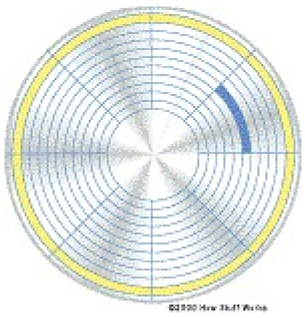
The MS- DOS fat-12 file system

1. Clusters vs. sectors

A sector is a fixed division of a track on a disk. In the image,



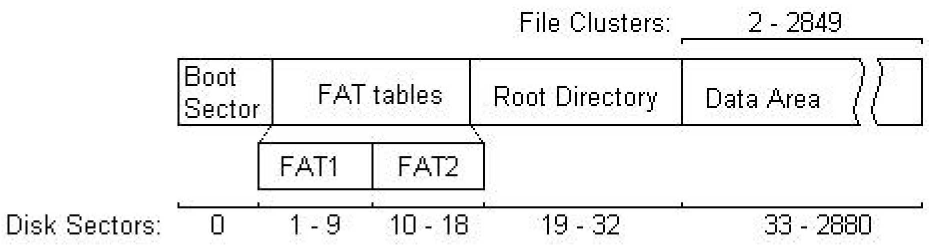
The yellow circle represents a single track, and the blue segment is a single sector. Sectors are fixed-size (typically 512 bytes on most media) and are the basic unit of access for any disk (see "Reading and Writing to the floppy disk or a disk image").

A cluster is one or more contiguous sectors and the fundamental unit for DOS files. The DOS file system only works in terms of clusters. The FAT tables index used and unused clusters. In this assignment we assume 1 sectors per cluster.

The distinction between FAT-12, FAT-16, and FAT-32 is merely the size of one entry in the FAT table. FAT-12 systems contain 12 bits per entry, while FAT-32 systems have 32 bits per entry. The FAT type is solely determined by the number of clusters on the media.

2. DOS disk layout

An MS-DOS floppy disk layout (FAT-12) consists of four major sections: the boot sector, FAT tables, root directory, and data area:



- **The boot sector** consists of the first sector (sector 0) on the volume or disk The boot sector contains specific information about the rest of organization of the file system, including how many copies of the FAT tables are present, how big a sector is, how many sectors in a cluster, etc. Below is a detailed description of how this specific information is saved in the boot sector of FAT file system.
- **FAT tables** are pointers to every cluster on the disk, and indicate the number of the next cluster in the current cluster chain, the end of the cluster chain, whether a cluster is empty, or has errors. The FAT tables are the only method of finding the location of files and directories on the rest of the disk (thus, if the FAT tables become corrupted, the data on the disk is also essentially lost). There are typically two redundant copies of the FAT table on disk for data security and recovery purposes. Below is a detailed description of FAT-12 file tables.
- **The Root Directory** is the primary directory of the disk. Unlike other directories located in the data area of the disk, the root directory has a finite size (14 sectors * 16 directory entries per sector = 224 possible entries, see "Directory structures and their fields" for more details) restricting the total amount of files or directories that can be created therein.
- **Data Area.** The first sector of the data area (sector 33) corresponds to cluster 2 of the file system (the first cluster is *always* cluster 2). The data area contains file and directory data and spans the remaining sectors on the disk.

3. The boot sector

The first sector on the volume or disk is the boot sector. The boot sector contains information about the rest of the file system structure. You may assume that all disks and image files in this assignment will conform to the FAT-12 standard. In the instance that you needed to determine for yourself what type of FAT file system you were mounting, and its physical layout on the disk, you would need to read and configure your program with the values shown in the boot sector. The boot sector contains the following values, which are aligned exactly as:

```
typedef struct {
    uint8_t     bootjmp[3];  /* 0  Jump to boot code */
    uint8_t     oem_id[8];   /* 3  OEM name & version */
    uint16_t    sector_size; /* 11 Bytes per sector hopefully 512 */
    uint8_t     sectors_per_cluster;    /* 13 Cluster size in sectors */
    uint16_t    reserved_sector_count;  /* 14 Number of reserved (boot) sectors */
    uint8_t     number_of_fats;         /* 16 Number of FAT tables hopefully 2 */
    uint16_t    number_of_dirents;      /* 17 Number of directory slots */
    uint16_t    sector_count;           /* 19 Total sectors on disk */
    uint8_t     media_type;             /* 21 Media descriptor=first byte of FAT */
    uint16_t    fat_size_sectors;       /* 22 Sectors in FAT */
    uint16_t    sectors_per_track;      /* 24 Sectors/track */
    uint16_t    nheads;                 /* 26 Heads */
    uint32_t    sectors_hidden;         /* 28 number of hidden sectors */
    uint32_t    sector_count_large;     /* 32 big total sectors */

} __attribute__ ((packed)) boot_record_t;
```

If you are interesting in a more detailed explanation about each of the fields, and what they mean, see the file "fat_paper.pfd"

For information on reading the boot sector into a structure similar to this one, see "Mapping disk data directly to structures").

4. Mapping disk data directly to structures

This exercise treats disk image file as a devise. That is, you are about to open the file of disk image and then read or write its contents as full (512-byte) disk sectors. Use the following functions to read and write the disk.

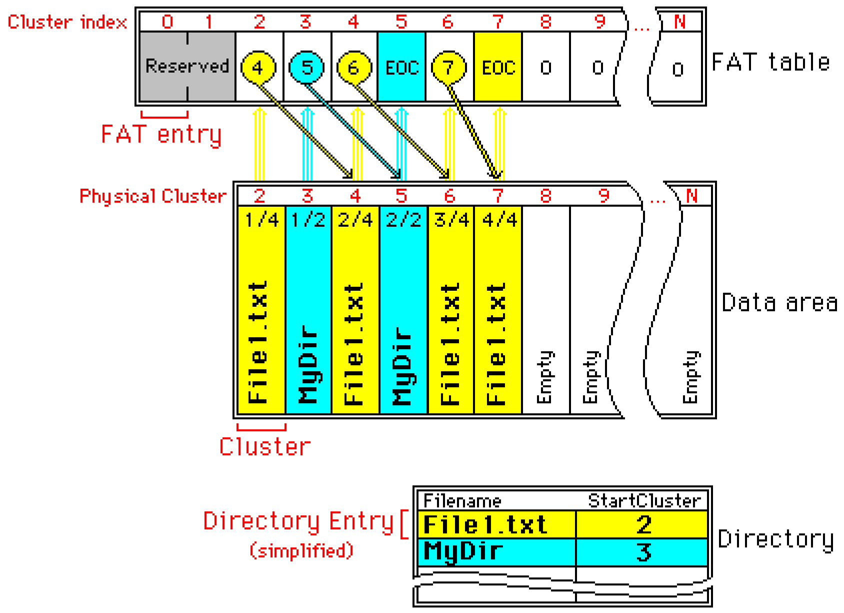
```
#define DEFAULT_SECTOR_SIZE          512
int fid; /* global variable set by the open() function */

int fd_read(int sector_number, char *buffer){
        int dest, len;
        dest = lseek(fid, sector_number * DEFAULT_SECTOR_SIZE, SEEK_SET);
        if (dest != sector_number * bps){
              /* Error handling */
        }
        len = read(fid, buffer, bps);
        if (len != bps){
                /* error handling here */
        }
        return len;
}

int fd_write(int sector_number, char *buffer){
        int dest, len;
        dest = lseek(fid, sector_number * DEFAULT_SECTOR_SIZE, SEEK_SET);
        if (dest != sector_number * bps){
              /* Error handling */
        }
        len = write(fid, buffer, bps);
        if (len != bps){
                /* error handling here */
        }
        return len;
}
```

Then copy from buffer to desired structure. See, for example, how to read in to the boot structure:

```
boot_record_t bs;
memcpy(bs, buffer, sizeof(boot_record_t);
```

5. Organization of the file allocation tables (FAT)

The FAT-12 tables are organized as a linear array of FAT entries. A FAT entry is a single *12 bit value* whose index matches the associated cluster in the data area of the disk. Because the first cluster in the data area of the disk corresponds to cluster 2, the *first two* FAT entries in the FAT table are reserved. You can think of the FAT table as a large array that contains pointers into the data area of the disk. It is also a linked-list of sorts, because you retrieve files that are longer than one cluster by referencing the FAT table, which points you to the next cluster of that file or directory.



A *12-bit* FAT entry may have the following values

- **0x000** Unused cluster.
- **0xFF0-0xFF6** Reserved cluster (like FAT entries 0 and 1).
- **0xFF7** Bad Cluster (contains errors and should not be used).
- **0xFF8-0xFFF** End of file/directory, also called EOC (end of cluster chain).
- **other numbers** are pointers (indexes) to the next cluster in the file/directory.

Cluster chains are a series of FAT entries that point to the next cluster in the file/directory, and are terminated by an EOC indicator--exactly like linked-lists. For example, in the graphic above, the cluster chain for **File1.txt** is retrieved by starting at the first cluster indicated in the directory for that file (the bottom box in the above graphic) and following the pointers in the FAT table until you encounter the EOC terminator. The chain would comprise clusters 2, 4, 6, and 7. The cluster chain for **MyDir** is 3, 5. You know that you've reached the end of a file or directory listing when you check the corresponding FAT entry for the current cluster, and find that it contains the EOC flag.

There are 3072 FAT entries in each FAT table (512 bytes per sector * 9 sectors = 4608 bytes. 4608 bytes / 1.5 bytes per FAT entry = 3072 FAT entries). Be aware that this is a few more entries than the disk has clusters; therefore, maximum capacity of the disk should be determined by the total number of clusters, not the number of possible FAT table entries.

Retrieving a single FAT entry, is slightly complicated because we can't easily create a data structure of 12-bit or 1.5 byte entries. You may store the FAT table any way you wish, but the following code sample assumes that you've chosen to store the FAT table in an array of unsigned chars. Because we chose this, we will need to grab an entire 2 bytes that cover the 12-bit FAT entry, with four bits to spare, and then extract just the 12 bits that we want (either the high-order or low-order bits, depending on whether the original FAT index was even or odd). The function for retrieving a FAT entry is provided below. In the following function, the variable FAT is the array of unsigned chars containing the FAT table (but check this function before you use it!).

GetFatEntry Takes an index into the FAT table, and returns a value contained in the 12-bit FAT entry.

```
#define _WORD(x)   ((unsigned char)(x)[0] + (((unsigned char)(x)[1]) << 8))

int GetFatEntry(int FATindex){
   unsigned int FATEntryCode; // The return value
   //   Calculate the offset of the WORD to get
   int FatOffset = ((FATindex * 3) / 2);
   if (FATindex % 2 == 1){   // If the index is odd

      FATEntryCode =  _WORD(&FAT[FatOffset]);
      FATEntryCode >>= 4; // Extract the high-order 12 bits
   }
   else{    // If the index is even

      FATEntryCode = _WORD(&FAT[FatOffset);
       FATEntryCode &= 0x0fff; // Extract the low-order 12 bits
   }
   return FATEntryCode;
}
```

6. FAT-12 file name and extension representation

File names in DOS traditionally have a limit of 8 characters for the name, and 3 characters for the extension (modern FAT file systems allow for longer file names). In your assignment, you will need to translate the file names that you are given, into a form that you can use to search directory entries (see "Directory structures and their fields"). There are a few things to be aware of:

- File/directory names and extensions are *not* null-terminated within the directory entry
- File/directory names always occupy 8 bytes--if the file/directory name is shorter than 8 bytes (characters) pad the remaining bytes with spaces (ASCII 32, or Hex 0x20). This also applies to 3-character extensions.
- File/directory names and extensions are *always* uppercase. Always convert given file/directory names to uppercase.
- Directory names can have extensions too.
- "FILE1" and "FILE1.TXT" are unique (the extension *does* matter).
- Do not accept file/directory names longer than 8 characters + 3 character extension.

Here are examples of how some file names would translate into the 11 bytes allocated for the file/directory name and extension in the directory entry (white space between quotes should be considered as spaces).

- `filename provided [01234567012]`
- `"foo.bar"      -> "FOO     BAR"`
- `"FOO.BAR"      -> "FOO     BAR"`
- `"Foo.Bar"      -> "FOO     BAR"`
- `"foo"          -> "FOO        "`
- `"foo."         -> "FOO        "`
- `"PICKLE.A"     -> "PICKLE  A  "`
- `"prettybg.big" -> "PRETTYBGBIG"`
- `".big"         -> *illegal*! file/directory names cannot begin with a "."`

7. Directory structures and their fields

A directory entry contains all of the information necessary to reference the contents of a file or directory on the volume, including additional information about the file/directory's attributes and other information including: time of creation/modification, date created, and size of the file (in bytes). All of this information is packed into a small 32-byte structure.

Directories in the data area of the disk can be of any length (number of sectors). However, because the root directory is of fixed size, only a set number of directory entries will fit.

Subdirectories--directories other than the root directory-- contain two additional directory entries by default. These are the dot [.] and dotdot [..] entries--you have probably seen these before. These entries are simply pointers, the first [dot] points to the current directory, kind of like the "this" pointer in C++. The second entry [dotdot] is a pointer to the parent directory. This is the directory that you are specifying with the command "`CD ..`". A directory that includes t[dot] and [dotdot] two entries is considered empty.

A directory entry contains the following fields:

```
/*
 * FAT structures
 */
typedef struct fat_file_date {
   uint32_t    day:5;
   uint32_t    month:4;
   uint32_t    year:7;
} __attribute__ ((packed)) fat_file_date_t;

typedef struct fat_file_time {
   uint32_t    sec:5;
   uint32_t    min:6;
   uint32_t    hour:5;
} __attribute__ ((packed)) fat_file_time_t;


/*
 * File info
 */
typedef struct fat_file {
   uint8_t     name[8];          /*  0 file name */
   uint8_t     ext[3];           /*  8 file extension */
   uint8_t     attr;             /* 11 attribute byte */
   uint8_t     winnt_flags;      /* 12 case of short filename */
   uint8_t     create_time_secs; /* 13 creation time, milliseconds (?) */
   uint16_t    create_time;      /* 14 creation time */
```

```
    uint16_t    create_data;        /* 16 creation date */
    uint16_t    last_access;        /* 18 last access date */
    uint16_t    h_first_cluster;    /* 20 start cluster, Hi */
    fat_file_time_t lm_time;        /* 22 time stamp */
    fat_file_date_t lm_date;        /* 24 date stamp */
    uint16_t    l_first_cluster;    /* 26 starting cluster number */
    uint32_t    size;               /* 28 size of the file */
} __attribute__ ((packed)) fat_dirent_t;
```

The file/directory's attributes can have the following values.

- `#define READ_ONLY 0x01`
- `#define HIDDEN    0x02`
- `#define SYSTEM    0x04`
- `#define VOLUME    0x08 // this is the volume label entry`
- `#define DIRECTORY 0x10`
- `#define ARCHIVE   0x20 // same as file`

For example, a file with read-only, hidden, system, and archive attributes set might be created by using the bitwise OR operator.

```
unsigned int attributes = 0x00;
attributes = READ_ONLY | HIDDEN | SYSTEM | ARCHIVE; // combine
these attributes; result is 0x27
```

The starting cluster field in subdirectory directory entries that refer to the root directory will point to cluster 0. Note that cluster 0 is reserved in the FAT table--attempting to look up that value should result in an error.

8. Something additional to know about DOS directories

The special code **0xe5** (as a character, this is a lower-case **a** with a circle above it) located in the first byte of a directory entry, indicates that this directory entry is free, and may be overwritten with a new entry in the future.

The special code **0x00** located in the first byte of a directory entry (name[0] of struct directory from paragraph "Directory structures and their fields"), indicates that the directory entry is free (same as for 0xE5), and there are no allocated directory entries after this one (all of the name[0] bytes in all the entries after this one are also set to 0).

9.FAT 12 long directory entries

Modern fat 12 file system supports more than 8 character file names (and 3 character extensions). When adding long directory entries to the FAT file system it was crucial that their addition to the FAT file system's existing design be essentially transparent to the earlier version. So, modern fat12 supports long directory entries that are presented as several 32 byte regular entries, while the fist regular entry has regular attributes (as it was in the earlier version) and remaining entries have their attribute field set to:

READ_ONLY | HIDDEN|SYSTEM|VOLUME (see parageaph " Directory structures and their fields" for more details about these #defined values).

In this assignment we will disregard information stored in long directory entries.