Allan Evans
Dept. of Electrical and Computer Engineering

Introduction

A sixteen bit file allocation table (FAT16) is a common method of allocating a memory device to hold information in an organized manner. It is one of the ways many consumer devices such as digital cameras, PDAs, and portable music devices organize their files. FAT16's ease of use and low overhead makes it an ideal solution for many applications.

This application note describes the implementation of a FAT16 interface for the Texas Instruments MSP430, a low-power 16-bit microcontroller. This interface, combined with the MSP430 and some basic memory device functions, can form the foundation for any device that needs to record and read information.

FAT16 Standard

The FAT16 standard is a standard designed by Microsoft. The original standard used 12 bytes instead of sixteen, and was created for organizing hard drives. Later, FAT16 was developed for larger hard drives because of limitations in the 12 byte standard. Today, FAT32 also exists and these three standards do not work with each other. Microsoft continues to maintain these standards. FAT16 now also commonly found in portable memory devices such as camera's that save information so it can be read in Windows with no additional software.

Comparison with other technologies

Fat16 can work with devices up to 2 GB in size, while being easy to implement if the 8.3 standard is used. FAT12 does not support drives that are found in current technologies, and FAT32 has a more difficult implementation. Fat16 allows functionality with any memory device currently on the market while minimizing the complexity of the interface.

The other standard beside FAT is NTFS. NTFS is a much more difficult to implement. FAT16 and all FAT file systems are simplistic, and thus they are not secure. If security is a primary concern, NTFS is a much better option. FAT16 is very easy to implement, but to maintain its simplicity, it sacrifices features like security.

Overview of a FAT16 File System

Table 1 contains a relative layout of a FAT16 file system with only one partition as it appears in memory.

Memory Device
Master Boot Record
FAT16 Boot Record
FAT Tables
Directory Table
DATA

Table 1: Overview of a simple FAT on a memory device

From table 1, it can be seen that the locations of everything absolutely rely on the information in the previous block. Only the relative location of information can be ascertained without getting information off of the device. Each FAT has different protocols governing what information these various sections contain, and where the information is located in the section.

Table 1 is the generic structure of a FAT with a single partition. Windows XP leaves off the Master Boot Record when it formats some of the portable memory devices available on the market. The general structure still applies; it just starts with the FAT Boot Record instead of the Master Boot Record.

FAT16 protocol dictates the rest, and various variables and the location of files etc. are all defined in the various tables. What follows is a step by step look at each of the major components.

Master Boot Record

The Master Boot Record (MBR) is always located at the very beginning of the memory; sector 0. It is the first set of code that the computer will read, and it does not contain very much information for a slave type storage device. The main purpose of the master boot record is to boot an operation system from it. This is why new formatting of external memory eliminates it altogether. The MBR is 512 bytes long (one sector on most memory devices), and it contains the partition table. Table 2 is the breakdown of the MBR and the hexadecimal offsets of each component.

Offset	Description	Size
000h	Executable Code (Boots Computer)	446 Bytes
1BEh	1st Partition Entry	16 Bytes
1CEh	2nd Partition Entry	16 Bytes
1DEh	3rd Partition Entry	16 Bytes
1EEh	4th Partition Entry	16 Bytes
1FEh	Executable Marker (55h AAh)	2 Bytes

Table 2: Layout of a Master Boot Record

The only entries that matter in developing a non-bootable memory interface are the partition tables. In almost all such cases, using one partition is the best solution for good memory management. Table 2 can be used to find the information in the MBR that is needed to interface with Windows. All that matters is the 1st Partition entry at 1BEh. The information in the entry can be found in table 3.

Offset	Description	Size
00h	Current State of the Partition	1 Byte
01h	Beginning of the Partition - Head	1 Byte
02h	Beginning of the Partition – Cylinder/Sector	2 Bytes
04h	Type of Partition	1 Bytes
05h	End of Partition - Head	1 Bytes
06h	End of the Partition – Cylinder/Sector	2 Bytes
08h	# of Sectors between MBR and Partition	4 Bytes
0Ch	# of Sectors in the Partition	4 Bytes

Table 3: Layout of a partition entry in the MBR

The only information needed from the entire Master Boot Record is the location of the FAT16 Boot Record at the beginning of the first partition. Non-Disk drives do not have cylinders and heads. They are composed entirely of sectors. The only entry table 3 that contains the information needed to find the beginning of the partition is the Beginning of the Partition based on the sector.

FAT16 Boot Record

The Fat16 Boot Record is the information located at the beginning of every partition. In the case of a windows XP formatted card, it happens to be located in sector 0. In cards formatted with an MBR, the MBR specifies where the partition begins and the FAT16 Boot record is the first sector (512 Bytes) of the partition. Table 4 diagrams its contents with the appropriate offsets.

Offset	Description	Size
00h	Jump Code and NOP	1 Byte
03h	OEM Name	8 Bytes
0Bh	Bytes per sector	2 Bytes
0Dh	Sectors per cluster	1 Bytes
0Eh	Reserved Sectors	2 Bytes
10h	# of copies of the FAT	1 Bytes
11h	Maximum root directory entries	2 Bytes
13h	# of Sectors in a partition smaller then 32MB	2 Bytes
15h	Media descriptor	1 Bytes
16h	Sectors per FAT	2 Bytes
18h	Sectors per track	2 Bytes
1Ah	# of heads	2 Bytes
1Ch	# of hidden sectors in the partition	4 Bytes
20h	# of sectors in the partition	4 Bytes
24h	Logical drive number of the partition	2 Bytes
26h	Extended signature (29h)	1 Bytes
27h	Serial Number of Partition	4 Bytes
2Bh	Volume name of the partition	11 Bytes
36h	FAT Name (FAT 16)	8 Bytes
3eh	Executable Codes	448 Byte
1FE	Executable Marker (55h AAh)	2 Bytes

Table 4: Layout of the FAT16 Boot Record

Much of the information in the FAT boot record is important in the handling of the rest of the information. The important entries involve size determination and location of the rest of the information. Anything implementing a FAT16 just needs to remove this information from the FAT16 Boot Record before finding the location of the Directory Table.

Directory Table

It is slightly odd that to implement a successful FAT16 you read information from the Boot Record and then skip the FAT tables and go straight to the directory table. The location of the directory table is the start of the partition + the number of reserved sectors + number of Sectors per FAT * the number of FATs. This is very easy to implement in software as long as the necessary variables were taken from the MBR and the FAT16 Boot Record and stored in RAM. The directory table contains all the file entries in the order they are entered. Each entry is 32 bytes long, and the directory table can contain 512 entries. The structure of the data file entries can be found in table 5.

Offset	Description	Size
00h	Name of the File	8 Bytes
08h	Extension of the File	3 Bytes
0Bh	Attribute	1 Bytes
16h	Time	2 Bytes
18h	Date	2 Bytes
1Ah	Start Cluster	2 Bytes
1Ch	File Size	4 Bytes

Table 5: File entry structure in the directory table

Windows does not require the attribute, time, or date of the file to properly read and use it. The rest of the entries need to be filled when the file is written and/or created. Once the file being written to is determined, a search through file names finds a match. The start cluster can then be found, and then the FAT tables need to be accessed to determine what cluster we need to read/write to.

FAT Table

The FAT table is the single most important element of the data management system. It contains the location of all of the parts of every file and how they are connected. There are almost always multiple copies of these tables because if they are lost, the data is very difficult to recover. To find the location of a particular cluster of a particular file, a single table can be used. When editing files, every FAT needs to be updated or Windows will have problems reading back the data.

The Fat table acts like a linked list of the sectors of a file. For example; File A starts in cluster 3, then the data continues in 5, 9, and ends in cluster 15. The directory table would have a start cluster for File A at 3. Then the entry in cluster three in the FAT table would have a 5 in it. This means that the file continues in 5. If you went to the entry for cluster 5, it would have a 9 in it. This continues until the last cluster in the chain which contains an FFh FFh. The FFh FFh signifies the end of a file.

When writing new information to a FAT it is important to know two things. One, data starts with cluster number two. The first two clusters are used for the FAT16 Boot Record and for the Directory table. Also, cluster two begins immediately after the Directory table, and every cluster follows in order after this. Two, 00h 00h is the entry that signifies an open cluster, and all cluster entries in the FAT table are 2 bytes long. When looking for the next open cluster, the FAT needs to be searched for the 00h 00h entry.

Implementation

The FAT16 interface is implemented as simply as possible. It supports only one directory (the root), only one partition, and assumes the memory device is as defragmented as possible. Due to memory (RAM) limitations with the MSP430, a quick FAT16 that has complete functionality is very difficult to implement. The interface is optimized for write speed, and for devices that are providing data through windows. It sacrifices the ability to optimize memory utilization in order to maximize speed.

Required Functions

The following FAT16 interface assumes the user has written the underlying function to communicate with the specific memory device. This implementation also assumes that the memory device is divided into 512 bytes sectors. The *sd_read_block()* command reads the data from a specified sector on the memory device. *sd_write_block()* writes a 512 byte buffer to the sector specified. With these two functions, the entire FAT16 interface can be created.

FAT Initialization

The initialization is not as complex as it could be. FAT16 is assumed so various standards do not have to sorted out and compensated for. The primary goal is to verify that FAT16 is being used. Determine the existence (or lack thereof) of a Master Boot Record, and load all the values needed to perform the rest of the FAT functionality.

The initialization begins by reading the very first sector on the memory device (sector 0). The OEM name “FAT 16” is looked for in its correct memory location. If it is not found, the first sector contains the Master Boot Record. If it is found then the first sector is the FAT16 Boot Record. If the first sector is the master boot record, then the 1st partition table entry is accessed and the start location is read. Otherwise, the start location for the partition is set as zero.

The initialization process then reads the FAT16 Boot Record at the head of the first partition.
The function will load most of the necessary values into RAM so later functions will have access to them. The sectors per cluster, the number of FAT tables, the sectors per FAT, first FAT location, and the number of reserved sectors are all taken. Then, the function uses some of the values to determine the start locations of every FAT table. Lastly, the beginning of the directory table is calculated and stored into memory.

The function is implemented by *init_fat()* and is specific to FAT16 devices. It loads the FAT16 Boot Record, and determines and saves all the necessary variables for correct FAT implementation.

Open or Create File

The open file command takes a file name. It will then either open the file or create it and place it into the directory structure. Opening a file is much different here then it is in windows. What opening means is initializing a particular file for read or write purposes . In the process, a memory structure with the information for the file will be created until the file is closed. Also, only one file can be opened at a time under the current implementation.

The first sector of the directory table is read. The function then searches through this sector of the root directory for the file. If the file has not been found yet, the next sector of the root directory is loaded. The function will search the entire root directory in this manner until the file is either found, or needs to be created.

If a name match occurs, some information from the file is loaded into memory. The file number, which is the files location in the directory (e.g. the 5th file would have file number 4), the start cluster, and the file size are taken. Then, the start sector (on the memory device itself) is determined and committed to memory.

If the file was not found, the function creates it. It looks for the first open entry in the root directory, and creates an entry with the name given and the file number of its location. The function sets a new file flag and the size as 0. The start cluster and start sector will be set later.

Next, the function loads the very last sector of the first FAT table. The FAT is then cycled through from back to the front until a cluster with information is found. The cluster is noted, and the cluster immediately after that is set as the next open cluster. The open file does this to find the last filled cluster so any write will not be fragmented, and nothing will be overwritten. If the file is new, this cluster is set as the start cluster of the file and an FFh FFh is put into the FAT's. This means the file has no information and starts and ends at the write cluster.

Lastly, the function determines the sector where writing will begin on the memory device based on a formula using the write cluster. This information is stored in RAM so we can update the FAT tables after anything is written. This function is run with the *open_file()* command.

Write File

The write file takes a 512 byte buffer of data and the information for the file currently opened and writes the data to the memory device. Real time write speed is essential for applications that need to write more information then RAM can hold. This is particularly important when taking information from the ADC, or through other inputs, and storing it in its raw form. For this purpose the *fast_write_file()* was optimized.

The information passed into the function is just the information needed to be written. The function then takes the data write location, and writes the information. It then increments the write location and sets the ending sector for when we update the FAT. It also adds 512 to the size of the file. This function is designed to be looped any number of times without having to re-initialize anything, and without having to write or receive any information that is not data.

The function will overwrite anything on the card in the sector it is writing and that is why we looked for the last used cluster when we open the file. That way, nothing can be overwritten, and the linear write works perfectly. The other drawback to this form of write occurs on a fragmented memory device. When fragmentation places information deep into the memory area, the only open space for this write is after that last piece of data. This can cause horrible memory efficiency and so it is recommended that a newly formatted card be used.

Read File

The read file takes a sector number of the file to read. Then, the start cluster in the FAT table is read, and the list is followed through until the cluster in which the read will occur is found. The sector that starts the cluster is loaded, and either incremented until the sector is reached or not (depending on the specific sector and the number of sectors per cluster).

A buffer is then loaded with the information in that particular sector. This allows the MSP430 to use a configuration file. The file can be of any format, but a .txt file is very easy to implement in this manner. Using a simple parser, any user configuration can be done in Notepad (or anything else) and then used to control the functioning of the device.

Close File

The close file function needs to update everything that was modified and could not be updated because of speed concerns. The function is called *close_file()* and it uses the information in the file and fat memory structures to update the FAT and the directory tables. If this function is not called, the data will not be accessible by a computer running Windows.

The function goes to the start cluster of the file and then follows it through the FAT along the linked list until it reaches the cluster where writing began. Then, the FAT tables are filled linearly (e.g. 7 to 8 to 9 to 10 etc.) Until we reach the ending cluster of our write. The last cluster in the sequence is filled with FFh FFh signifying the end of the file.

The close file also updates the size of the file. This is very important because windows will be unable to access data if the size information is not correct.

Updates and Changes

Note that the following are additions not yet implemented. These functions and enhancements to current functionality may or may not be completed. Once finished, this document will be updated to reflect the better FAT16 implementation.

The major issue with the current implementation is the reliance on all aspects of the functionality to adhere to a defragmented memory device. The opening and closing of files are both designed with the linear fast write in mind. This hampers the addition of better write functions. Due to write speed concerns, these issues could not be circumvented.

Much more functionality is supported by FAT16. Likely future functions would allow the creation of directories, overwriting sectors of functions, writing optimally for memory instead of speed, and deleting files. Current functions would need to be altered to support these new functions, but that should take very little. The last concern would be to implement more rigorous error checking.

Summary

FAT16 is a very good file system for small memory formats. It is ideal for microprocessor based devices that need a format to provide data directly to the user via Windows. It is better then FAT12 for device size allowance, and easier to implement then FAT32.

The FAT16 interface here is ideal for writing in real time to a memory device. It has some problems as an active memory manager, but digital cameras, recorders, and sensors are ideally suited to its speed. Further functionality can be developed to fit device specifics with little change to what exists. This interface is best suited with SD, MMC, or USB memory interfaces.

Appendix A

```
#include  <msp430x16x.h>
#include "main.h"
#include "sd.h"
#include "fat.h"
#include "util.h"

// initialize the card function... probably needs a lower level call?
int init_fat( sd_context_t *sdc, FAT16_t *FAT, u8 *sd_buffer)
{
   int i;

   // Hard-coded for now.
   FAT->Bytes_Sector = 512;

   sd_read_block (sdc, 0, sd_buffer);   // read in the very first cluster
//    sd_wait_notbusy (sdc);

   if (bytecmp (&sd_buffer[54], "FAT", 3) == 0)
   {
      FAT->P_Start = 0;
   }
   else
   {
      // find the start of the first partition w/ these 4 enties
      // set the start of the partition to the FAT info
      FAT->P_Start = from_LE_32(*((u32 *)(&sd_buffer[454])));
   }

   if (sd_read_block(sdc, FAT->P_Start, sd_buffer) == 0)  // read in the partition block
information
      return 0;

//    sd_wait_notbusy (sdc);

   FAT->Secs_Cluster = sd_buffer[13];                  // set the number of sectors per cluster
   FAT->Num_Fats = sd_buffer[16];                      // record the number of Fats
   FAT->Secs_Fat = from_LE_16(*((u16 *)(&sd_buffer[22]))); // sets the Sectors per FAT
   FAT->Res_Secs = from_LE_16(*((u16 *)(&sd_buffer[14]))); // Sets the number of reserved
Sectors

   FAT->Fat_Start[0] = FAT->P_Start + FAT->Res_Secs;              // Set first FAT table location
   for (i=1; i<FAT->Num_Fats && i < MAX_FATS; i++)               // loop to set all Fat start points
   {
      FAT->Fat_Start[i] = FAT->P_Start + FAT->Secs_Fat * i + FAT->Res_Secs;
```

```
    }

    FAT->Root_Start =
        FAT->P_Start + FAT->Res_Secs +  FAT->Secs_Fat * FAT->Num_Fats;

    return 1;
}



/*

Funtion: open_file

Purpose:
  Opens or creates a file in the root directory on the card.  Loads all necesarry
  information into a FILE struct so we can quickly use it to optimize our write and
  read functions.
*/

int open_file( sd_context_t *sdc, FILE_t *FILE, FAT16_t *FAT, u8 *sd_buffer, u8 *filename)
{
//    unsigned char file_name[11];
    long int i = 0;
    int j = 0;
    int k = 0;
    int temp = 0;
    u8 quit = 0;
    u8 newfile = 0;

    //converts the input filename into array format  loop could be backward********

    for (i=0; i<11; i++)
        FILE->File_Name[i] = ' ';

    i = 0;
    do
    {
        FILE->File_Name[i] = filename[i];
        i++;
    }
    while (filename[i] != '.' && i < 8);

    i++;   //increment i to get past the period

    //set the extension
```

```
        FILE->File_Name[8] = filename[i];
        FILE->File_Name[9] = filename[i + 1];
        FILE->File_Name[10] = filename[i + 2];


        sd_read_block (sdc, FAT->Root_Start, sd_buffer);   // read in the root directory
//      sd_wait_notbusy (sdc);

//      bytecpy (FILE->File_Name, file_name, 11);

        i = 0;
        //loop through all files to find if the file already exists
        do
        {
            temp = i * 32;

            //do this to see if we overan a cluster and need to increment to continue
            if(temp == 512)
            {
                sd_read_block (sdc, FAT->Root_Start + j, sd_buffer);
//              sd_wait_notbusy (sdc);
                j++;
                i = 0;
            }

            //check to see if we have a name match
            if (bytecmp (FILE->File_Name, &sd_buffer[temp], 11) == 0)
            {
                /* We have a match, break loop and assign current file attributes */
                FILE->file_number = i + 16 * j;     //state what file # it is (in the root directory)

                // figure out the start cluster of the file so we can mess with the FAT
                FILE->start_cluster = from_LE_16(*((u16 *)(&sd_buffer[temp + 26])));

                // set the starting sector for the file so we dont have to do the math later
                FILE->start_sector = (FILE->start_cluster - 2) * FAT->Secs_Cluster + FAT->Root_Start + 32;

                //load the current file size into this noise!
                FILE->file_size = from_LE_32(*((u32 *)(&sd_buffer[temp + 28])));


                //create a break so we can end this loop
                quit = 1;
//              newfile = 1;
            }
            i++;
        }
        while (sd_buffer[temp] != 0x00 && sd_buffer[temp] != 0xeb && quit == 0);

        //set file number if it is a new file
        if(quit == 0)
        {
            newfile = 1;
            FILE->file_number = (i-1) + 16 * j;     //state what file # it is (in the root directory)
            FILE->file_size = 0;
        }


        // Loop through to find the last used cluster and then increment 1
        //set the conditions for the loop
        quit = 0;
        i = FAT->Secs_Fat - 1;

        do
        {
            sd_read_block (sdc, FAT->Fat_Start[0] + i, sd_buffer); //read in the last block of fat- go
    backward
    //        sd_wait_notbusy (sdc);

              for(temp = 511; temp >= 0 ; temp -=2)            // cycle through all 2 byte combos
              {
                  if((sd_buffer[temp] != 0 || sd_buffer[temp-1] != 0) && quit == 0)  // check to see if its
    full
                  {
                    FILE->Fat_OpenCluster = (i*256) + (temp + 1)/2;       //set the open cluster for writing
                    quit = 1;                                   // prepare to quite


                        // if the file is a new file, this writes FF to the start sector for the file,
                        // so it exists.... and the Fat update in the close file will work!

                        if( newfile == 1)
                        {
                            for(k=0; k < FAT->Num_Fats; k++)
                            {
                                *((u16 *)(&(sd_buffer[temp + 1]))) = to_LE_16 (0xFFFF);
                                sd_write_block(sdc, FAT->Fat_Start[k] + i, sd_buffer);
                            }
                        }
                  }
              }
              i--;
        }
        while (i >= 0 && quit == 0);                    // conditions to continue or break


        // set the first sector of the new file
        if(newfile == 1)
        {
              // figure out the start cluster of the file from the fat with the FAT
              FILE->start_cluster = FILE->Fat_OpenCluster;

              temp = FILE->file_number / 16;

              sd_read_block(sdc, FAT->Root_Start + temp, sd_buffer);
    //          sd_wait_notbusy(sdc);

              bytecpy (&sd_buffer[(FILE->file_number - (temp* 16))*32], FILE->File_Name, 11);

              *((u16 *)(&sd_buffer[(FILE->file_number - (temp* 16))*32 + 26])) =
                    to_LE_16(FILE->start_cluster);

              sd_write_block(sdc, FAT->Root_Start + temp, sd_buffer);
              // set the starting sector for the file so we dont have to do the math later
              FILE->start_sector = (FILE->start_cluster - 2) * FAT->Secs_Cluster + FAT->Root_Start + 32;
        }

        FILE->Fat_OpenSec = (FILE->Fat_OpenCluster - 2) * FAT->Secs_Cluster + FAT->Root_Start +
    32;
        FILE->end_sector = FILE->Fat_OpenSec;

         return 1;
    }


    /*

    Funtion: fast_write_file

    Purpose:
      Designed to write to a file linearly as fast as possible,  This should be faster for a write then
      the ADC is to convert so we can write information to the card faster then we can read it in.
      This assumes only one file is open... all function currently assume that!
    */

    int fast_write_file( sd_context_t *sdc, FILE_t *FILE, u8 *sd_buffer)
    {
        //write the information to the cluster
        sd_write_block_async(sdc, FILE->end_sector, sd_buffer);

        //increment the end cluster by one
        FILE->end_sector++;
        FILE->file_size += 512;

        return 0;
    }

    /*

    Funtion: close_file

    Purpose:
      Designed to close out a file and update all of the FAT tables and the partition table
    */

    int close_file( sd_context_t *sdc, FILE_t *FILE, FAT16_t *FAT, u8 *sd_buffer)
    {


        int i = 0;
        int j;
        u32 next_cluster = FILE->start_cluster;
        u32 next_cluster_temp = FILE->start_cluster;
        u32 end_cluster = (FILE->end_sector - 32 - FAT->Root_Start)/(FAT->Secs_Cluster) + 2;
        u32 current_sector = 0;
        u32 current_pos = 0;
        u32 num_clusters;
        int temp = next_cluster / 256;  // should round down right? *******

        //open the FAT and go to the first cluster of the file.... then follow it until we get to the write
    start

        sd_read_block (sdc, FAT->Fat_Start[0] + temp, sd_buffer);
    //   sd_wait_notbusy (sdc);

        while(next_cluster != FILE->Fat_OpenCluster)
        {
            temp = next_cluster / 256;  // should round down right? *******
            next_cluster_temp = next_cluster;

            if (temp * 256 == next_cluster)
            {
              sd_read_block (sdc, FAT->Fat_Start[0] + temp, sd_buffer);
```

```
//          sd_wait_notbusy (sdc);
        }

        next_cluster =
           from_LE_16(*((u16 *)(&sd_buffer[ (next_cluster_temp * 2)-( 512 * temp )])));

    }

   //once we get to the write start, we need to fill in the FAT tables with everything until the
write end
   num_clusters = end_cluster - FILE->Fat_OpenCluster;
   current_sector = FAT->Fat_Start[0] + temp;
   current_pos =  (next_cluster_temp * 2) - ( 512 * temp );

   for(i = 0 ; i <= num_clusters ; i++)          // cycle through enties until done
   {

      next_cluster++;

      if(i != num_clusters)
      {
         // set the cluster to point to the next cluster
         *((u16 *)(&(sd_buffer[current_pos]))) = to_LE_16 (next_cluster);
      }
      else
      {
         // set the last cluster to FF

         *((u16 *)(&(sd_buffer[current_pos]))) = to_LE_16 (0xFFFF);

         //write to all FATs
         for(j = 0; j < FAT->Num_Fats; j++)
         {
            sd_write_block(sdc, current_sector + FAT->Secs_Fat * j, sd_buffer);
         }
      }


      //increment the position and check to see if we need to load the next buffer
      current_pos += 2;

      if(current_pos == 512)
      {
         //write to all FATs
         for(j = 0; j < FAT->Num_Fats; j++)
         {
                sd_write_block(sdc, current_sector + FAT->Secs_Fat * j, sd_buffer);
            }

            current_pos = 0;
            current_sector++;

            sd_read_block(sdc, current_sector, sd_buffer);
//            sd_wait_notbusy (sdc);
        }

    }

//updates to the partition table

//read in the root directory block where the file is
temp = (FILE->file_number/16);
sd_read_block (sdc, FAT->Root_Start + temp, sd_buffer);
//              sd_wait_notbusy(sdc);

//set the size of the file
*((u32 *)(&(sd_buffer[(FILE->file_number - temp * 16)*32 + 28]))) = to_LE_32 (FILE->file_size);

//write it back to the root table
sd_write_block(sdc, FAT->Root_Start + temp, sd_buffer);

return 1;
}

/*

Funtion: read_file

Purpose:
  Designed to read a single 512 location on a file....
*/

int read_file( sd_context_t *sdc, FILE_t *FILE, FAT16_t *FAT, u8 *sd_buffer , u16 read_start)
{

    int j;
    u16  read_cluster = read_start / FAT->Secs_Cluster;
    u32 next_cluster = FILE->start_cluster;
    u32 next_cluster_temp = FILE->start_cluster;

    int temp = next_cluster / 256;  // should round down right? *******
```

```
   //open the FAT and go to the first cluster of the file.... then follow it until we get to the read
start

   sd_read_block (sdc, FAT->Fat_Start[0] + temp, sd_buffer);
//   sd_wait_notbusy (sdc);

   for(j= 0; j < read_cluster; j++)
   {
      temp = next_cluster / 256;  // should round down right? *******
      next_cluster_temp = next_cluster;

      if (temp * 256 == next_cluster)
      {
         sd_read_block (sdc, FAT->Fat_Start[0] + temp, sd_buffer);
//          sd_wait_notbusy (sdc);
      }

      next_cluster =
         from_LE_16(*((u16 *)(&sd_buffer[ (next_cluster_temp * 2)-( 512 * temp )])));

   }


   u16 read_sector = (next_cluster-2) * FAT->Secs_Cluster + FAT->Root_Start + 32 + (read_start -
read_cluster*2);
   sd_read_block (sdc, read_sector, sd_buffer);
//   sd_wait_notbusy (sdc);


return 1;
}
```

Appendix B

```
#ifndef FAT_H
#define FAT_H

#include "sd.h"

#define MAX_FATS 4


// This is the struct for the file system.  We should use one more for the actual file itself.
typedef struct
{
   u8 Num_Fats;           // Total Number of Fats
   u32 Fat_Start[MAX_FATS];        // Start Location of each fat here
   u32 P_Start;             // this is the start of the partition block
   u8 Secs_Cluster;        // This is the number of sectors per cluster
   int Bytes_Sector;  // This is the number of bytes per cluster (defaut 512)
   u16 Secs_Fat;            // The number of sectors per fat Table
   u32 Root_Start;           // This is the start of the root directory
   u16 Res_Secs;            // The number of reserved sectors
} FAT16_t;

typedef struct
{
   unsigned char File_Name[11];        // Name of the File
   u16 start_cluster;         // Cluster in which the file begins
   u32 start_sector;          // Sector on SD card where the file begins
   u32 Fat_OpenCluster;        // cluster on SD card where the write begins
   u8 file_number;            // This is the position in the RD of the file (starts at 0)
   u32 Fat_OpenSec;           // This is one greater then the last occupied sector (for fast write)
   u16 end_sector;          // sector in which write ends
   u32 file_size;          // This is one greater then the last occupied sector (for fast write)

} FILE_t;

int init_fat( sd_context_t *sdc, FAT16_t *FAT, u8 *sd_buffer);
int open_file( sd_context_t *sdc, FILE_t *FILE, FAT16_t *FAT, u8 *sd_buffer, u8 *filename);
int fast_write_file( sd_context_t *sdc, FILE_t *FILE, u8 *sd_buffer);
int close_file( sd_context_t *sdc, FILE_t *FILE, FAT16_t *FAT, u8 *sd_buffer);
int read_file( sd_context_t *sdc, FILE_t *FILE, FAT16_t *FAT, u8 *sd_buffer, u16 read_start);
#endif
```

References

[1] Dobiash, Jack. FAT16 Structure Information, http://home.teleport.com/~brainy/fat16.htm

[2] Foust, F. Secure Digital Interface for the MSP430
http://www.egr.msu.edu/classes/ece480/goodman/fall/group05/deliverables/index.html