntries);

        return true;
}

bool FS_IncrementFile()
{
        if (curFileEntrySector == curDir.lastSector || curFileEntrySector >= curDir.firstSector + curDir.size-1 )
        {
                return false;
        }
        else
        {
                curFileEntrySector++;
                HD_StartReadSector(curFileEntrySector);
                HD_Read512();
                memcpy(&curFile, hd_sector, sizeof(curFile) );
                return true;
        }
}

bool FS_DecrementFile()
{
        if (curFileEntrySector == curDir.firstSector )
        {
                return false;
        }
        else
        {
                curFileEntrySector--;
                HD_StartReadSector(curFileEntrySector);
                HD_Read512();
                memcpy(&curFile, hd_sector, sizeof(curFile) );
                return true;
        }
}

// Downloads a file from the mp3 player to the PC
bool FS_GetFile(const char *mp3Filename, const char *pcFilename)
{
        bool result;
        FILE *outFile;
```

```
        unsigned long i;
        unsigned long size,block;
        FS_RootDirectory();
        result=false;

        for (i=0; i< curDir.size && !result; i++, FS_IncrementFile() )
        {

                if (!strcmp(mp3Filename, curFile.name) )// if found
                {
                        outFile=fopen(pcFilename,"wb");
                        if (!outFile)
                        {
                                printf("Could not open [%s]\n",pcFilename);
                                return false;
                        }

                        size=curFile.size;

                        if (size == 0 )
                        {
                                printf("GetFile: Size is 0, abort!\n");
                                return false;
                        }

                        block=FS_StartReadFirstDataSector();
                        FS_FinishReadSector();

                        while (size!=0)
                        {
                                size-=block;
                                fwrite( hd_sector, block,1, outFile);
                                block=FS_StartReadNextDataSector();
                                FS_FinishReadSector();
                        }

                        fclose(outFile);
                        result=true;
                }
        }

        if (!result)
        {
                printf("Mp3 File [%s] was not found!\n",mp3Filename);
        }
        return result;
}

// Adds a file to the root directory....
bool FS_AddFile(const char *pcFilename, const char *mp3Filename)
{
        FILE *inpFile;
        unsigned long readLen,  totalBytes, totalSectors, dataSector, entrySector;
        struct file_entry *f = (struct file_entry *)hd_sector;

        FS_RootDirectory();
        if ( curDir.lastSector - curDir.firstSector + 1 == curDir.size )
        {
                printf("FS_AddFile: The Root Directory is full!\n");
                return false;
        }
```

```
if ( strlen(mp3Filename) >=NAME_MAX )
{
        printf("FS_AddFile: [%s] is has length greater than %d!\n",mp3Filename, NAME_MAX );
        return false;
}

// Find first free entry
if (curDir.size == 0)
{
        entrySector = curDir.firstSector;
}
else
{
        while ( FS_IncrementFile() );
        entrySector = curFileEntrySector+1;
}

// Find First Free sector for file data.......
dataSector = FS_FindFreeDataSector();
if (!dataSector)
{
        FS_TopDirectory();
        dataSector = curDir.special;
        FS_RootDirectory();
}
        totalBytes=totalSectors=0;

inpFile=fopen(pcFilename,"rb");
if (!inpFile)
{
        printf("FS_AddFile: Could not open input file [%s]\n",pcFilename);
        return false;
}

readLen = fread( hd_sector, 1,512, inpFile ); // read one sector

if (readLen == 0 || ferror(inpFile) )
{
        printf("FS_AddFile: Error while reading file!\n");
        return false;
}

while (readLen>0)
{
        HD_Write512();
        HD_StartWriteSector(dataSector);
        dataSector++;
        totalSectors++;
        totalBytes+=readLen;
        readLen = fread( hd_sector, 1,512, inpFile ); // read one sector
}
fclose(inpFile);

// Now write file entry
memset(hd_sector,0,512);
strcpy(f->name, mp3Filename);
f->nameLen      = strlen(mp3Filename);
f->entryType    = ENTRY_FILE;
f->firstSector  = dataSector - totalSectors;
f->lastSector   = dataSector -1;
```

```
        f->size         = totalBytes;


        HD_Write512();
        HD_StartWriteSector( entrySector);

        // Update table for parent directory....
        HD_StartReadSector(curDirSector);
        HD_Read512();
        f->size++;      // Increment Count
        curDir = *f;
        HD_Write512();
        HD_StartWriteSector(curDirSector);

        return true;
}


void FS_PrintFile()
{
        printf("FILE: Entry @ sector = %ld\t",curFileEntrySector);

        switch( curFile.entryType )
        {
                case ENTRY_FILE:        printf("FILE_ENTRY"); break;
                case ENTRY_DIRECTORY:   printf("DIR_ENTRY "); break;
                default :               printf("???????????");
        }
        printf("\t");
        printf("[%s\\%s]  (%4ld)->(%4ld) [%4ld], bytes: %7ld\n",
curDir.name,curFile.name, curFile.firstSector, curFile.lastSector, curFile.lastSector - curFile.firstSector +1, curFile.size);
}


unsigned long FS_StartReadFirstDataSector()
{
        if (curFile.entryType != ENTRY_FILE )
        {
                printf("FS_ReadFirstDataSector: Error, This is not a file!\n");
                return 0;
        }

        curFileSector = curFile.firstSector;
        curFileBytesLeft = curFile.size;

        if (curFile.size == 0 )
        {
                printf("FS_ReadFirstDataSector: Warning: File is zero size!\n");
                return 0;
        }

        HD_StartReadSector(curFileSector);

        if (curFileBytesLeft > 512)
        {
                curFileBytesLeft -= 512;
                return 512;
        }
        else
        {
                unsigned long x;
```

```
                x=curFileBytesLeft;
                curFileBytesLeft = 0;
                return x;
        }
}
unsigned long FS_StartReadNextDataSector()
{
        if (curFileBytesLeft == 0 )
        {
                return 0;
        }
        else // more data left
        {
                curFileSector++;        // move to next sector
                HD_StartReadSector(curFileSector);

                if (curFileBytesLeft > 512)
                {
                        curFileBytesLeft-=512;
                        return 512;
                }
                else
                {
                        unsigned long x;
                        x= curFileBytesLeft;
                        curFileBytesLeft=0;
                        return x;
                }
        }
}

void FS_FinishReadSector()
{
        HD_Read512();
}

bool FS_CurDirIsEmpty()
{
        return curDir.size == 0;
}

char  *FS_CurDirName()
{
        return curDir.name;
}
unsigned char   FS_CurDirNameLength()
{
        return curDir.nameLen;
}

unsigned short  FS_CurDirNumEntries()
{
        return curDir.size;
}

char  *FS_CurFileName()
{
        return curFile.name;
}

unsigned char   FS_CurFileNameLength()
```

```
{
        return curFile.nameLen;
}

unsigned long   FS_CurFileSize()
{
        return curFile.size;
}

unsigned char  *FS_BufferPointer()
{
        return hd_sector;
}

unsigned short  FS_BufferSize()
{
        return sizeof(hd_sector); // 512
}
```

References

[1] Microsoft Corporation. Microsoft extensible firmware initiative: Fat32 file system specification. 2000.