CpE5160 Experiment #5
Creating a File System

Introduction

The purpose of this experiment is to create a file system that will allow the microcontroller to find and read an MP3 file from the SD card. The file system will need to locate the Master Boot Record and then the BIOS Parameter Block and read the information stored there. This information will then be used to determine the FAT type (FAT16 or FAT32), the starting sector of the FAT (File Allocation Table), the starting sector of the root directory and the starting sector of the data area. The root directory entries can then be printed using the serial port. A simple user interface is given that will allow the user to select a directory entry. If this entry is a directory, then the entries in that directory can then be printed. If the entry is a file, then the first sector can be displayed. Some simple controls will be needed to display more of the file or to return to the directory.

There are some file systems made for embedded systems available on the internet. A search for Tiny FAT, Tiny File System, Petite FAT or Petite File System will yield several choices. I have briefly looked at the Petite File System, but I have not tried to implement it. These file systems may or may not work with our system, but they may give some useful insight on how to complete this experiment. If you try to implement one of these file systems, it is important to keep in mind that we are trying to implement a real-time operating system and that eventually the file will need to be read in blocks as one of the tasks in the system.

Reading a Sector from the SD Card

A function will be needed to interface the file system functions with the SD Card functions from the second experiment. This function will read the sector whose number is passed to it as a parameter. This function should be placed in its own file so that it can be changed without altering the file system. Anytime a file system function needs to read a sector from the disk, it will call this function. This will give the file system more portability because this function can be changed to access other types of disks and the file system functions will not need to be changed.

Reading Values from a Block

One of the most basic operations that will be needed for this experiment is the ability to read 8-, 16- and 32-bit values from an SD card block. The read sector function mentioned in the previous paragraph will read a sector from the disk and put the data into SRAM. In this experiment, some functions are needed that can read a specific value from the block of data stored in SRAM. The functions will have one parameter that specifies the starting SRAM address of the block of data and another parameter that gives the offset address of the value in that block.

Since some of the values have multiple bytes, it is important to note that these values are stored in little-endian format. This means that the least significant byte (LSB) will be stored at the offset address given as a parameter. Incrementing the address by one will

point to the next most significant byte. A 16-bit value is stored in two bytes and a 32-bit value is stored in four bytes. The values can be put together into a uint16_t or uint32_t value by shifting and OR'ing each successive value. You could also try typecasting the 8-bit pointer to the array of values into a pointer to a 16-bit or 32-bit array and then reading a 16-bit or 32-bit value from the array.

Locating the BIOS Parameter Block

Block 0 of the SD card will contain either the BIOS Parameter Block (BPB) or the Master Boot Record. The BPB can be identified by the first byte of the block which is 0xEB or 0xE9 (an x86 architecture jump instruction). Typically, SD cards are formatted like a hard disk drive (HDD) and usually block 0 contains the Master Boot Record. The most important value for this experiment is the relative sectors value stored at offset 0x01C6. This value is a 32-bit value, and it contains the sector number of the BPB.

An important difference in Standard Capacity and High-Capacity SD cards should be noted. In High-Capacity SD cards the block size is fixed at 512 bytes which corresponds with the typical sector size. The address required for the READ_SINGLE_BLOCK command (CMD17) is the block number which is usually the same as the sector number. Therefore, any calculated sector number can be used as the block number on the SD card.

In Standard Capacity SD cards, the block size can be adjusted from 1 byte to 1024 bytes with CMD16, and each byte of the SD card has its own address. The recommendation is to set the block size to 512 bytes to be compatible with a High-Capacity SD Card sector size. However, the address required for the READ_SINGLE_BLOCK command (CMD17) is the address of the first byte of the block. Therefore, a sector number must be multiplied by 512 (number of bytes in a sector) to get the correct address.

Reading Values from the BIOS Parameter Block

The following values will need to be read from the BPB.

FAT	Name	Offset	Size	Typical Value
16/32	BytesPerSector*	0x0B	16-bits	512
16/32	SectorsPerCluster*	0x0D	8-bits	64
16/32	RsvdSectorCount	0x0E	16-bits	Used to align Clusters
16/32	NumberFATs	0x10	8-bits	2
16	RootEntryCount	0x11	16-bits	0 for FAT32
16/32	TotalSectors16	0x13	16-bits	If 0, read TotalSectors32
16	FATsize16	0x16	16-bits	0 on FAT32
16/32	TotalSectors32	0x20	32-bits	Depends on disk size
32	FATSize32**	0x24	32-bits	Depends on disk size
32	RootCluster**	0x2C	32-bits	2

*These values need to be stored as global variables for use in more than one function.
** These values are found only in FAT32 BIOS Parameter Blocks

These values will be used to calculate the FAT type, the first sector of the FAT, the first sector of the root directory and the first data sector. The calculated values will need to be stored as global variables for use in more than one function. Some of the values may be

used to create alternate values to simplify some calculations. For example, the BytesPerSector value is used to divide in some calculations. Since this value is a power of two, the divide can be done with a right shift and the amount of the shift is determined from the BytesPerSector value.

File System Calculations

The first calculations that must be performed are used to determine the type of FAT. The series of calculations can be found in section 3.5 of the Microsoft FAT Specification that is on canvas. The FAT type value and the RootDirectorySectors value need to be stored as global variables for future calculations. The FAT type is used to determine how many bytes each FAT entry requires. This is used in the FATOffset calculations in section 4.1. This value is 2 for FAT16 or 4 for FAT32 which represents how many bytes each FAT entry requires. An alternative is to use 1 for FAT16 and 2 for FAT32 and left shift the cluster number by this amount. The RootDirectorySectors value is needed if FAT16 is supported to specify how many sectors are in the root directory.

The start of the FAT is determined with the ReservedSectorCount value. It is important to note that all the sector calculations in the Microsoft FAT Specification are relative to the BIOS Parameter Block (start of the volume). Therefore, the SD Card block number for the start of the FAT is found by adding the MBR_RelativeSectors value to the ReservedSectorCount value. The start sector of the FAT should be stored as a global variable since it will be needed when locating the next cluster of a file.

The first sector of the root directory for a FAT16 system is located immediately after the FAT. This sector is found by adding the FATsize multiplied by the number of FATs to the start sector of the FAT. This calculation can be found in Section 6.6 of the Microsoft FAT specification. The size of the area reserved for the Root Directory is given by the RootDirectorySectors value. The first sector of data follows immediately after the root directory area. This is shown with the calculation FirstDataSector in section 6.7 of the Microsoft FAT specification. Note that the MBR_RelativeSectors value should be added to this value to find the actual SD card block number. The starting sector of the root directory and first data sector are values that should be stored as global variables for use by other file system functions.

In FAT32, the RootDirectorySectors value is 0 and the root directory location is given by the RootCluster entry in the BIOS Parameter Block. This value is usually 2 unless the disk has a bad cluster(s). The FirstDataSector calculation is the same as for FAT16. Placing the RootCluster value into the equation for FirstSectorofCluster gives the starting sector of the root directory. The starting sector of the root directory and first data sector are values that should be stored as global variables for use by other file system functions.

The location of a file or sub-directory is given as the first cluster value stored in a directory entry. However, the read sector function needs a sector (block) number. Therefore, the FirstSectorofCluster calculation from section 6.7 of the Microsoft FAT specification is used to determine the starting sector of a cluster. It should be noted that the cluster value for the root directory stored in directory entries is 0. If this value is used

in the FirstSectorofCluster calculation, it will lead to problems. A check should be in place so that when first sector of a cluster is calculated and the cluster value is 0, then the sector value returned is the first sector of the root directory.

The File Allocation Table (FAT)

The file allocation table is a list of every cluster on the disk. Each entry in this table describes how a cluster is used on the disk. An entry for a FAT16 type disk is 16-bits and is 32-bits for a FAT32 type disk. FAT16 values should be typecast into 32-bit values for compatibility with the system. It should be noted that only 28 bits are used for the FAT entry on a FAT32 type disk. The most significant four bits are reserved and should be masked off by AND'ing with 0x0FFFFFFF. A table is given in section 4 of the Microsoft FAT Specification that describes what each entry means. An entry of all '0's indicates an unused cluster. An entry of a value from 2 to the maximum cluster count indicates an allocated cluster. The value points to the next cluster used in a file. An entry of all '1's indicates the last cluster of a file (end of file). There are also values for reserved clusters (0xFFF8 to 0xFFFE or 0xXFFFFFF8 to 0xXFFFFFFE) and defective clusters (0xFFF7 or 0xXFFFFFF7).

When reading a file, the current cluster number is used to locate which sectors contain the data of the file. The SectorPerCluster value indicates how many data sectors are in each cluster. The sectors will be loaded into SRAM one at a time in sequence. When the last sector is loaded, then the next cluster must be located. The current cluster number also corresponds to an entry in the FAT which contains the number of the next cluster in the file.

The procedure for finding the next cluster from the FAT is outlined in this paragraph with reference to the calculations given in section 4.1 of the Microsoft FAT specification. The first step is to determine the sector number where the corresponding FAT entry is located. The calculation ThisFATSecNum is used to find this value. The starting sector of the FAT calculated earlier should be substituted for the BPB_ResvdSecCnt value in the equation shown in section 4.1. The Start of FAT value includes the MBR_RelativeSectors value so that the resulting sector number can be used to read a block from the SD card. This sector is then read into SRAM using the read sector function. The second calculation is ThisFATEntOffset and it is used to determine the offset address within this sector where the FAT entry can be found. This value is read (either 16-bits or 32-bits depending in the FAT type), then modified (FAT16 typecast into a 32-bit value or FAT32 values AND'ed with 0x0FFFFFFF) as mentioned in an earlier paragraph and returned. The cluster number in this entry can then be used to load the next sector of the file into memory. This process should continue until the end of file value is found in the FAT entry.

Some microcontrollers can perform shift operations quicker than divide operations. So, an alternative method to the calculations given in Section 4.1 are:

ThisFATSecNum=StartofFAT+(N>>(BytsPerSecShift-FATOffsetShift))

ThisFATEntOffset=(N<<FATOffsetShift)&(BPB_BytsPerSec-1)

Reading the Root Directory

The root directory is the top-most directory on the disk. From this directory, the user may navigate to other directories or files whose entries are found in this directory. A directory is a list of 32-byte entries. The FAT16 root directory is limited to the number of entries given in the RootEntryCount value. The FAT32 root directory is contained in the first data sector and does not have a limit on the number of entries it can have. If the FAT32 root directory exceeds the contents of its original cluster, then another cluster is allocated.

A directory entry consists of a short file name entry and an optional long file name entry. The file attribute byte is located at offset 0x0B in a directory entry. A long file name entry does not contain any file information other than the name. The file attribute for a long file name is 0x0F. The long file name may require more than one entry. The first byte of a long filename entry is the ordinal number. The first entry of a long file name has an ordinal number of 0x01. The last entry of a long file name has the ordinal number OR'ed with 0x40. The entries of a long file are in memory in reverse order where the last entry of the name is first in memory (at the lowest address). More information on long file names can be found in section 7 of the Microsoft FAT Specification.

The short file name entry associated with the long file name immediately follows the long file name entries. A check sum of the short file name is at offset address 0x0D of each long file name entry. Section 7.2 of the Microsoft FAT Specification details how this checksum is generated. The short file name entry has the short file name stored in the offset addresses 0x00 to 0x0A. The pattern is 8 bytes for the name and three bytes for the extension. An implied period is between the name and extension. The attribute value for a directory is 0x1X (X indicates other attribute bits) and a file has the attribute value of 0x2X. Additional attribute values may be added to these for read-only (0x01), hidden (0x02) and system (0x04). The short file name entry has the cluster information required for locating a sub-directory or a file. If the FAT type is FAT16, then the start cluster of the file or sub-directory is a 16-bit value found at offset 0x1A. If the FAT type is FAT32, then the low word is found at offset 0x1A. The upper word is a 16-bit value found at offset 0x14. These two values are combined to create the 32-bit cluster number. The file size is a 32-bit value found at offset 0x1C. More information on directories can be found in section 6 of the Microsoft FAT Specification.

The Print_Directory and Read_Dir_Entry functions

Two functions will be provided to the programmer for this experiment. Both files require some of the global variables that were stored when the drive was mounted. This means that these functions need a method of sharing the global values that were found when mounting the drive. These methods will be discussed in the procedure section. Some editing may be required to make sure the names match your system.

The Print_Directory function has a starting sector as an input parameter and a pointer to a 512-byte block of SRAM. It reads the starting sector into SRAM and looks for short file name entries. It creates an enumerated list of entries printing the name and [DIR] to indicate directories. It does not include hidden or system files on the list. The function

searches the entire cluster until it finds the last entry of the directory and returns the number of entries found. If it finishes the current cluster before finding the last entry of the directory, then it sets the most significant bit of the return value. The programmer may choose to check this bit and write the code to support directories that span multiple clusters. This is optional and not required for this experiment. This optional code should save the number of entries found and then determine the starting sector of the next cluster of the directory. The Print_Directory function can be called again, and the number of entries returned is then added to the previously saved value. The programmer will need to determine how to handle the extended enumerated list, since the Print_Directory function starts the enumeration at 1 each time it is called.

The Read_Dir_Entry function requires the same starting sector that was given to the Print_Directory function as well as a pointer to a 512-byte block of SRAM. Another parameter sent to this function is the directory entry number that should be opened. It uses the same method as the Print_Directory function to find entries. When the entry number matches, the function reads the cluster number and returns that value. If the entry number could not be found or some other error occurs, then the most significant bit of the return value will be set (return_value=0x80000000). If the entry that is located is a directory, then bit 28 is set (return_value=0x10000000|first cluster number) and if the entry is a file, then just the first cluster number is returned. The programmer can use this cluster number to determine the starting sector of that entry.

If the entry is a directory, then the upper four bits should be removed so that bit 28 being set does not mess up the first sector calculation. Then the cluster number should be used to determine the starting sector of that directory. The sector number can then be passed to the Print_Directory function to list the contents of that directory.

If the entry is a file, then the cluster number can be used to print the contents of the file one sector at a time. In a later experiment, the cluster number will be sent to a function that sends the contents of the file to the MP3 decoder to play the song.

Procedure

The following equipment will be required for this part of the experiment:

- ATMEGA324PB Xplained Development board
- IO1 Xplained expansion board with SD card
- PC with
    - Atmel Studio 7
    - Terminal Program (Putty)
- Source code files
    - Print_Directory function
    - Read_Dir_Entry function
    - Solution files for Experiment#3

Bonus points (up to 10 bonus points) will be given for supporting FAT16 in addition to FAT32. There are a few things that are required to support FAT16. The first item is that

FAT16 is usually on standard capacity cards. These cards have a default block size of one byte, so if a standard capacity card is detected CMD16 should be used to adjust the block size to 512 bytes for easier compatibility (The Experiment #3 solution files already adjust the block size of standard capacity cards). Also, the CMD17 address parameter is for a byte address instead of a block number on a standard definition card. The calculated sector number (block number) must be converted to a byte address by multiplying by block size (512) when accessing a standard capacity card. I accomplished this by placing a SD card type dependent left shift on the sector number. The shift is 0 (no shift) for high-capacity cards and 9 (same as multiply by 512) for standard capacity cards. This shift can be placed in the read_sector function so the adjustment is made anytime a sector is loaded from the SD Card using that function. Another issue arises concerning FAT16 support when reading from the file allocation table. A FAT16 entry has two bytes (16-bits) and a FAT32 entry has four bytes (32-bits). The appropriate read value function should be used when reading FAT entries and they both have some required modifications to the entry number. This will be needed in the find_next_clus function in step 6. The final item needed for supporting FAT16 is finding the root directory and placing the appropriate limit on the number of sectors which would be done in the mount drive function. I have a few FAT16 formatted SD cards available for testing when working in the 210 lab. They should not be removed from the 210 lab and only borrowed for a short amount of time so that all students who wish to use them have access to them. Please return standard capacity SD cards to me after this experiment is complete.

There are two sets of directory function files. One uses global values, and these are defined in the C source code file. It is up to the programmer to decide on a method of accessing these global variables, such as using the extern keyword to share the declarations to other files or writing read and write functions for the variables. The other defines a FS_values_t type that is a structure which holds all the values for the file system. A function is provided that will return a pointer to a declared structure. The programmer can declare a pointer and use the returned pointer value to access the structure values as needed. The arrow operator is used to access a structure using a pointer (for example: structure_pointer->element_name=value to assign a value to a structure element).

1) It is recommended that before writing any C code for this experiment, the programmer should use the code from experiment#3 to display SD card blocks and do the calculations by hand to find the sector numbers for the BIOS Parameter Block (BPB), the start of the FAT, the root directory and the first data sector (Note that on FAT32 disks the root directory is typically in the first data sector). By doing this the programmer will know how to perform the calculations and if the values calculated in the program are correct. The numbers I determined from my SD card should match yours (but may not) and are given for a guide: Relative Sectors=8192 (sector of the BPB), RsvdSecCnt=6332, StartofFAT=14524 and FirstDataSec=16384. This is a FAT32 type file system and the RootClus=2, so the root directory is also

located at 16384.

2) The print_directory and read_dir_entry functions are provided to the programmer. They can be included into the programmer's file system source code file, or the programmer may wish to leave them in a separate source code file. The given files define the global values needed for the file system and used in these functions. It is up to the programmer to determine how to access these global variables. If you are using the struct, a function is given to export the address of the struct for use as a pointer in other files.

3) (Func: 5pts) Create a separate code and header file for a read sector function. This function will contain <u>all</u> the steps required to read one sector (block) from the SD Card (including nCS assertion). The prototype should match the function calls in the given directory functions source code or you will need to edit these files. Therefore, the prototype should be:
*uint8_t read_sector(uint32_t sector_number, uint16_t sector_size, uint8_t * array_for_data);*

   The return value is an error flag that indicates that the sector was not able to be read.

4) (Func: 5pts) Write three functions that when given an offset address can read an 8-bit value, a 16-bit value and a 32-bit value. A second parameter for this function is a pointer to an array stored in SRAM where the sector of data from the SD Card is stored. Note that multi-byte values are stored in little-endian format on MS FAT File System devices. The recommended prototypes for these functions are:

   uint8_t read_value_8 (uint16_t offset, uint8_t array_name[]);
   uint16_t read_value_16 (uint16_t offset, uint8_t array_name[]);
   uint32_t read_value_32 (uint16_t offset, uint8_t array_name[]);

   The functions should accomplish the following tasks:

   a. Add the offset value to the array_name pointer to determine the starting address of the value to be read.
   b. Read each byte into the correct location of the return value. I did this by first setting the return value to 0. The return value is left shifted by 8 to make room for the next byte to be read. Each byte is read (starting with the MSB) and ORed with the return value. The process is repeated until all the bytes of the number are read.
   c. It may be helpful to print the values returned by these functions to verify that they are working correctly. The uint32_t, uint16_t and uint8_t values can be printed with: *printf("%lu",uint32_value);* *printf("%u",uint16_value);* and *printf("%u",uint8_value);* You can use *%lX* and *%X* to print hexadecimal values instead of decimal values if it makes it easier to verify the value. The l modifier will be marked by the editor, but compiles without errors or warnings and is

needed to print 32-bit values correctly.

5) (Func: 15pts)Write an initialization or mount drive function for the file system. The recommended prototype for this function is:

uint8_t mount_drive(void); or *uint8_t mount_drive(uint8_t * array);*

The return value is an error flag indicating if the drive was mounted successfully or not. It should accomplish the following tasks:

a. The first step is to find the BIOS Parameter Block (BPB). First read sector number 0 into SRAM. Determine if this sector is the BIOS Parameter Block (BPB) or the Master Boot Record (MBR) by reading the byte at offset 0. If it is not 0xEB or 0xE9, then this is most likely the MBR. If it is the MBR, then use the *read32* function to read the relative sectors value at offset 0x01C6. The programmer may wish to print this value for debugging purposes. This value should be the sector number for the BPB. Read the possible BPB sector into SRAM. Determine if this sector is the BPB by reading the byte at offset 0. If it is not 0xEB or 0xE9, then return with an error (BPB not found).

b. Read the values outlined in the background information and perform the calculations to determine the FAT type. The programmer may wish to print values and the FAT determination for debugging purposes. Bonus points will be given for adding the code to support FAT16. <u>If FAT16 is not supported, then the function should exit with an error if a FAT16 SD card is detected.</u>

c. Perform the calculations to determine the starting sector number for the FAT, the first data sector and the starting sector of the root directory. These values should be stored as global variables. Again, the programmer may wish to print these values for debugging purposes.

(Debug) If you have not already, the programmer may wish to debug the mount drive source code. Add the *mount_drive* function to the initialization section of the main function at some point after the SD card is initialized. Print or use the debugger to observe the values calculated in the *mount_drive* function.

6) (Func: 4pts) Write a function that calculates the starting sector of a cluster. This function has an input parameter of the cluster number and performs the calculation for First Sector of a Cluster given in section 6.7 of the Microsoft FAT specification. The return value for this function is the sector number. This gives a function prototype of:

uint32_t first_sector(uint32_t cluster_num);

One issue that this function should address is that the cluster number for the root directory is given as 0. Therefore, if the input parameter is 0, then the return value should be the first sector of the root directory.

7) (Func: 7pts) When all the sectors of a cluster have been accessed, the file allocation table (FAT) must be accessed to read the next cluster in the file. Write a function that when given the current cluster number, it returns the next cluster number of the file. This function will also need a pointer to an available 512-byte array in SRAM. The function parameter would then look like:

   uint32_t find_next_clus(uint32_t cluster_num, uint8_t array[]);

   The function should perform the following steps:
   a. Calculate the sector number in which the cluster number entry can be found (ThisFATSecNum).
   b. Use the read sector function to read that sector into the 512-byte array in SRAM.
   c. Calculate the offset address for the cluster number entry (ThisFATEntOffset).
   d. Use the read32 (for FAT32) or read16 (for FAT16, if supported) to read the entry.
   e. If the value is for a FAT32 system, the cluster number only uses 28-bits and the upper four bits must be masked off before returning the value.
   f. If the value is for a FAT16 system, it should be typecast into a uint32_t and then returned. The end of file value for FAT16 (0xFFFF) will need to be modified to be the same as the end of file value for FAT32 (0x0FFFFFFF).

8) (Debug)If you did not start debugging your source code earlier, then The project from experiment #3 would be a good starting point for creating the project for experiment #5. Add the *mount_drive* function to the initialization section at some point after the SD card is initialized. The initialization section should conclude with assigning a current directory variable with the First Root Directory Sector. The main function will then enter the super loop described in the next step. For debugging purposes, the programmer may wish to call the print_directory function with the first root directory sector and create an endless do-nothing super loop at this point. This will allow the programmer to verify that the SD card is mounted as a drive correctly and that the root directory is printed. In the next step, the print_directory function will be moved to be called inside the super loop.

9) (Func: 5pts) The super loop should first call the print_directory function with the current directory variable. The print_directory function returns the number of entries found in the directory. Next, prompt the user for an entry number.

An error check should be done to reject an entry number larger than the number of entries found in a directory. This entry number should then be used as the input parameter for the read_dir_entry function. This will return a cluster number and bit 28 will be set '1' for a directory or clear '0' for a file. Bit 31 will be set to '1' if an error occurred. Mask these upper four bits off and use this cluster number to update the current directory value or open a file.

(Func: 4pts) If the directory entry selected is another directory (a sub-directory), then use the first_sector function to calculate the first sector of that directory and update the current directory value. The super loop should then repeat which will call the print_directory function and prompt the user for the next entry number.

(Func: 5pts) If the directory entry selected is a file, then the file should be opened. At this point, opening the file will mean that it should be printed sector by sector. A print_file function can be created to do all the steps described. The first cluster number of the file is sent to the function along with a pointer to a 512-byte array. Use the first_sector function to find the first sector of the file. And then use the read_sector and print_memory functions to print that sector. Prompt the user to press a key to continue or press another to exit. If the user presses the continue key, then print the next sector. If all the sectors of a cluster are printed, then use the find_next_clus function to find the next cluster and the first_sector function to find the first sector of that cluster. The programmer might find it useful for debugging to print the cluster and sector number before each block is printed. I also printed the sector of the cluster so that I knew when a new cluster should be found. If the user presses the exit key or if the end of the file is printed, then return to the super loop. In the super loop, print the current directory and then prompt for the next entry number. The purpose of the print_file function is to verify that a file can be found (look for the text ID3 to start some of files). Another purpose is to test the find_next_clus function. Since there can be several sectors in a cluster, the programmer may wish to print all the sectors in a cluster without stopping to shorten testing time.

10) (up to 15 total bonus points) Another bonus part for this experiment is to create a function that can print the long file name of a directory entry. This function can be added to the print_directory function to print the long file name instead of the short file name. One caution is that not every directory entry has a long file name entry (the . and .. directory entries only have short file names), so the short file name must be printed in those situations. Information on long file name entries is given in section 7 of the Microsoft FAT specification.

11) Make sure each group member is named in the comments. Upload the completed project to Canvas for grading.

Grading:

Functionality:	Code works as described in the assignment: Registers set as needed and actions occur as specified; Calculations are correct in code;…	Points awarded as specified in the steps marked graded. (50 points)
Compile Errors:	If submitted code does not compile, then a minimum of 25% of the functionality points will be deducted. More may be deducted for multiple errors.	Minimum of 25% of functionality points deducted.
Organization:	Separating hardware drivers (read_sectors), and application code.	10 points
Readability:	Source code is well commented. Descriptive names are used for functions, constants, and variables.	10 points
Correctness	No patches or work-arounds are used in the source code to get the correct functionality.	10 points
Bonus:	FAT16 support and/or Long File Name Support (10 pts each, 15pts maximum)	Up to 15 points