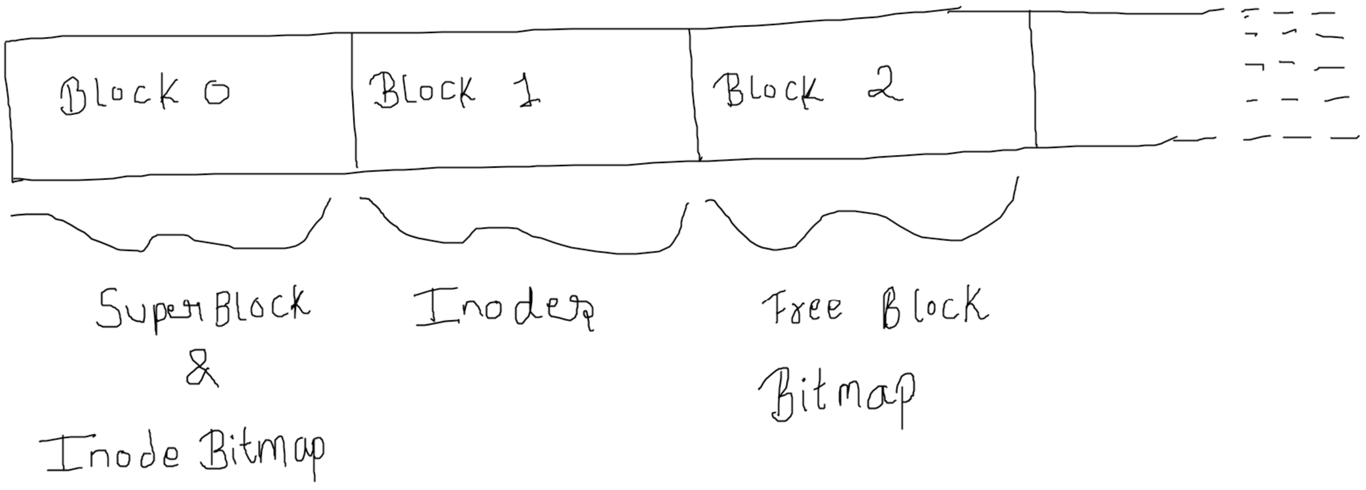
MashiFS Documentation

- MashiFS works on disk block size of 4096 or 4KB.
- This size works completely fine for all efficient I/O performance and on-disk structures storage.
  To create a disk with size 1 MB we would need 256 disk blocks.
- There are some disk blocks reserved by MashiFS for its use to identify and locate user data.
- All data types for MashiFS is at types.h header file.
- A newly created disk contains reserved blocks and free blocks.
- Disk block 0 is reserved for superblock and Inode Bitmap.
- Disk block 1 is reserved for storing Inodes.
- Disk block 2 is reserved for storing Free Block bitmap, since total bitmaps depends on TotalBlocks selected by user there can be more disk blocks after 2 for storing bitmaps.
- Disk block after last free block bitmap are Free blocks which are used by available to user and file system for storing user data (file and directories)



MashiFS File System Implementation is divided in three steps :

- Shell client
- Disk Interface
- File System Implementation

Shell Client

Shell Client is a shell interface which is used to interact with File System and Disk.
Once a disk with MashiFS is mounted shell provides various commands to help interact with user data inside disk.

The commands shell client provides are :

Mount() :

This hook mounts a disk by name DiskName and initialise MashiFS on it.

Returns **F_SUCCESS** if everything goes right else **F_FAIL** if we can't mount disk.

Create() :

This hook creates and mounts a disk by name DiskName and initialise MashiFS on it.

TotalBlocks parameter addresses how many disk blocks user wants for MashiFS.

Returns **F_SUCCESS** if everything goes right else **F_FAIL** if we can't create disk.

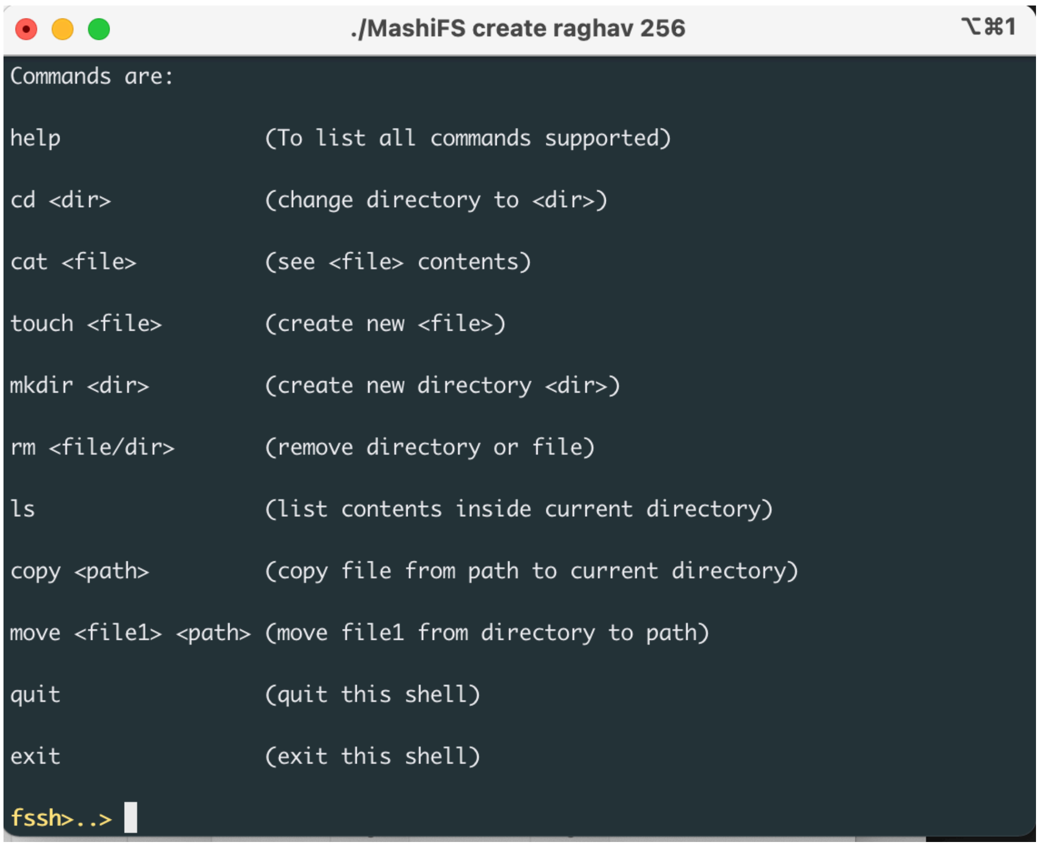
Shell() :

This hook runs Shell and performs all basic file system commands.

An infinite loop is created and runs till user inserts "quit" or "exit" command in shell.

To simplify implementation more each command is hashed to an integer value using commad_hashmap().
Using hashes for commands allows us to use them without string clutter.

The supported command are :



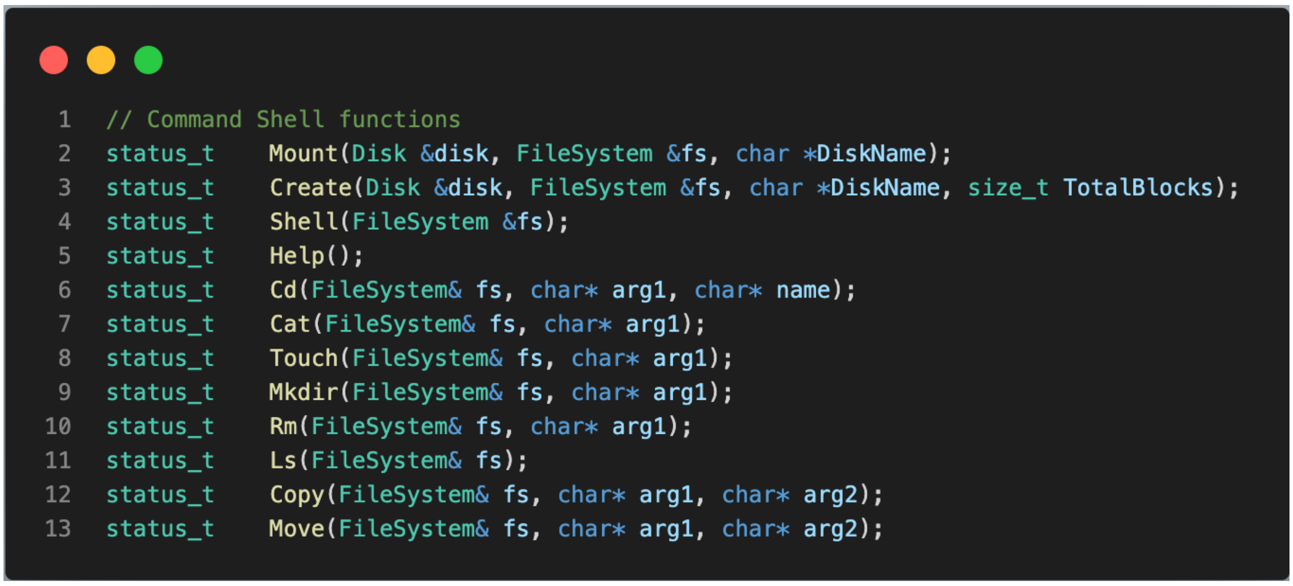
```
./MashiFS create raghav 256

Commands are:

help                (To list all commands supported)

cd <dir>            (change directory to <dir>)

cat <file>          (see <file> contents)

touch <file>        (create new <file>)

mkdir <dir>         (create new directory <dir>)

rm <file/dir>       (remove directory or file)

ls                  (list contents inside current directory)

copy <path>         (copy file from path to current directory)

move <file1> <path> (move file1 from directory to path)

quit                (quit this shell)

exit                (exit this shell)

fssh>..>
```

Returns **F_SUCCESS** if everything goes right else **F_FAIL** if we can't exit shell successfully.

Command Hooks :

To have an elegant implementation each FS has its own hook to run it on file system effectively.

Every hook returns **F_SUCCESS** if command is executed correctly else **F_FAIL** if something wrong happens.



```
// Command Shell functions
status_t    Mount(Disk &disk, FileSystem &fs, char *DiskName);
status_t    Create(Disk &disk, FileSystem &fs, char *DiskName, size_t TotalBlocks);
status_t    Shell(FileSystem &fs);
status_t    Help();
status_t    Cd(FileSystem& fs, char* arg1, char* name);
status_t    Cat(FileSystem& fs, char* arg1);
status_t    Touch(FileSystem& fs, char* arg1);
status_t    Mkdir(FileSystem& fs, char* arg1);
status_t    Rm(FileSystem& fs, char* arg1);
status_t    Ls(FileSystem& fs);
status_t    Copy(FileSystem& fs, char* arg1, char* arg2);
status_t    Move(FileSystem& fs, char* arg1, char* arg2);
```

Disk Interface

Disk class in disk.h file provides an interface to disk and its operations.

Whenever a disk operation is need like Reading/Writing disk block we use functions of disk class which provides simple and elegant way to handle operations.

Create() :

Create a new disk with given name as parameter and TotalBlocks as size of disk.

Mount() :

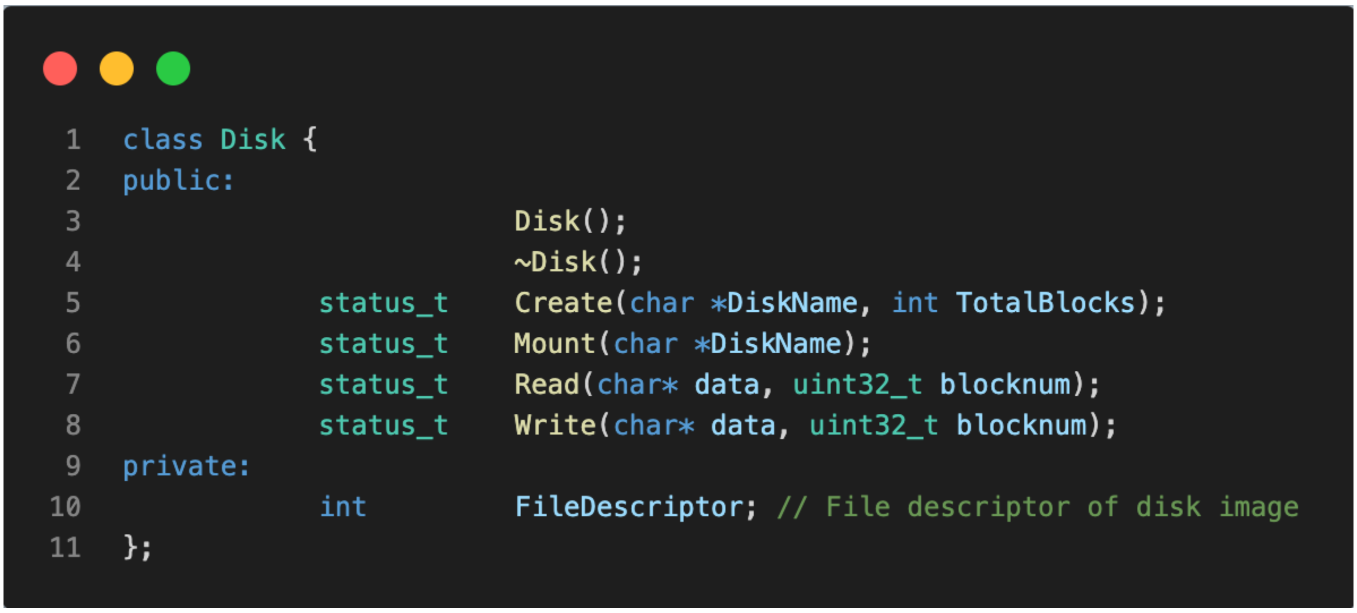
Mount an existing disk with given name.

Read() :

Read data from disk at block number into buffer passed as parameters.

Write() :

Write data to disk at block number from buffer passed as parameters.



```
class Disk {
public:
                        Disk();
                        ~Disk();
            status_t    Create(char *DiskName, int TotalBlocks);
            status_t    Mount(char *DiskName);
            status_t    Read(char* data, uint32_t blocknum);
            status_t    Write(char* data, uint32_t blocknum);
private:
            int         FileDescriptor; // File descriptor of disk image
};
```

All disk class hooks returns **F_SUCCESS** if disk operation is performed successfully else **F_FAIL** if there is some error.

File System Implementation

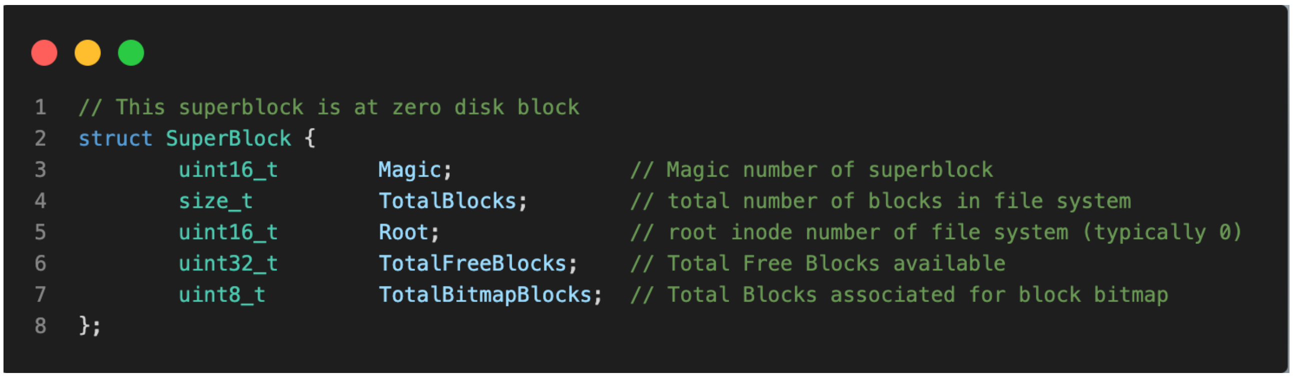
File system Implementation is divided into different hooks which works independently and performs tasks.
There are several file system structures which are Implemented.

SuperBlock :

SuperBlock is used to identify MashiFS and root inode of file system (typically 0).

Magic number of SuperBlock is SB_MAGIC (0x4C5).

SuperBlock is written in disk block 0 and at offset 0.



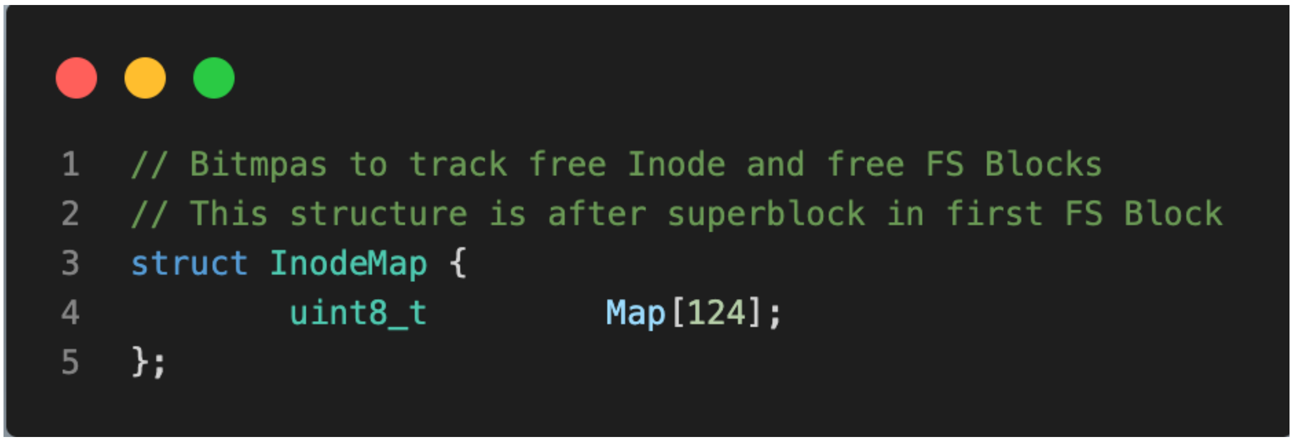
```
// This superblock is at zero disk block
struct SuperBlock {
        uint16_t        Magic;              // Magic number of superblock
        size_t          TotalBlocks;        // total number of blocks in file system
        uint16_t        Root;               // root inode number of file system (typically 0)
        uint32_t        TotalFreeBlocks;    // Total Free Blocks available
        uint8_t         TotalBitmapBlocks;  // Total Blocks associated for block bitmap
};
```

Inode Bitmap :

Inode Bitmap is used to track free Inodes in File System.

There are total of 127 Inodes available for MashiFS at current Implementation.

Inode Bitmap reside at disk block 0 and at offset sizeof (SuperBlock).



```
// Bitmpas to track free Inode and free FS Blocks
// This structure is after superblock in first FS Block
struct InodeMap {
        uint8_t         Map[124];
};
```

Inodes :

Inodes are used to track the contents of directory/file in MashiFS.

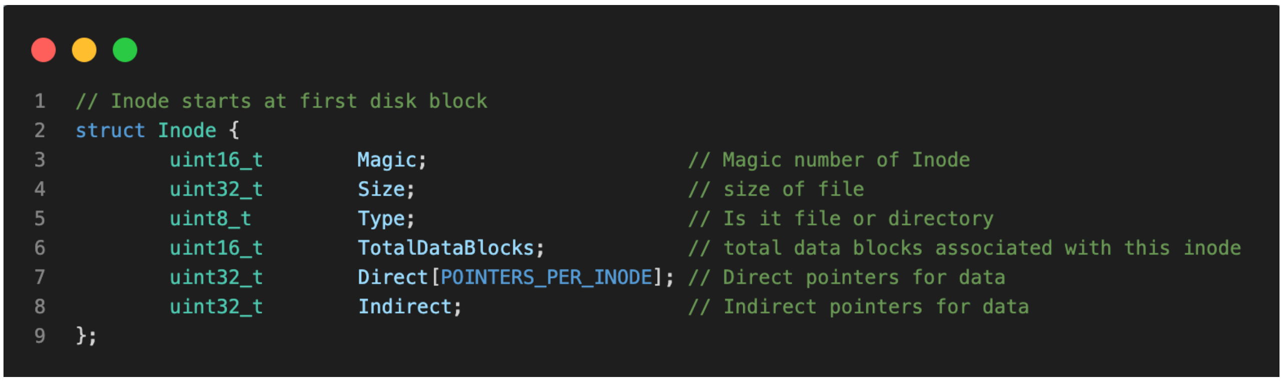
All Inodes reside at disk block 1 and at offset 0.

There are total of 127 Inodes at current Implementation of MashiFS.

Each directory or file has its own Inode.

Inode contents disk block number for directory or file data.

Magic number for Inode is IN_MAGIC (0xCEF)



```c
// Inode starts at first disk block
struct Inode {
        uint16_t        Magic;                      // Magic number of Inode
        uint32_t        Size;                       // size of file
        uint8_t         Type;                       // Is it file or directory
        uint16_t        TotalDataBlocks;            // total data blocks associated with this inode
        uint32_t        Direct[POINTERS_PER_INODE]; // Direct pointers for data
        uint32_t        Indirect;                   // Indirect pointers for data
};
```

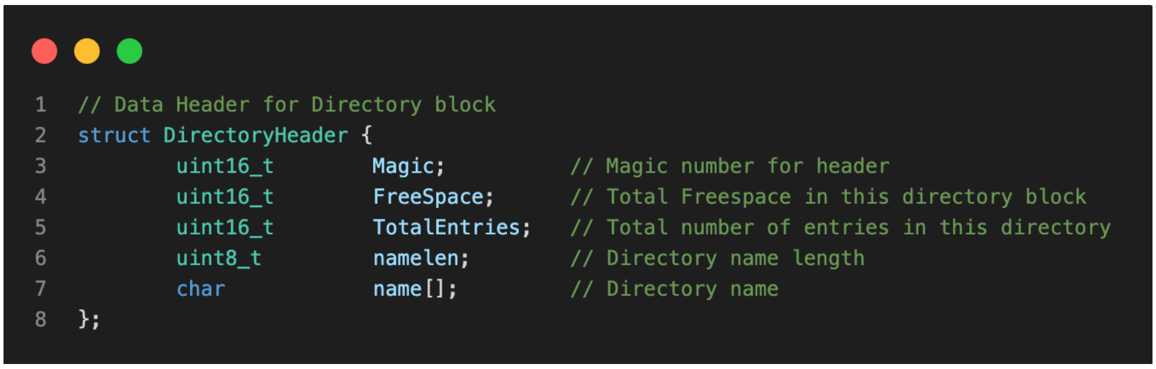
Directories :

Each Directory data block contains its own Directory Header for Identification of data.

Directory Header contains number of entries stored in that directory by user.

All directory entry are at the same data block at offset sizeof (Directory Header)

Directory Header Magic number is DIR_MAGIC (0xCEE)

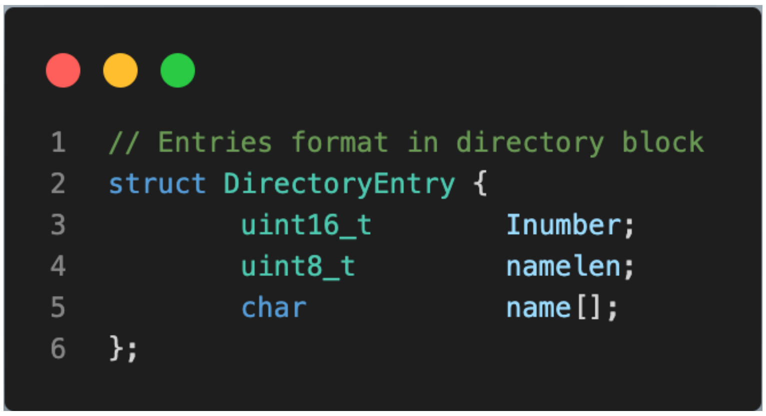


```c
// Data Header for Directory block
struct DirectoryHeader {
        uint16_t        Magic;          // Magic number for header
        uint16_t        FreeSpace;      // Total Freespace in this directory block
        uint16_t        TotalEntries;   // Total number of entries in this directory
        uint8_t         namelen;        // Directory name length
        char            name[];         // Directory name
};
```

A Directory Entry contains its Inode number and name which is used by file system hooks for command operations.

Inode number provides all the information for that entry inside directory.

There could be several directory blocks, Total number of data blocks associated with directory is mentioned by its respective Inode with Inode -> TotalDataBlocks



```c
// Entries format in directory block
struct DirectoryEntry {
        uint16_t        Inumber;
        uint8_t         namelen;
        char            name[];
};
```