FAT32 File Structure

Prof. James L. Frankel
Harvard University

Version of 7:09 PM 30-Sep-2025

FAT32 Source Documentation

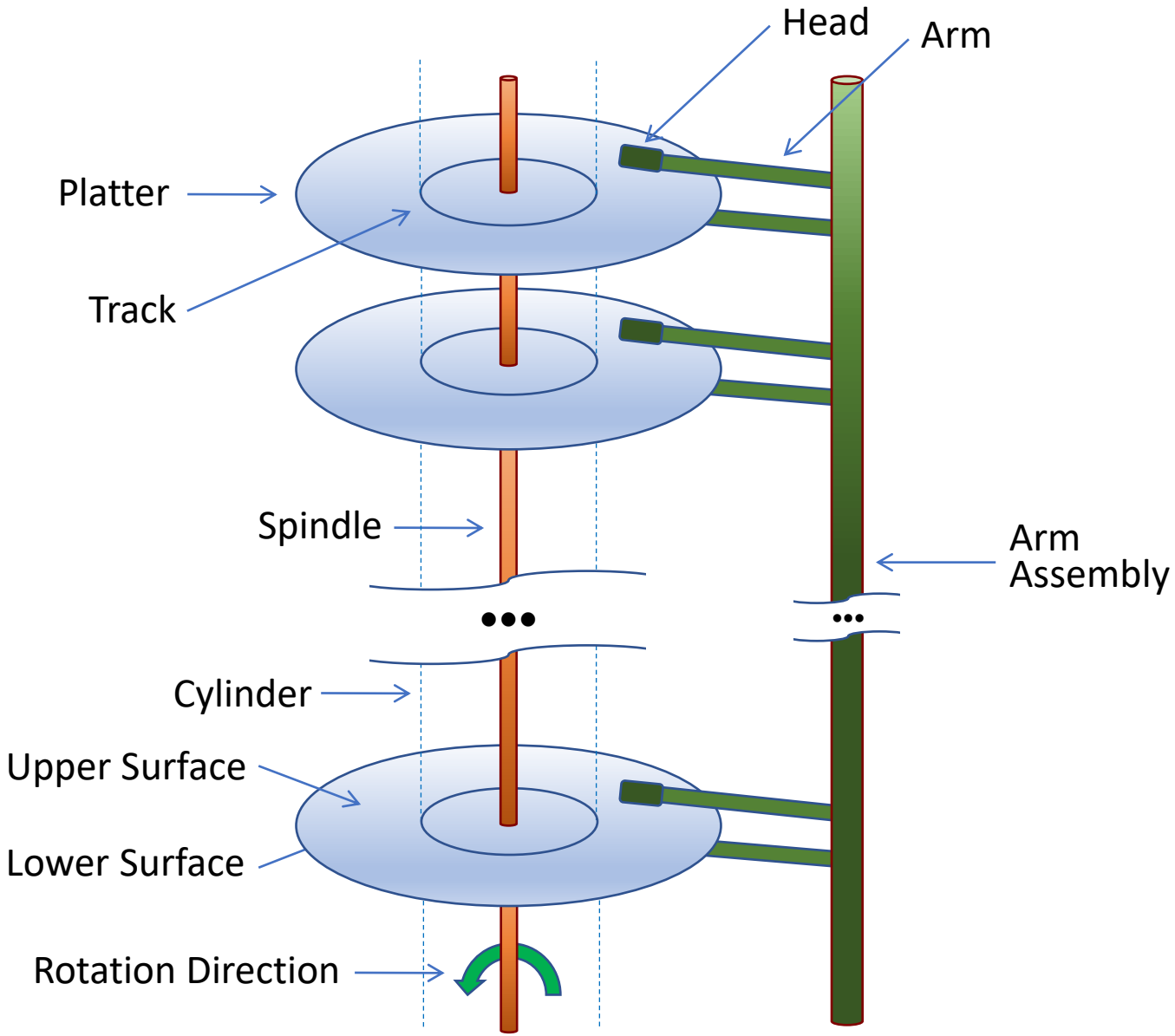
- The reference document you should use is the **Microsoft Extensible Firmware Initiative FAT32 File System Specification**
  - On class web site under The NXP/Freescale ARM -> microSDHC Card
  - It is available on the class web site at https://cscie92.dce.harvard.edu/fall2025/Microsoft%20Extensible%20Firmware%20Initiative%20FAT32%20File%20System%20Specification,%20Version%201.03,%2020001206.pdf under **The NXP/Freescale ARM -> SD Documents**
- Important correction to this document concerns the DIR_CrtTimeTenth field in the FAT 32 Byte Directory Entry Structure
  - The name and description of this field is incorrect
  - Instead of DIR_CrtTimeTenth, we will use the name DIR_CrtTimeHundth
  - Here is the correct description of this field (to update the text on page 23):
    - Hundredths of a second time at file creation time. This field contains a count of hundredths of a second. Because the seconds portion of the DIR_CrtTime field denotes a creation time with a granularity of 2 seconds, this field contains a number of hundredths of a second (0 to 199, inclusively) that denotes a number of seconds from 0 to 1.99, inclusively, that may increment the number of seconds in addition to supplying the number of hundredths of a second.
- There is also a typo on page 25 where a field is referred to as DIR_CrtTimeMil (which does not exist), and, as corrected here, should be DIR_CrtTimeHundth

SD Documentation

- Documentation for the SD controller in the K70
  - K70 Sub-Family Reference Manual, Rev. 4, Oct-2015 in Chapter 57 (PDF Page 2004)
  - Describes how to interact with the SD card
    - Send commands to the card, receive results from the card, etc.
- Documentation for the SD card
  - On class web site under The NXP/Freescale ARM -> SD Documents
  - SD Association, Part 1 Simplified: Physical Layer Simplified Specification, Version 8.00
  - Describes how to interact with the processor inside the SD card
    - Commands that the card understands, format of results from the card, etc.

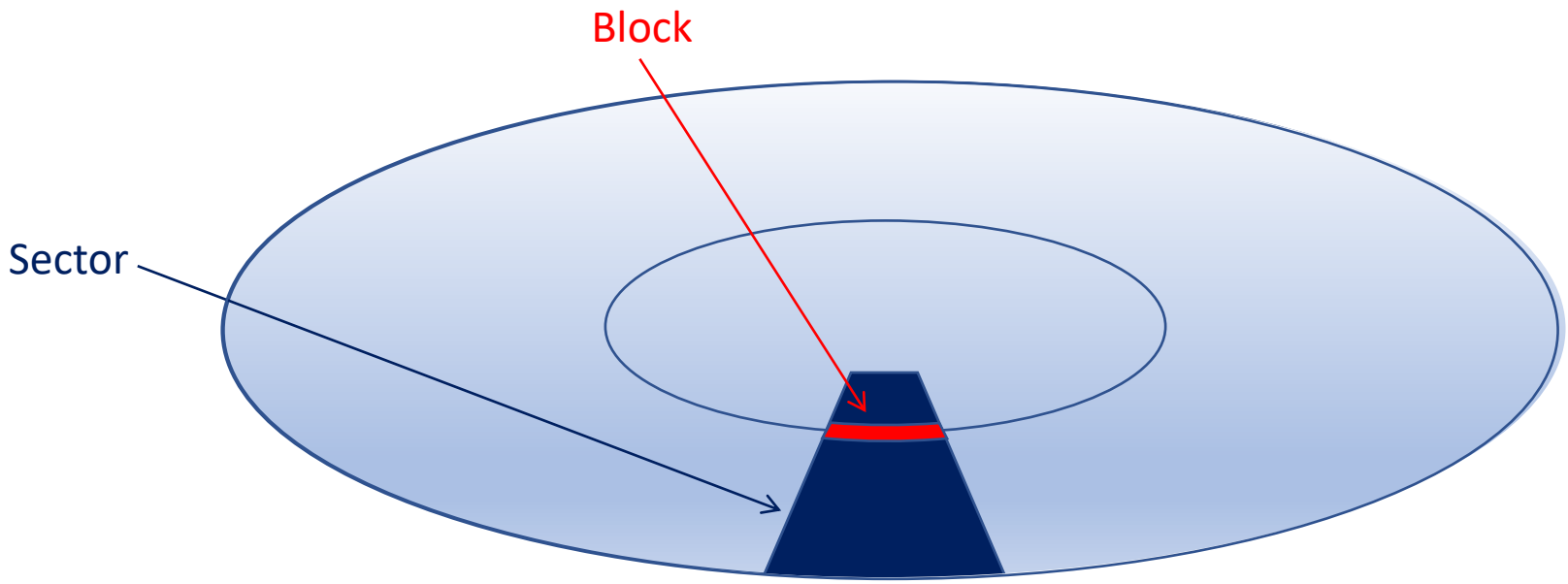
Historical Perspective of Mass Storage Devices

- A **platter** is a disk (usually made of aluminum) coated with a magnetic (ferrite) material
- A platter may have an upper and a lower **surface**
- Disks are stacked together and are held by a central **spindle**
- The spindle is used to rotate all of the disks
- A set of recording/playback **heads** is moved by an **arm assembly** to be positioned at a particular radius from the spindle on all disks simultaneously
  - Each head is able to read or write bits to a disk; bits are written serially one bit at a time
- A single head is suspended above a **track** on the disk
- All tracks at an arm assembly position is referred to as a **cylinder**


Head
Arm
Platter
Track
Spindle
Arm
Assembly
Cylinder
Upper Surface
Lower Surface
Rotation Direction

Addressing Data on the Mass Storage Device

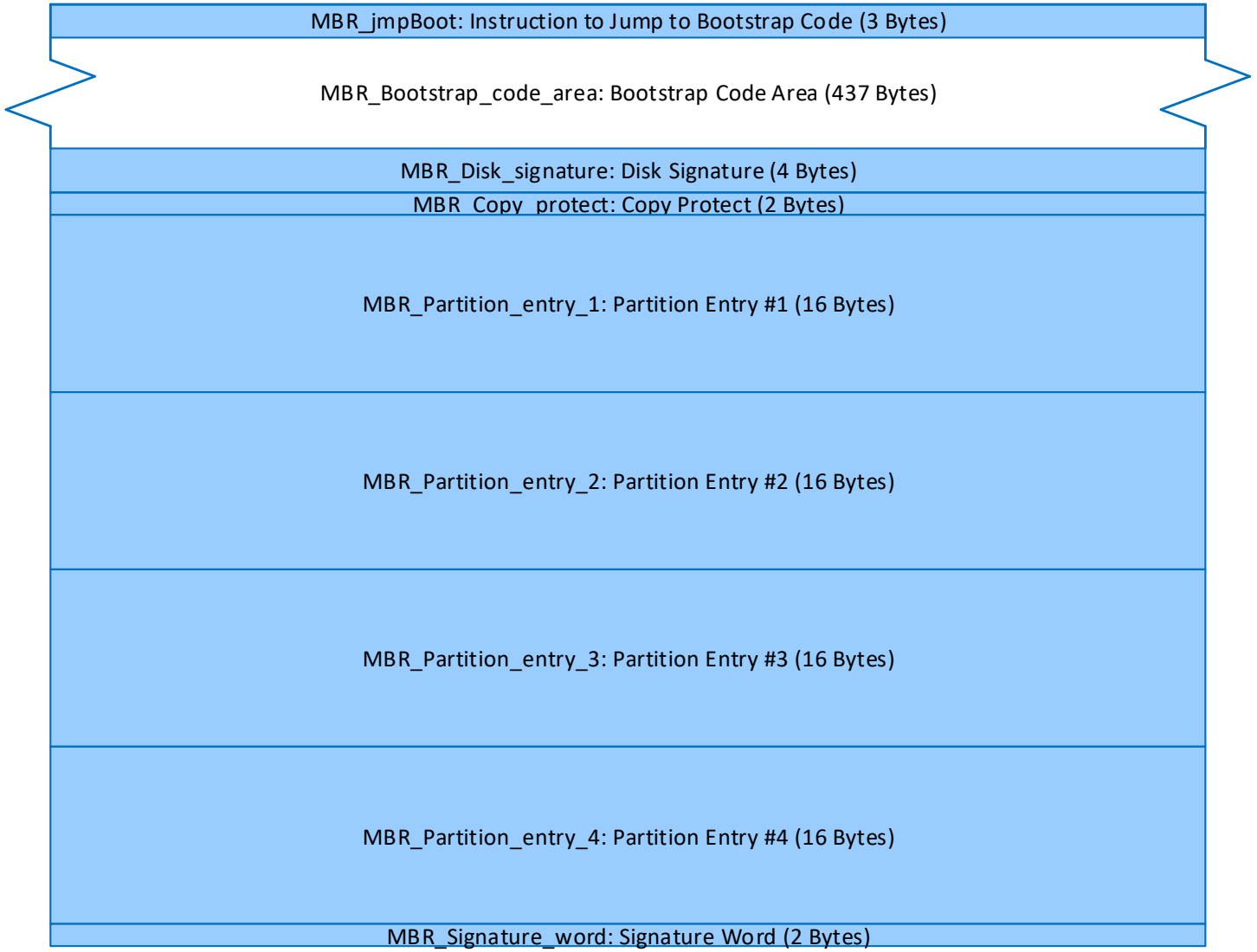
- Sectors
  - A sector is formally a pie-shaped wedge of a circle/disk, however, we now use the term to denote the intersection of a pie-shaped wedge and a track
  - The term **sector** is used to refer to the smallest readable or writeable quantity
    - The term **block** is used to refer to the operating system/software smallest readable or writeable quantity and *is used synonymously with sector*
    - In our code, block is the term used in the lowest-level microSDHC code
  - Every sector is an equal length sequence of bytes
  - Our sectors must be 512 bytes long
- Originally a sector address was given by a **Cylinder/Head/Sector** address (**CHS**)
  - CHS numbers are 1-origin unsigned integer (merely historical, may have holes in the numbering scheme)
- Now sector address is usually a **Logical Block Address** (**LBA**)
  - 0-origin unsigned integer


Block
Sector

Master Boot Record (MBR)

- First sector on Mass Storage Device (Physical Sector 0) is either:
  - Boot Sector (also known as the Volume Boot Record) of a file system/structure, if the Mass Storage Device is *not partitioned*
  - MBR, if the Mass Storage Device is *partitioned*
- The first sector originally contained the boot loader
  - BIOS (Basic Input/Output System) in ROM or Flash loads the boot loader
  - UEFI (Unified Extensible Firmware Interface) is the BIOS replacement
- The MBR has gone through many iterations
  - MBR
    - The boot loader was later deprecated
    - Mass Storage Device was limited to 2TB (32-bit sector number * 512-byte sector size)
    - Up to four primary partitions
    - Augmented to allow an extended partition
  - GUID Partition Table (GPT)
    - Can co-exist with MBR
    - Allows MSDs larger than 2TB

Master Boot Record (MBR)



MBR Fields

- MBR_jmpBoot
  - x86 instruction to jump to boot loader
- MBR_Disk_signature
  - Identifies this disk
- MBR_Copy_protect
  - If contains 0x5A5A, then copy protected
- MBR_Partition_entry_1 through 4
  - Partition entries for primary partitions 1 through 4
- MBR_Signature_word
  - Must be 0xAA55
  - Confirms that this is either an MBR or a boot sector

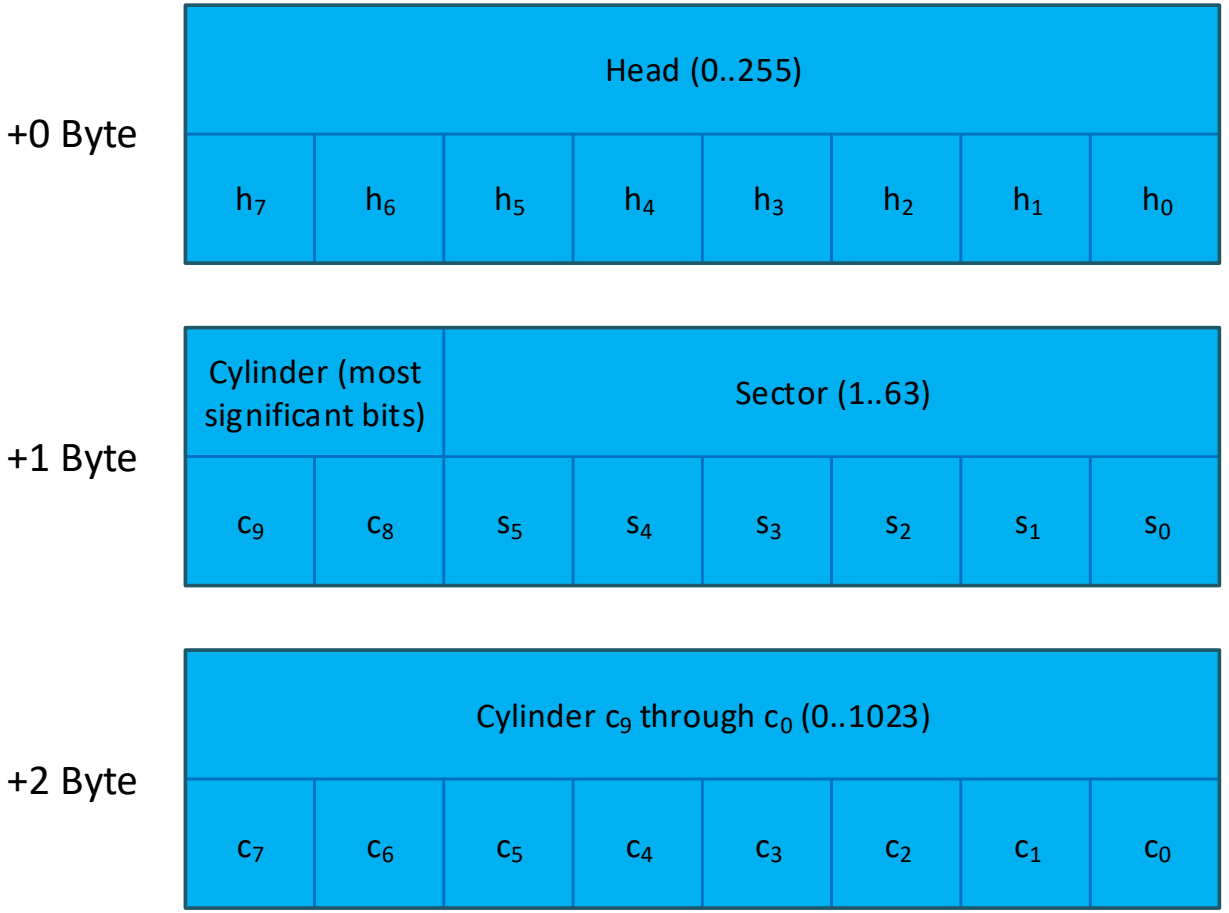
MBR Partition Entry

status: Status (1 Byte)
first_sector_CHS_address: CHS (Cylinder/Head/Sector) Address of First Absolute Sector in Partition (3 Bytes)
type: Partition Type (1 Byte)
last_sector_CHS_address: CHS (Cylinder/Head/Sector) Address of the Last Absolute Sector in Partition (3 Bytes)
first_sector_LBA: LBA of First Absolute Sector in Partition (4 Bytes)
sectors_in_partition: Number of Sectors in Partition (4 Bytes)

MBR Partition Entry Fields

- status
  - Most significant bit indicates if the partition is bootable (active)
- first_sector_CHS_address
  - CHS (Cylinder/Head/Sector) bits are shown on the next slide
- type
  - Partition type (see partial table on following slide)
- last_sector_CHS_address
- first_sector_LBA
  - Logical Block Address of first absolute sector in partition
- sectors_in_partition
  - Number of sectors in partition

Cylinder/Head/Sector (CHS) Format



Partition Types – Partial Table

Type Code	Meaning
0x00	Empty partition entry
0x07	NTFS
0x0B	FAT32 with CHS (Cylinder, Head, Sector) Addressing
0x0C	FAT32 with LBA
0x0F	Extended Partition with LBA
0x82	Linux Swap Space
0x83	Linux File System

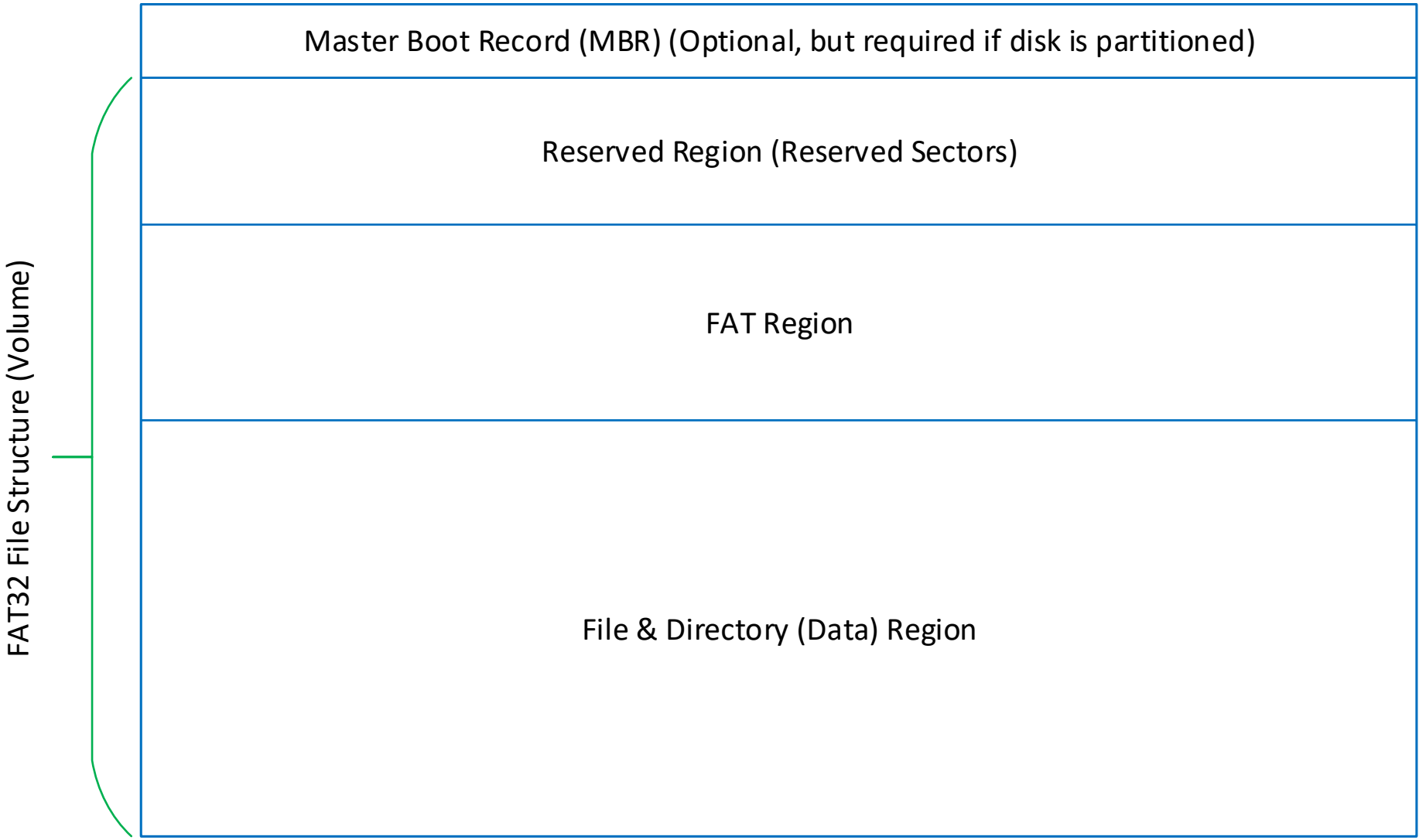
FAT32 Basic Concepts

- Clusters
  - A **cluster** is a ***contiguous*** group of a fixed number of sectors
  - Clusters are addressed with a 0-origin integer
- The type of a sector number and a cluster number is uint32_t
- A FAT32 file structure is composed of sectors and clusters

Byte Order of Integers in FAT32

- Each sector can be thought of as an array of bytes (uint8_t)
- If a multibyte integer is present in a FAT32 file structure sector (*not necessarily including user binary data*), it is represented in **little endian byte order**
  - This means that the LSB (Least Significant Byte) of that integer is stored in the lowest address byte in that integer in the sector
  - Correspondingly, the MSB (Most Significant Byte) of that integer is stored in the highest address byte in that integer in the sector
- Interesting note: Over IP (Internet Protocol) **big endian byte order** is used (and is referred to as **network byte order**)
- The FAT32 standard includes 16- and 32-bit integers for which this applies
- The NXP K70 ARM also represents integers in memory in little endian order
- When writing portable code, if the computer uses big endian byte order, the bytes in multibyte integers need to be reordered on both input and output

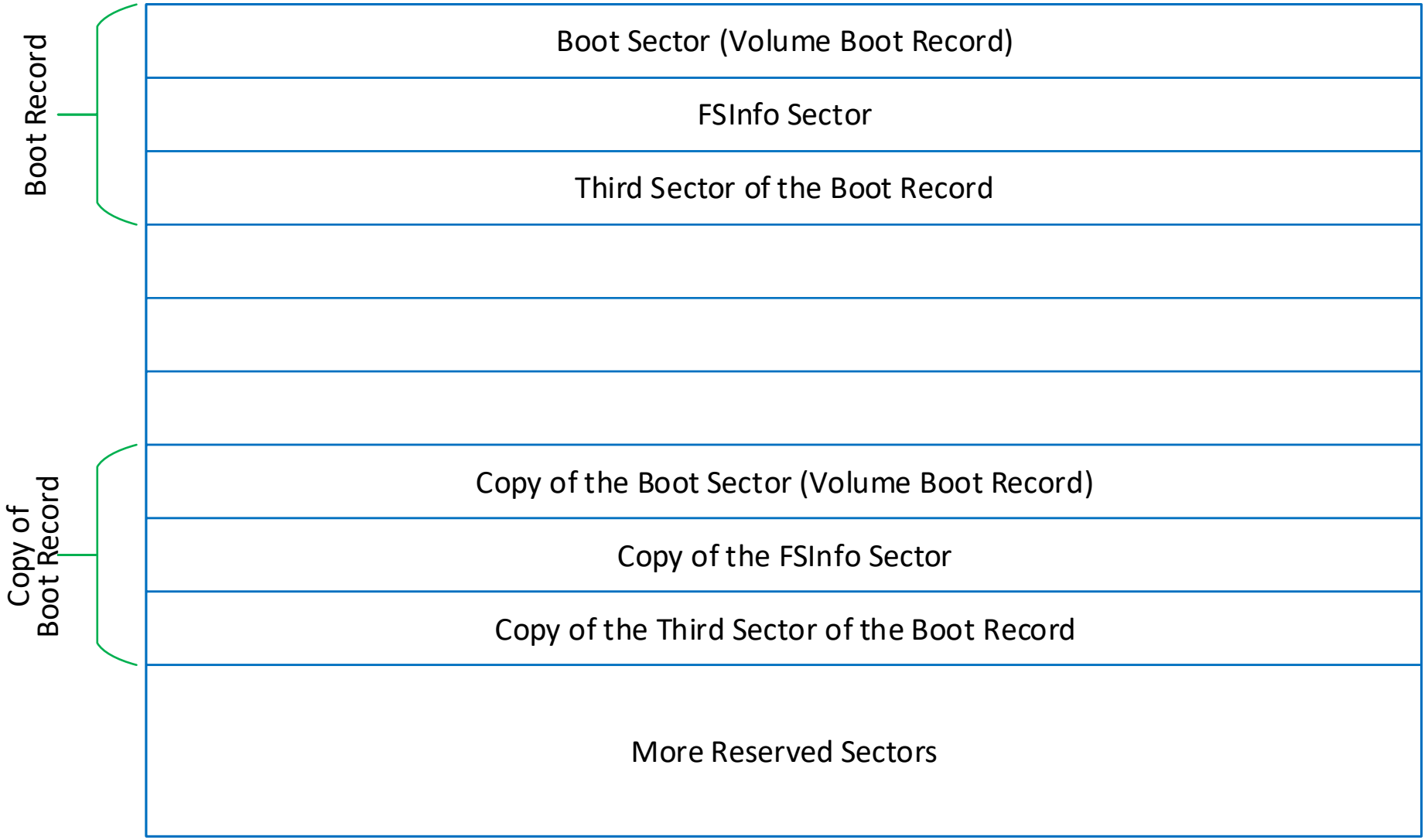
Overall Mass Storage Device Structure



Regions in a FAT32 File Structure

- Starting at sector 0, there are **reserved sectors**
  - These sectors as a group are referred to as the **Reserved Region**
- Following the reserved sectors is the **FAT Region**
  - This region has two copies of the FAT
    - The number of FATs is stored in the boot sector (BPB_NumFATs)
  - The FAT indicates which clusters are used and which are free
    - For used clusters, the FAT indicates the next cluster that logically follows in a linked-list fashion
- Following the FATs are the data clusters
  - This region begins on an even cluster boundary
  - The first file in the data clusters is the root directory
    - **The number of the first cluster (the root directory cluster) is 2 (root_directory_cluster)**
  - This set of data clusters is referred to as the **File & Directory (Data) Region**

Reserved Region



The Reserved Sectors

- The number of reserved sectors is stored in the boot sector (BPB_RsvdSecCnt)

- Sector 0: The boot sector
  - Includes the BIOS Parameter Block (BPB)
- Sector 1 (BPB_FSInfo): The FSInfo sector (Second sector of the boot record)
- Sector 2: The third sector of the boot record
- Sector 6 (BPB_BkBootSec): Copy of the boot sector (First sector of the copy of the boot record)
- Sectors 7 & 8: The second and third sectors of the copy of the boot record

- Let’s look at them in more detail

Boot Sector

- Sector 0 is the boot sector
- In addition to other data, the boot sector contains:
  - The number of bytes per sector
  - The sector number of the FSInfo sector
  - The number of sectors per cluster
  - The first sector number of the FAT
  - The number of sectors per FAT
  - The number of the first cluster in the root directory
  - The total number of data clusters (total_data_clusters)
    - Note that total_data_clusters is ***not*** the number of the last data cluster + 1 because the first data cluster is where the root directory is located (root_directory_cluster) is cluster number 2 – not 0

FSInfo Sector

FSInfo Sector

- The sector number of the FSInfo Sector is in the boot sector
- The FSInfo sector has two hints about free clusters
- FSI_Nxt_Free is a suggested starting cluster number from which to search for a free cluster
  - If it is equal to FSI_NXT_FREE_UNKNOWN, then this field must not be used and the search for a free cluster must start with cluster 2 (root_directory_cluster)
  - This field is furnished to reduce the time to find a free cluster
- FSI_Free_Count is the approximate number of free clusters
  - If it is equal to FSI_FREE_COUNT_UNKNOWN, then this field must not be used and there is no estimate for the number of free clusters
  - This field is furnished to approximate the free space in the FAT32 file structure

File Allocation Table (FAT)

- The File Allocation Table (FAT) itself is composed of a set of contiguous sectors starting at sector first_FAT_sector
- BPB_RsvdSecCnt from the boot sector contains the number of the first sector in the FAT
- BPB_FATSz32 from the boot sector contains the number of sectors in each FAT
- BPB_NumFATs from the boot sector contains the number of FATs (additional FATs are for reliability/redundancy)
  - So, BPB_RsvdSecCnt+(BPB_NumFATs*BPB_FATSz32) is equal to the first data sector number
- Each sector in the FAT contains a list of uint32_t cluster numbers

FAT Details

- The first two entries in the FAT are not used for allocation
- Only 28 bits of each FAT entry are used, but the existing high-order 4 bits (from formatting) must be maintained on a write
- FAT[0] == 0xFFFFF00 | BPB_Media
- FAT[1] == 0xFFFFFFF == FAT_ENTRY_ALLOCATED_AND_END_OF_FILE
- Each FAT entry indicates the state of that numbered cluster; thus:
  - if FAT[cluster] == FAT_ENTRY_FREE, then cluster is **free**
  - if FAT[cluster] >= FAT_ENTRY_RESERVED_TO_END, then cluster is **used** (*i.e.,* allocated) & is the **last cluster in the file** (*i.e.,* end-of-file)
  - if FAT[cluster] == FAT_ENTRY_DEFECTIVE_CLUSTER, then cluster is **defective**
  - otherwise, the cluster is **used**, and FAT[cluster] contains the **next cluster number** in the file in a linked list of clusters
- When writing the FAT entry for the last cluster in a file, use the value FAT_ENTRY_ALLOCATED_AND_END_OF_FILE
- See the next slide for an illustration of a possible FAT and Data Region

This illustration shows a FAT and Data Region in which: (1) cluster 2 (in the data region) is used and has no following clusters, (2) cluster 3 is used and is followed by clusters 5 then 8, (3) cluster 4 is used and has no following clusters, (4) all other clusters are free

FAT[0] & FAT[1] are special; In the **FAT32 File Structure** slides, see the **FAT Detail** slide

◯ is being used to designate the FAT_ENTRY_FREE #define'd constant

× is being used to designate the FAT_ENTRY_ALLOCATED_AND_END_OF_FILE #define'd constant

FAT Details from some Windows OSes

- Two high-order FAT[1] bits may be used to indicate if a volume was cleanly removed and if a volume had an I/O error during the last time it was mounted
  - FAT_ClnShutBitMask
    - If bit is 1, volume is “clean”
    - If bit is 0, volume is “dirty”
      - This indicates that the file system driver did not dismount the volume properly the last time it had the volume mounted
      - It would be a good idea to run a Chkdsk/Scandisk disk repair utility on it, because it may be damaged
  - FAT_HrdErrBitMask
    - If this bit is 1, no disk read/write errors were encountered
    - If this bit is 0, the file system driver encountered a disk I/O error on the volume the last time it was mounted, which is an indicator that some sectors may have gone bad on the volume
      - It would be a good idea to run a Chkdsk/Scandisk disk repair utility that does surface analysis on it to look for new bad sectors

Cluster Allocation

- To find a free cluster,
  - If FSI_Nxt_Free is not FSI_NXT_FREE_UNKNOWN, then start looking in the FAT at the entry for cluster FSI_Nxt_Free
  - If FSI_Nxt_Free is equal to FSI_NXT_FREE_UNKNOWN, then start looking in the FAT at the entry for cluster 2 (root_directory_cluster)
- The last valid cluster number in the FAT is total_data_clusters+1
  - Remember: FAT entry numbers 0 & 1 are special, 2 (root_directory_cluster) is the first data cluster, etc.
  - Because each sector is 512 bytes and because each FAT entry is 4 bytes, the number of FAT entries in a sector is 128
    - For a 32GB microSDHC with 32 sectors per cluster,
      - The total data clusters = 1,946,912
      - Therefore, the FAT needs 1,946,912/128 = 15,210.25 sectors for FAT entries
      - My boot sector reports: BPB_FATSz32 = 15,211 sectors for each FAT, so that makes sense!

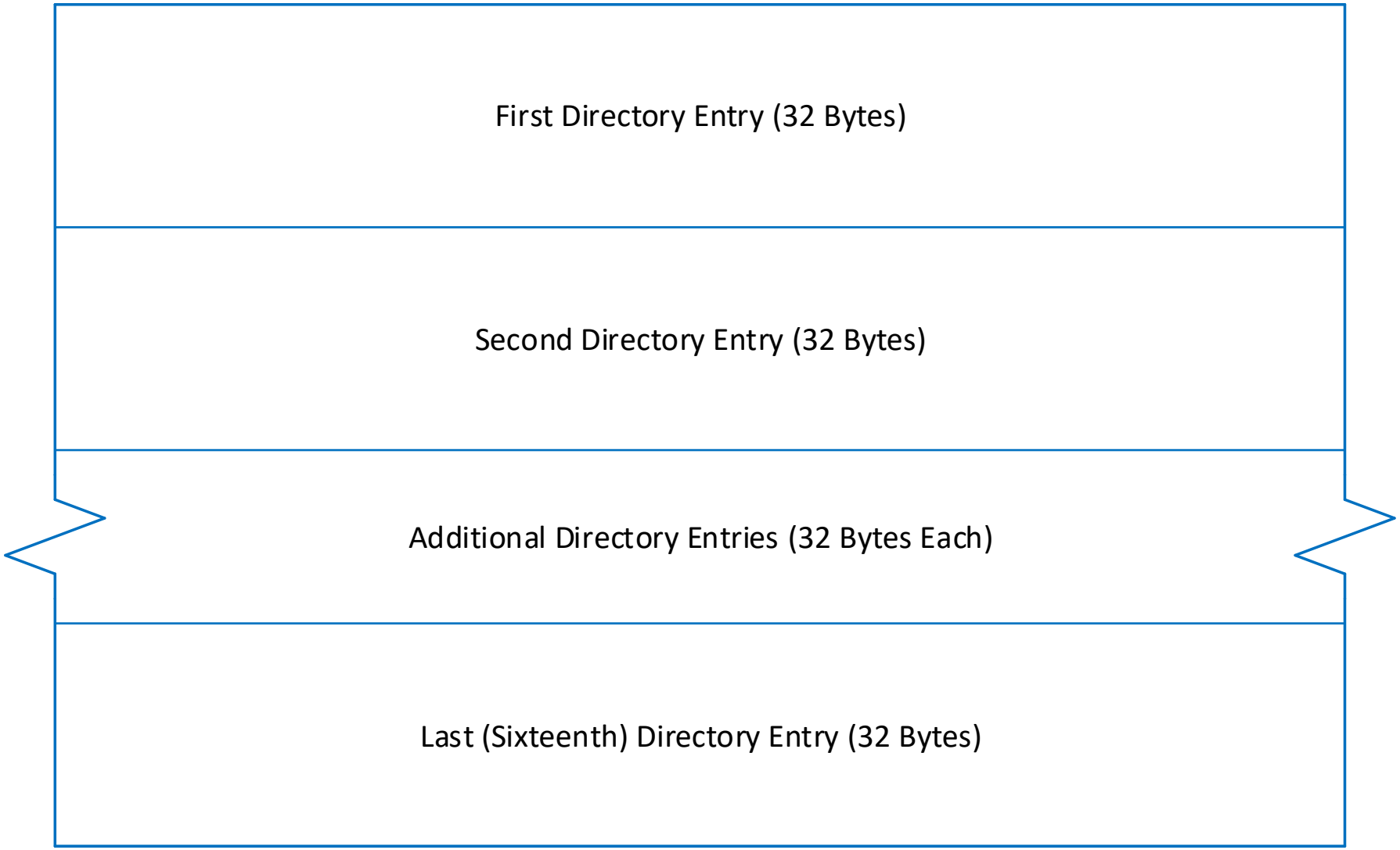
Files (Both Regular Files and Directories)

- Files are allocated in units of clusters
- There may be unused bytes at the end of the last sector
- There may be unused sectors at the end of the last cluster
- Bytes in a file are accessed by first reading bytes in a sector, then continuing to read sectors in a cluster, then progressing to the next linked cluster

- The end of a file of a file is determined by knowing the length of the file from the directory

- A file may be either a regular file or a directory

Each Sector in a Directory



Directories

- Since a directory is just a file, it is composed of an integral number of clusters
- A header file containing struct's for directories is available in the class web site as directory.h
- Each sector in a directory is composed of directory entries; each directory entry is 32 bytes in size (*i.e,* sizeof(struct dir_entry_8_3))
- There are bytes_per_sector/sizeof(struct dir_entry_8_3) directory entries in each directory sector
- The first byte in a directory entry determines if the entry is either: (1) unused and to be skipped, (2) unused and marks the end of the directory, or (3) used
  - If DIR_Name[0] == DIR_ENTRY_UNUSED, then entry is unused and should be skipped
  - If DIR_Name[0] == DIR_ENTRY_LAST_AND_UNUSED, then entry is unused and the end of the directory has been reached
  - Otherwise, the entry is used

Short Filename Directory Entry

Short Filename

- Short filename entries can hold just an 8.3 name
  - The first short filename character DIR_Name[0] may not be a space (0x20)
  - There is an implied period (‘.’) between the main part and the extension except when the extension is all spaces
  - No lowercase characters may be in the short filename
  - The following characters may not appear in the short filename: lowercase characters, ‘””’ (double quote, 0x22), ‘*’ (asterisk, 0x2A) through ‘,’ (comma, 0x2C), ‘.’ (period, 0x2E) through ‘/’ (slash, 0x2F), ‘:’ (colon, 0x3A) through ‘?’ (question mark, 0x3F), ‘[’ (open bracket, 0x5B) through ‘]’ (close bracket, 0x5D), and ‘|’ (vertical bar, 0x7C)
    - In addition to the constraints above, in our implementation, we will also not allow any character with values less than 0x20 and we will also not allow any character with values greater than 0x7E
  - Terminate and fill both the main filename part and the extension name field with spaces (0x20)
  - If the extension field is all spaces, then the period (‘.’) separating the main filename part and the extension is not part of the filename

Short Filename Entries

- In addition to the 8.3 name, DIR_Name (11 bytes), there are many other fields
  - The integer number of the first cluster of the file, DIR_FstClusHI & DIR_FsrClusLO (4 bytes)
  - Attributes of the file, DIR_Attr (1 byte)
  - The size of the file in bytes, DIR_FileSize (4 bytes)
  - Creation date and time, DIR_CrtDate, DIR_CrtTime, and DIR_CrtTimeHundth (5 bytes)
    - Accurate to a hundredth of a second resolution
    - Optional
  - Last modification date and time, DIR_WrtDate & DIR_WrtTime (4 bytes)
    - Accurate to a two second resolution
    - Required
  - Last access date, DIR_LstAccDate (2 bytes)
    - Optional

Short Filename Entry Identification

- The first byte of the short filename, DIR_Name[0], if equal to:
  - DIR_ENTRY_UNUSED then the entry is free – the entry is to be skipped
  - DIR_ENTRY_LAST_AND_UNUSED then the entry is free and there are no active entries after this one
  - Otherwise, the entry is used and it is necessary to determine if the entry is for a short filename or for a long filename
- If the entry is used, then the attribute byte, DIR_Attr, is examined to determine if the entry is for a short filename or for a long filename
- If you are not handling long filename entries, then it is necessary to skip over long filename entries that you encounter

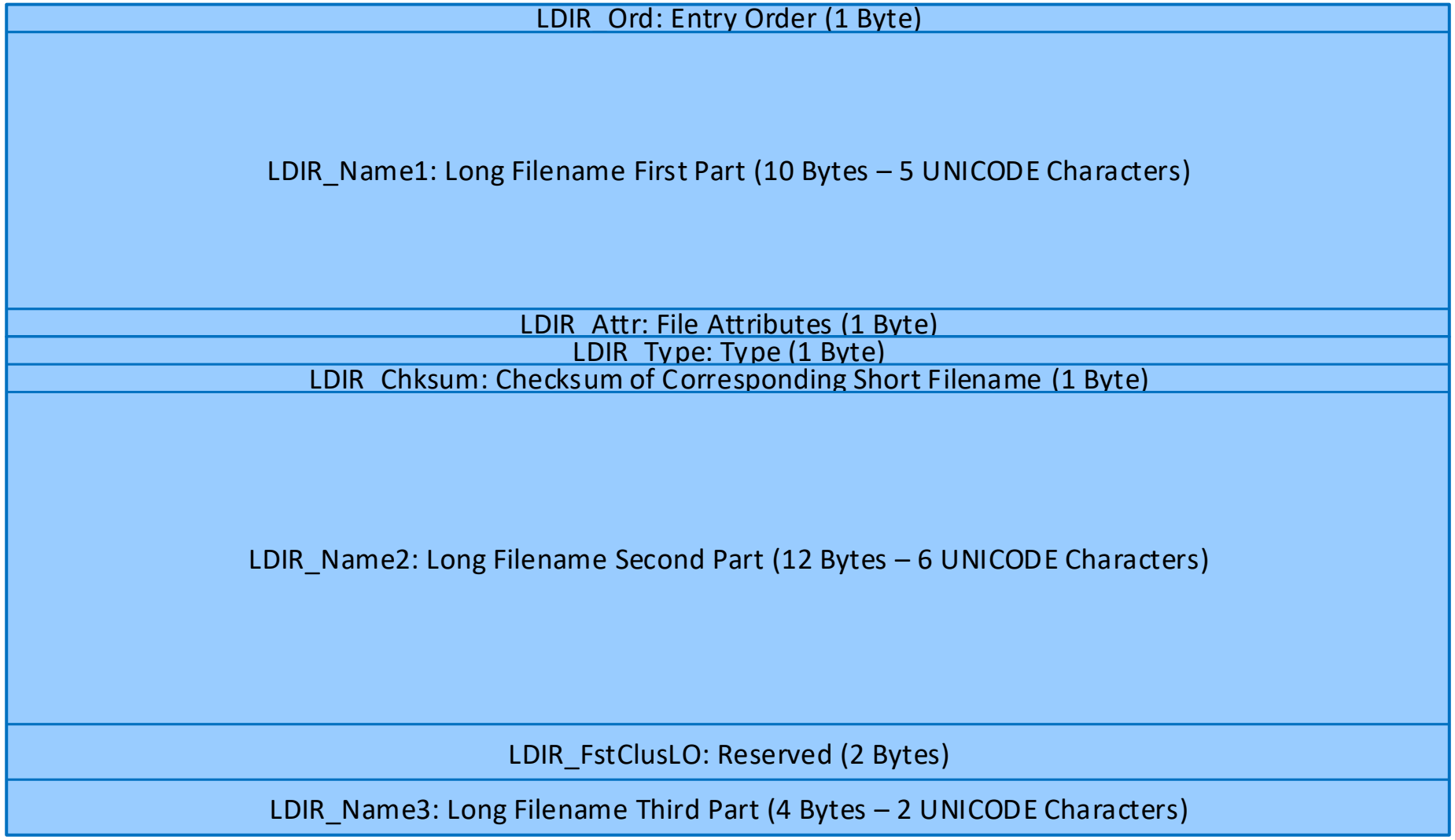
Short Filename Entry Attribute Byte

- The attributes of the file, DIR_Attr, byte consists of:
  - #define DIR_ENTRY_ATTR_READ_ONLY 0x01
  - #define DIR_ENTRY_ATTR_HIDDEN 0x02
  - #define DIR_ENTRY_ATTR_SYSTEM 0x04
  - #define DIR_ENTRY_ATTR_VOLUME_ID 0x08
    - DIR_ENTRY_ATTR_VOLUME_ID is set only in the root directory
  - #define DIR_ENTRY_ATTR_DIRECTORY 0x10
  - #define DIR_ENTRY_ATTR_ARCHIVE 0x20
- A short filename entry attribute cannot have the DIR_ENTRY_ATTR_READ_ONLY, DIR_ENTRY_ATTR_HIDDEN, DIR_ENTRY_ATTR_SYSTEM , and DIR_ENTRY_ATTR_VOLUME_ID bits all asserted; this configuration is reserved to identify a long filename entry and is referred to as DIR_ENTRY_ATTR_LONG_NAME
  - First apply the DIR_ENTRY_ATTR_LONG_NAME_MASK to DIR_Attr byte before comparing to DIR_ENTRY_ATTR_LONG_NAME
- Again, these #define'd constants are available in directory.h on the class web site

Short Filename Dates and Times

- Date Fields (2 byte field size)
  - Date is relative to the MS-DOS epoch of 1/1/1980
  - Bits 15-9: Count of years from 1980, valid value range 0-127 inclusive (represents 1980-2107)
  - Bits 8-5: Month of year, 1 = January, valid value range 1-12 inclusive
  - Bits 4-0: Day of month, valid value range 1-31 inclusive
- Time Fields (2 byte field size)
  - Bits 15-11: Hours, valid value range 0-23 inclusive
  - Bits 10-5: Minutes, valid value range 0-59 inclusive
  - Bits 4-0: 2-second count, valid value range 0–29 inclusive (0-58 seconds)
- TimeHundth Field (1 byte field size)
  - Hundredths of a second, valid value range 0-199 inclusive
  - Because the seconds portion of the Time field above denotes a time with a granularity of 2 seconds, this field may increment the number of seconds in addition to supplying the number of hundredths of a second

Long Filename Directory Entry



Long Filename Entries

- Each long filename entry can hold up to 13 characters of the filename
- A long filename may require multiple long filename entries
- Each long filename has an associated short filename entry and that short filename entry must be the last entry for that group
- All long filename entries and their associated short filename entry must be continuous directory entries, but not necessarily in the same sector or cluster
- The long filename entries do not have the data fields that are in the short filename entry; that data comes from the associated short filename entry
- The maximum length of a long filename is 255 characters (not including a terminating NUL character)

How to Generate a Short Filename from a Long Filename

- It's complicated and explained in the **Microsoft Extensible Firmware Initiative FAT32 File System Specification**, Version 1.03 on the class web site
  - See **The Basis-Name Generation Algorithm** on pages 30-31

Long Filename Entries

- A long filename entry contains the following fields
  - A portion of the long filename in three separate fields: LDIR_Name1 (10 bytes), LDIR_Name2 (12 bytes) & LDIR_Name3 (4 bytes) (total of 26 bytes)
    - Why so many bytes? Twice as many as we would have thought! They're 16-bit UNICODE chars.
    - Long filenames are terminated by a NUL character and they are padded with 0xFFFF half-words
    - An 8-bit ASCII character can be stored in the UNICODE character as the LSByte with the MSByte set to 0
  - Long filename attributes of the file, LDIR_Attr (1 byte)
    - This must be equal to DIR_ENTRY_ATTR_LONG_NAME
  - Long filename type, LDIR_Type (1 byte)
    - This must be equal to 0
  - Long filename artifact of low-half first cluster number, LDIR_FstClusLO (2 bytes)
    - This must be equal to 0
  - Long filename checksum, LDIR_Chksum (1 byte)
    - This must be equal to the checksum computed from the name in the associated short filename entry
  - Long filename order byte, LDIR_Ord (1 byte)
    - A value of 0xE5 indicates that the entry is unused
    - A value of 0 indicates that the end of the directory has been reached (that entry's contents is ignored)
- A struct containing these members is available in directory.h

Constraints Applied to Long Filenames

- Long filenames may contain upper and lower case ASCII characters, the space character (‘ ‘), the period character (‘.’) and the plus (‘+’), comma (‘,’), semicolon (‘;’), equal sign (‘=‘), open bracket (‘[‘), and close bracket (‘]’) characters
  - Embedded spaces are allowed, but leading and trailing spaces are ignored
  - Leading and embedded periods are allowed, but trailing periods are ignored
- No two filenames can have the same name, independent of case (*i.e.,* only one file with name a.B, A.b, etc. can exist)
  - This includes short and long filenames (*i.e.,* a short filename cannot be the same as a long filename, independent of case)

Algorithm to Compute a Short Filename Checksum

```
static uint8_t computeChecksum(uint8_t shortFilename[]) {
  uint8_t checksum;
  int i;


  checksum = 0;
  for(i = 0; i < 11; i++) {
    checksum = ((checksum & 1) ? 0x80 : 0) + (checksum >> 1) +
shortFilename[i];
  }
  return checksum;
}
```

Assembling a Long Filename

- Each long filename directory entry contains an assembly order byte, LDIR_Ord
  - If bit 6 in the LDIR_Ord byte is set (value of that bit is 0x40 – #define'd as LDIR_NAME_END), it indicates that the entry is for the characters at the end (*i.e.,* the rightmost characters) of the long filename
  - The first entry for a long filename must have bit 6 set in its LDIR_Ord byte
  - Bits 5-0 in the LDIR_Ord byte (we'll refer to this value as the ***order***) designate the position of this entry's filename in composing the complete long filename
    - The first segment of the long filename has its order set as 1
    - The second segment of the long filename has its order set as 2
    - And so forth

Creating a New Entry in a Directory

- Read through the directory to ensure that no entry already exists with the new filename
- Find either an unused directory entry (designated by DIR_ENTRY_UNUSED) or, if no unused entry is found, extend the directory by adding an entry to the end
  - Extending a directory may be as simple as using the entry tagged as DIR_ENTRY_LAST_AND_UNUSED for the new entry and tagging the next entry as DIR_ENTRY_LAST_AND_UNUSED
  - But, if DIR_ENTRY_LAST_AND_UNUSED tags the last entry in a sector, then extending the directory may require writing DIR_ENTRY_LAST_AND_UNUSED into the first directory entry in the next sector
    - However, if there are no more sectors in the current cluster, then the FAT needs to be searched to find a free cluster (FAT_ENTRY_FREE) and the FAT has to be updated to indicate that the formerly free cluster will be linked after the last cluster in the directory
      - In addition, remember to indicate in the FAT that the formerly free cluster is now FAT_ENTRY_ALLOCATED_AND_END_OF_FILE
    - Then, the first directory entry in the first sector of the newly allocated cluster would need to be tagged as DIR_ENTRY_LAST_AND_UNUSED

Creating a Directory Entry for a New Directory (1 of 2)

- A directory entry for a new directory must conform to the following
  - DIR_Attr must have the DIR_ENTRY_ATTR_DIRECTORY bit set
  - DIR_FileSize must be set to 0
  - At least one cluster must be allocated for the directory and DIR_FstClusHI & DIR_FstClusLO must contain the first cluster number allocated
    - We want you to allocate one cluster
    - The FAT entry for that cluster must indicate FAT_ENTRY_ALLOCATED_AND_END_OF_FILE
    - The contents of the entire cluster must be initialized to 0
  - All directories except for the root directory must have the following two entries at the beginning of the directory:
    - The first entry must have the name "." (*i.e.,* a single dot/period character)
    - The second entry must have the name ".." (*i.e.,* two dot/period characters)

Creating a Directory Entry for a New Directory (2 of 2)

- Both “.” and “..” are directories and therefore should both be tagged with attribute DIR_ENTRY_ATTR_DIRECTORY and with a file size of 0
- The “.” directory entry refers to the current directory
  - Therefore, the DIR_FstClusHI & DIR_FstClusLO fields must contain the cluster number of first cluster in the current directory
- The “..” directory entry refers to the parent directory
  - Therefore, the DIR_FstClusHI & DIR_FstClusLO fields must contain the cluster number of first cluster in the parent directory
    - If the parent directory is the root directory, then the DIR_FstClusHI & DIR_FstClusLO fields must be set to 0
    - NOTE: When setting the CWD (Current Working Directory) to cluster number 0 from a “..” directory entry, the actual cluster number will come from the boot sector (available as variable “root_directory_cluster”)

Creating a Directory Entry for a New Regular (Non-Directory) File

- A new regular (non-directory) file must conform to the following
  - DIR_Attr must have the DIR_ENTRY_ATTR_DIRECTORY bit clear
  - DIR_FileSize must be set to 0
  - A zero length file must have no clusters allocated for it and, therefore, the DIR_FstClusHI & DIR_FstClusLO fields must be set to 0

- When the first bytes are written to the file,
  - An initial cluster needs to be allocated
  - That cluster's number needs to be written into the appropriate directory entry's DIR_FstClusHI & DIR_FstClusLO fields
  - The FAT entry for that cluster must indicate FAT_ENTRY_ALLOCATED_AND_END_OF_FILE
  - The bytes written must eventually be written to sectors in the file
  - The DIR_FileSize must be appropriately updated to contain the total file size

Extending a New Regular (Non-Directory) File

- When appending bytes to a file,
  - The bytes written must eventually be written to sectors in the file
  - The DIR_FileSize in the file's directory entry must eventually be appropriately updated to contain the total file size
  - If a sector becomes full and if there is an additional following sector in the current cluster, then writing continues to the beginning of that following sector in the current cluster
  - If the last sector in the last cluster in the file becomes full, an additional cluster needs to be allocated (by searching in the FAT for a free cluster) and writing continues to the first sector in that newly allocated cluster
    - The newly allocated cluster's number needs to be written into the FAT entry of its previous cluster – this performs the cluster linking that is maintained by the FAT
    - And, as required, the FAT entry for the newly allocated cluster must be changed from being free (FAT_ENTRY_FREE) to indicate it is FAT_ENTRY_ALLOCATED_AND_END_OF_FILE

Additional Directory Validation Steps

- The **Microsoft Extensible Firmware Initiative FAT32 File System Specification** has some additional guidelines in the section entitled Validating The Contents of a Directory on pages 32-33
  - Reserved fields in a directory entry have *no* required value
  - The values in reserved fields in a directory *should not be reset to 0* (or any other value) for upward (future) compatibility
  - To determine the kind of short filename entry, look at the DIR_ENTRY_ATTR_VOLUME_ID and DIR_ENTRY_ATTR_DIRECTORY bits
    - If both bits are zero, then the file is a *regular file*
    - If either bit is set (but *not* both), then the file is a *directory*
    - If both bits are set, then the entry is *invalid*
  - The DIR_NTRes/LDIR_Type byte (offset 12) has *no* required value

Additional Notes for FAT Directories

- The **Microsoft Extensible Firmware Initiative FAT32 File System Specification** has some additional guidelines in the section entitled Other Notes Relating to FAT Directories on pages 33-34
  - The maximum length for any regular file is 4,294,967,295 bytes – this is the largest unsigned integer that can be represented in the 32-bit file size in a directory entry
  - The maximum *number* of directory *entries* that can exist in any one directory is 65,536
    - This allows a 16-bit unsigned integer to be used to identify a directory entry within a directory
      - This implies that the maximum size of a directory is 65,536 * 32 bytes
    - Of course, there needs to be an entry used for DIR_ENTRY_LAST_AND_UNUSED

NXP K70 ARM microSDHC

- The K70F120M chip includes a controller to assist in accessing the microSDHC
- In software, we are providing functions to you to initialize the microSDHC, access the FAT, translate a cluster number into the first sector in that cluster, read and write sectors
- You need to write all of the software to read and write files and directories, create and delete files

Simplifications

- Your Problem Set 3 solution is simplified in a number of ways
  - You don't need to deal with the endian issue because both our computer and the FAT32 use little endian byte ordering
  - You are allowed to use the GNU C "packed" extension for structs, but there is probably no reason to do so
  - You must use an external program/computer to format your microSDHC
    - That is, we will not be implementing a format command
  - You do not need to support sub-directories, but this is optional for extra credit
    - This means that an acceptable solution can manipulate files in just the root directory
  - You do not need to support long filenames, but this is optional for extra credit
    - This means that all filenames must be in 8.3 format
    - Limitations on short filenames: Only uppercase characters are stored in the directory entry
  - You should use my FAT manipulation package as shown on a later slide
    - My code maintains consistent FAT data and writes the FAT to both FATs in the file structure whenever a FAT entry is modified
    - Remember to my call invalidate_entire_FAT_cache function before allowing the microSDHC card to be removed from the K70

Student Implemented File Structure Function Interfaces

- All functions return an int which indicates success if 0 and an error code otherwise (only some errors are listed)

- All student implemented functions are declared in SDHC_FAT32_Files.h on the class web site
- directory.h on the class web site is a header file with declarations of struct's and #define's for directories

int file_structure_mount(void);

  - Indicates that the microSDHC card is to be made available for these function calls
  - Actions this function should take: call microSDCardDetectConfig; check to see if a microSDHC card is present; if not present, return an error; otherwise, call microSDCardDisableCardDetectARMPullDownResistor to disable the pull-down resistor in the K70 ARM and then call sdhc_initialize, set the cwd to the root directory
  - Returns an error code if the file structure is already mounted

int file_structure_umount(void);

  - Indicates that the microSDHC card is no longer accessible
  - Action this function should take: ensure that all modified sectors in the student-implemented disk cache have been written to the disk, call invalidate_entire_FAT_cache to ensure that all modified sectors in the supplied FAT cache have been written to the disk, call sdhc_command_send_set_clr_card_detect_connect to re-enable the card-present-resistor
  - Returns an error code if the file structure is not mounted

Student Implemented Directory Function Interfaces

- cwd is the current working directory
- All functions return an int which indicates success if 0 and an error code otherwise (only some errors are listed)

int dir_set_cwd_to_root(void);

- Sets the cwd to the root directory
- This is the initial action before the FAT32 file structure is made available through these interfaces

int dir_ls(int full);

- Display on stdout the cwd's filenames (**full** == 0) or all directory information (**full** == 1); implementing **full** is optional

int dir_ls_init(void **statepp);

- Start an iterator at the beginning of the cwd's filenames
- Returns in ***statepp** a pointer to a malloc'ed struct that contains necessary state for the iterator
- Optional; must be implemented with dir_ls_next
- dir_ls_init and dir_ls_next are intended to enable filename completion at the shell command line when a tab character is entered

int dir_ls_next(void *statep, char **filenamep);

- Uses **statep** as a pointer to a struct malloc'ed and initialized **by dir_ls_init** that contains iterator state
- Returns in ***filenamep** the malloc'ed name of the next filename in the cwd; Caller is responsible to free filename's memory
- Returns NULL for ***filenamep** if at end of the directory or if the malloc was not successful; If returning NULL, the malloc'ed struct pointed to by **statep** is free'd
- Optional; must be implemented with dir_ls_init

int dir_find_file(const char *filename, uint32_t *firstCluster);

- Search for **filename** in the cwd and return its first cluster number in **firstCluster**
- Returns an error code if filename is not in the cwd
- Returns an error code if the filename is a directory

int dir_set_cwd_to_filename(const char *filename);

- Search for **filename** in cwd and, if it is a directory, set the cwd to that filename
- Returns an error code if filename is not in the cwd
- Returns an error code if the filename is not a directory
- Implementing this function is optional

int dir_create_file(const char *filename);

- Create a new empty regular file in the cwd with name **filename**
- Returns an error code if a regular file or directory already exists with this filename
- Returns an error code if there is no more disk space to create the regular file

int dir_delete_file(const char *filename);

- Delete a regular file in the cwd with name **filename**
- Returns an error code if a file with this name does not exist
- Returns an error code if a file with this filename is currently open
- Returns an error code if the file with this name is a directory

int dir_create_dir(const char *filename);

- Create a new empty directory in the cwd with name **filename**
- Returns an error code if there is no more disk space to create the directory
- Implementing this function is optional

int dir_delete_dir(const char *filename);

- Delete a directory in the cwd with name **filename**
- Returns an error code if a file with this name does not exist
- Returns an error code if a directory with this filename contains any files or directories
- Returns an error code if the file with this name is not a directory
- Implementing this function is optional

Student Implemented File Function Interfaces

- All functions return an int which indicates success if 0 and an error code otherwise (only some errors are listed)

typedef uint32_t file_descriptor;

- A **file_descriptor** is used as an index into an array of structures describing open files
- All entries in the array are initially closed and not associated with any open files

int file_open(const char *filename, file_descriptor *descrp);

- Search for **filename** in the cwd and, if successful, store a file descriptor for that file into ***descrp**
- Returns an error code if filename is not in the cwd
- Returns an error code if the filename is not a regular file

int file_close(file_descriptor descr);

- Close the file associated with **descr** and disassociate **descr** from that file
- Returns an error code if the file descriptor is not open
- Frees all dynamic storage associated with the formerly open file descriptor and indicates that the descriptor is closed

int file_getbuf(file_descriptor descr, char *bufp, int buflen, int *charsreadp);

- Read sequential characters at the current offset from the file associated with **descr** into the buffer pointed to by **bufp** for a maximum length of **buflen** characters; The actual number of characters read is returned ***charsreadp**
- Returns an error code if the file descriptor is not open
- Returns an error code if there are no more characters to be read from the file (EOF, this is, End Of File)

int file_putbuf(file_descriptor descr, const char*bufp, int buflen);

- Write characters at the current offset into the file associated with **descr** from the buffer pointed to by **bufp** with a length of **buflen** characters
- Returns an error code if the file descriptor is not open
- Returns an error code if the file is not writeable (See DIR_ENTRY_ATTR_READ_ONLY), optional to implement read-only
- Returns an error code if there is insufficient disk space in which to write the characters

Supplied .h & .c Files

- Students are supplied with the following .h & .c files that should be made part of your project
  - SDCardReader.h
    - #define's SDHC_USE_UART, if UART is to be used for debugging output; otherwise, console I/O is used
  - breakpoint.h
    - #define's __BKPT() to cause the debugger to be invoked
  - uart.h & uart.c
    - Low-level UART (serial) devices
  - uartNL.h & uartNL.c
    - UART extensions to uart.h & uart.c
  - microSD.h & microSD.c
    - Low-level I/O to the microSD card
  - bootSector.h & bootSector.c
    - Reads and parses the MBR and boot sector
  - fsInfo.h & fsInfo.c
    - Reads and parses the FSInfo sector
  - FAT.h & FAT.c
    - Reads and writes the FAT
  - SDHC_FAT32_Files.h
    - Declarations for student-implemented functions
  - directory.h
    - #define's & struct's to be used for access to directory entry fields

Supplied microSDHC Interfaces/structs from microSD.h/.c (1 of 2)

- Routine to configure the microSD Card Detect switch & pull-down resistor in the ARM; This must be called before calling microSDCardDetectedUsingSwitch or microSDCardDetectedUsingResistor

  void microSDCardDetectConfig(void);

- Routine to read the state of the microSD Card Detect switch; Returns true if the card is detected and false otherwise

  int microSDCardDetectedUsingSwitch(void);

- Routine to read the state of the microSD Card internal resistor; The internal resistor is enabled when a card is inserted into the microSD slot, when new code is loaded into the K70, and when explicitly enabled by calling sdhc_command_send_set_clr_card_detect_connect; Returns true if the card is detected and false otherwise

  Important: Because detecting the presence of a microSD card is unreliable once a card has been inserted, mounted, unmounted, removed, and then reinserted, please do *not* rely on card presence detection using the resistor – **just use the switch to detect the card**

  int microSDCardDetectedUsingResistor(void);

- Routine to disable the microSD Card Detect ARM internal pull-down resistor; This must be called after conducting SD card detection using the SD card internal resistor and before initializing the SD card interface

  void microSDCardDisableCardDetectARMPullDownResistor(void);

- Initializes the SDHC interface; A microSDHC card must be present in the TWR-K70F120M microSD card slot before calling this function; This must be called before calling any of the following functions; This function also disconnects the 50k Ohm pull-up resistor inside the SD card; Returns the rca (Relative Card Address); After calling sdhc_initialize, all variables declared in the furnished header files are appropriately initialized

  uint32_t sdhc_initialize(void);

- enum sdhc_status;
  struct sdhc_card_status;

- enum sdhc_status sdhc_read_single_block(
    uint32_t rca,
    uint32_t block_address,
    struct sdhc_card_status *card_status,
    uint8_t data[512]);

- enum sdhc_status sdhc_write_single_block(
    uint32_t rca,
    uint32_t block_address,
    struct sdhc_card_status *card_status,
    const uint8_t data[512]);

- Routine to connect the 50k Ohm (nominal value, specified range is 10k Ohm to 90k Ohm) pull-up resistor inside the SD card; This should be called when unmounting (*i.e.,* finished using) the SD card; The function can be invoked only after calling sdhc_initialize

  enum sdhc_status sdhc_command_send_set_clr_card_detect_connect(
    uint32_t rca);

Supplied microSDHC Interfaces/structs from microSD.h/.c (2 of 2)

- Comments on **enum sdhc_status** and **struct sdhc_card_status**
- When calling functions that return an **enum sdhc_status**, such as **sdhc_read_single_block**, **sdhc_write_single_block**, and **sdhc_command_send_set_clr_card_detect_connect**, you must ensure that the return value is equal to **SDHC_SUCCESS**
  - The return value should always be **SDHC_SUCCESS**, therefore you are welcome to call **__BKPT()** if you see any other return value
  - **__BKPT()** is declared in **breakpoint.h** and causes the caller to end up in the debugger
- Functions that accept a **struct sdhc_card_status *** want a preallocated struct to be passed to them so that a detailed card status can be returned; an example follows:

```
struct sdhc_card_status card_status;
uint8_t sector[512];
if((sdhc_read_single_block(rca, sectorNumber, &card_status, sector) != SDHC_SUCCESS) {
  __BKPT();
}
```

Supplied Boot Sector Data/Interfaces from bootSector.h/.c

- Number of bytes per sector

  uint16_t bytes_per_sector;
- Number of sectors per cluster

  uint32_t sectors_per_cluster;
- Cluster number of the root directory

  uint32_t root_directory_cluster;
- Total number of data clusters; Remember that the number of the first data cluster is 2

  uint32_t total_data_clusters;

- Returns the number of the first sector in cluster **cluster**

  uint32_t first_sector_of_cluster(uint32_t cluster);

Supplied FSInfo Sector Data from fsInfo.h

- Value for FSI_Nxt_Free that indicates it is unknown

  #define FSI_NXT_FREE_UNKNOWN 0xffffffff

- A hint of the starting cluster number from which to search for a free cluster

  uint32_t FSI_Nxt_Free;

  - If it is equal to FSI_NXT_FREE_UNKNOWN, then this value must not be used and the search for a free cluster must start with cluster 2 (root_directory_cluster)
  - This value is furnished to reduce the time to find a free cluster
  - Hint: Even if FSI_Nxt_Free is not equal to FSI_NXT_FREE_UNKNOWN and if a search for a free cluster starting at cluster FSI_Nxt_Free is unsuccessful, it is still possible that there is a free cluster before cluster FSI_Nxt_Free

Supplied FAT Region Data from FAT.h

- Mask for the low-order 28 bits of a FAT entry; these are the bits that are meaningful

  #define FAT_ENTRY_MASK 0x0fffffff

- Special FAT entries
  - The entry is not used (*i.e.,* free/available):

    #define FAT_ENTRY_FREE 0
  - The entry is defective and should not be used:

    #define FAT_ENTRY_DEFECTIVE_CLUSTER 0x0ffffff7
  - When *reading* a FAT entry, end of a cluster linked list is any entry greater-than or equal to FAT_ENTRY_RESERVED_TO_END

    #define FAT_ENTRY_RESERVED_TO_END 0x0ffffff8
  - When *writing* a FAT entry, FAT_ENTRY_ALLOCATED_AND_END_OF_FILE should be used for the last allocated cluster in a directory or file

    #define FAT_ENTRY_ALLOCATED_AND_END_OF_FILE 0x0fffffff

- Returns the FAT entry for cluster **cluster**
  rca is the Relative Card Address returned from **sdhc_initialize**
  Any necessary I/O is performed by this function

  uint32_t read_FAT_entry(uint32_t rca, uint32_t cluster);

- Updates (i.e., writes) the FAT entry for cluster **cluster**
  rca is the Relative Card Address returned from **sdhc_initialize**
  nextCluster is the value to be written to the FAT entry for cluster **cluster**
  Any necessary I/O is performed by this function

  void write_FAT_entry(uint32_t rca, uint32_t cluster, uint32_t nextCluster);

- Indicate that the entire FAT cache should be invalidated
  This function should be called whenever a microSD card is unmounted so that if a new card is later mounted, the previous contents of the FAT cache is not used

  Note that the FAT cache is implemented as a single-sector write-through cache to ensure that the primary and backup FATs on the microSD card are always kept up-to-date whenever a FAT entry is updated using the write_FAT_entry function

  void invalidate_entire_FAT_cache(void);

Supplied Directory Data/structs from directory.h (1 of 3)

- struct for a short directory entry

```
struct dir_entry_8_3 {
    uint8_t DIR_Name[11];              /* Offset 0 */
    uint8_t DIR_Attr;                  /* Offset 11 */
    uint8_t DIR_NTRes;                 /* Offset 12 */
    uint8_t DIR_CrtTimeHundth;         /* Offset 13 */
    uint16_t DIR_CrtTime;              /* Offset 14 */
    uint16_t DIR_CrtDate;              /* Offset 16 */
    uint16_t DIR_LstAccDate;           /* Offset 18 */
    uint16_t DIR_FstClusHI;            /* Offset 20 */
    uint16_t DIR_WrtTime;              /* Offset 22 */
    uint16_t DIR_WrtDate;              /* Offset 24 */
    uint16_t DIR_FstClusLO;            /* Offset 26 */
    uint32_t DIR_FileSize;             /* Offset 28 */
  };
```

- The MS-DOS Epoch year for FAT32 directory dates (Midnight between December 31st, 1979 and January 1st, 1980)

```
#define MS_DOS_EPOCH 1980
```

Supplied Directory Data/structs from directory.h (2 of 3)

- struct for a long directory entry

```
struct dir_entry_long {
    uint8_t LDIR_Ord;                   /* Offset 0 */
    uint16_t LDIR_Name1[5];             /* Offset 1 */
    uint8_t LDIR_Attr;                  /* Offset 11 */
    uint8_t LDIR_Type;                  /* Offset 12 */
    uint8_t LDIR_Chksum;                /* Offset 13 */
    uint16_t LDIR_Name2[6];             /* Offset 14 */
    uint16_t LDIR_FstClusLO;            /* Offset 26 */
    uint16_t LDIR_Name3[2];             /* Offset 28 */
  };
```

- Number of UNICODE characters in a long directory entry

```
#define DIR_ENTRY_LONG_FILE_NAME_CHARS_PER_ENTRY 13
```

- Bit value that indicates a long directory's entry name is the rightmost (end) portion of that long name

```
#define LDIR_ORD_NAME_END_BIT 6
#define LDIR_ORD_NAME_END_MASK (1<<LDIR_ORD_NAME_END_BIT)
```

- Mask for the order number (1-origin) of a long directory's entry name

```
#define LDIR_ORD_ORDER_MASK 0x3F
```

Supplied Directory Data/structs from directory.h (3 of 3)

- #define's for first byte (DIR_Name[0]/LDIR_Ord) in a directory entry

```
#define DIR_ENTRY_LAST_AND_UNUSED 0x0
#define DIR_ENTRY_UNUSED 0xE5
```

- #define's for the attribute byte (DIR_Attr/LDIR_Attr)

```
#define DIR_ENTRY_ATTR_READ_ONLY_BIT 0
#define DIR_ENTRY_ATTR_READ_ONLY_MASK (1<<DIR_ENTRY_ATTR_READ_ONLY_BIT)
#define DIR_ENTRY_ATTR_HIDDEN_BIT 1
#define DIR_ENTRY_ATTR_HIDDEN_MASK (1<<DIR_ENTRY_ATTR_HIDDEN_BIT)
#define DIR_ENTRY_ATTR_SYSTEM_BIT 2
#define DIR_ENTRY_ATTR_SYSTEM_MASK (1<<DIR_ENTRY_ATTR_SYSTEM_BIT)
#define DIR_ENTRY_ATTR_VOLUME_ID_BIT 3
#define DIR_ENTRY_ATTR_VOLUME_ID_MASK (1<<DIR_ENTRY_ATTR_VOLUME_ID_BIT)
#define DIR_ENTRY_ATTR_DIRECTORY_BIT 4
#define DIR_ENTRY_ATTR_DIRECTORY_MASK (1<<DIR_ENTRY_ATTR_DIRECTORY_BIT)
#define DIR_ENTRY_ATTR_ARCHIVE_BIT 5
#define DIR_ENTRY_ATTR_ARCHIVE_MASK (1<<DIR_ENTRY_ATTR_ARCHIVE_BIT)
#define DIR_ENTRY_ATTR_LONG_NAME (DIR_ENTRY_ATTR_READ_ONLY_MASK |          \
DIR_ENTRY_ATTR_HIDDEN_MASK | \
DIR_ENTRY_ATTR_SYSTEM_MASK | \
DIR_ENTRY_ATTR_VOLUME_ID_MASK)
#define DIR_ENTRY_ATTR_LONG_NAME_MASK (DIR_ENTRY_ATTR_READ_ONLY_MASK | \
DIR_ENTRY_ATTR_HIDDEN_MASK | \
DIR_ENTRY_ATTR_SYSTEM_MASK | \
DIR_ENTRY_ATTR_VOLUME_ID_MASK | \
DIR_ENTRY_ATTR_DIRECTORY_MASK | \
DIR_ENTRY_ATTR_ARCHIVE_MASK)
```