# Logical Structure of the Drive

For my filesystem, the drive will be formatted as the following diagram:

|  |  |  |
| --- | --- | --- |
| FAT | Root Directory | Data Blocks |

The File Allocation Table will contain an array if integers for each block in the data section. Thus it will have entries, and occupy bytes of the total drive. In the FAT, each entry will be initialized to 0, except for the first entry which will be initialized to -1. will be the number of the block that block is linked to. In this way, blocks can be made discontinuous in the drive. A -1 in the FAT will mean that the block is the last block in the chain. Since the root directory starts at block 0, it is safe to assume that no block will ever link to block 0, and therefore a 0 in the FAT will indicate that the block is not currently allocated.

Each block in the data region will contain either the data for a file, or a directory listing. If a block contains a file, it will simply be occupied by the raw data of the file. If the block is a directory, it will contain several *entry* structs that define the contents of the directory. The *entry* will be defined as follows:

The block pointed to by *start* will then contain either a subdirectory or a file. Each block will then be described by the following union:



With the FAT and the blocks thus defined, the drive can be described with the following struct:



Additionally, a filePointer struct will be defined to refer to an opened file to be written to and read from:



The *meta* element contains the file’s metadata as given in the *entry* struct, and the *ptr* and *currentBlock* integers refer to the current location in the file to be written to or read form. *ptr* will always lie between 0 and BLOCK\_SIZE, and thus refer to an offset from the start of *currentBlock* rather than an absolute location in the file.

# Details of Implementation

To make the best use of the above structs, the drive will be memory-mapped into the program. This will allow the program to view the drive as an entirely logical device, and the OS will deal with the low-level file operations. The following system call will map the virtual drive file to a *drive* struct within the program:



The PROT\_READ and PROT\_WRITE flags will allow the drive to be both written to and read from, and the MAP\_SHARED flag will synchronize the drive in memory with the virtual drive on disk. After being mapped to memory, the file descriptor may be closed.

The drive operations may then be defined as follows:

|  |  |
| --- | --- |
| Function Description | Pseudocode |
| createFile – Creates a new file (or directory) at a given location | Check if filepath is valid or if it already exists  Find first empty space in directory, add a block to directory if needed  Reserve a block for the new file  Write the new file metadata to the directory |
| openFile – returns a filePointer struct referencing the requested file | Traverse filepath to find file’s metadata  Build filePointer struct with given metadata  Return struct |
| writeFile – writes data to the given file | Write data up to the end of the current block  Write additional blocks of data, reserving blocks as needed  Update pointer and current block |
| readFile – reads a given length of data from the given file into a given buffer | Copy data up to end of block into buffer  Copy data from next blocks into buffer, until len reached or end of file reached |
| closeFile – closes a given file | Free the filePointer |
| deleteFile – deletes a given file | Set all blocks in the FAT related to this file to 0  Remove file metadata from directory |