Abstract:

DFFS (pronounced "doofus") – An intrinsically De-Fragmenting File System.
It's a simplified version of FAT32/FAT16 optimized for block transfers.

DFFS manages files in the FAT format except that chains are not used. Chains had their origin in the move from magnetic tape to disk storage technology. Files were designed to be created with zero length and then gradually expanded as needed. When files were put on disk drives, a method to handle expanding files was needed. Chained clusters was the solution used by Microsoft and others to manage disk storage.

Today we have FAT, which is the universal format used on today's pocket flash drives, SD cards and desktop hard drives. FAT maintains a table of pointers used to determine the sequence of clusters used by a file. The FAT entries associated with a file form a chain, where the first entry points to the second, which points to the third, etc. This allows files to be fragmented so as to handle the expanding-file paradigm. DFFS requires that FAT chains be sequential numbers. They can't jump around, so the clusters of a file are contiguous. So, chains in the usual sense are eliminated. The FAT table is used simply to keep track of allocated and bad clusters.

The advantage of DFFS is that is doesn't need to consult the FAT table when accessing files because it already knows where the clusters are. Repositioning the file pointer is instant because there is no need to traverse a FAT chain to calculate the new position. Such repositioning is important to databases that store data in blocks. Contiguous clusters make for simpler, more compact firmware as well as streamlining physical access because files aren't fragmented. DFFS requires less RAM and ROM than FAT to get good performance.

The fundamental operation in DFFS is RELOCATE, which when given a directory entry and a desired file size, scans the FAT table for a contiguous empty region of that size and relocates the entire file. Expanding files may need to relocate sometimes. This is where DFFS can be painful when used with the expanding-file paradigm. You don't want a bunch of relocations taking place. The pain can be minimized by using RESIZE-FILE to allocate enough storage in the first place.

Whenever a chain is encountered during normal operation (such as when a directory is scanned or a file is opened), RELOCATE is invoked to get rid of the chain. This way, chains introduced by Windows and other operating systems are automatically removed. DFFS partitions look like FAT but without fragmentation.

DFFS interfaces to the physical drive using functions that read and write a series of one or more 512-byte sectors. Functions are also provided to power the drive up/down and test for the presence of the drive.

Some units of measurement need to be defined in order to describe the file handle structure:

- SECTOR is a 512-byte basic unit of physical storage.
- CLUSTER is a an n-sector unit of virtual storage addressed by the FAT table. Files are an integral number of clusters.
- S/C is sectors per cluster. Usually 8. Maximum is 128.
- PHYBYTE is a 64-bit physical byte address consisting of a 55-bit sector number and 9-bit byte selector.

File handles are structures containing the following data:

Name	*Bits*	*Contents*
`Volume`	8	Volume ID of this file
`Ddir`	56	Phybyte of the directory entry. 0 if unused handle.
`Dbase`	55	Starting (phybyte<<9) of the file
`FAM`	9	File Access Method flags: R, W, BIN, CR
`DFP`	64	Current file position ($0..2^{64}$)
`Dlimit`	64	Maximum file position

The maximum disk size (and file size) supported by the file handles is 2^{64} bytes = 1.8×10^{10} TB. With hard disk size doubling every two years, this limit should occur around the year 2055. A FAT32 drive has a limit of 2TB because of limitations of the FAT and partition tables. Multiple partitions don't push the limit past 2TB and they only complexify the code, so only one partition is supported. If FAT compatibility weren't needed, DFFS could use a non-FAT method of tracking blank clusters in order to beat the 2TB limit.

Directory entries point to files or directories. If they point to directories, the file length is 0. In order to find the length of a directory, it must be scanned until a terminator is found. Files have a known length. In order to scan a directory, you open it as a file. DFFS figures out the length of this "file" by scanning until the terminator. The root directory has a zero-length string as its name. Typical directory names present in a directory are "." which points to the current directory and ".." which points to the parent directory.

There are many clusters on a FAT32 drive, so there are many entries in the FAT table. A 40GB drive formatted with 32KB clusters has about 262K FAT entries. Finding free clusters in this table can take considerable time if the disk isn't empty, since it's a linear search. As an example of the worst case, consider an SD card clocked at 25 Mhz. The 32-bit FAT entries can be read at a rate of 780K/sec so if a 4GB card is almost full, it takes up to 0.8 second to scan the FAT table. This scan is done at startup, building a sorted table of empty regions.

Note that today's NAND memory card physical layer specifications have limits far below 2TB:

Standard SD:	2GB
High Capacity SD:	32GB
CompactFlash:	137GB

DFFS will typically be most used in resource constrained embedded systems, so the implementation doesn't go out of its way to go beyond 137GB. That means sector numbers fit in 32 bits.

Physical access to a drive is handled by functions that open/close the drive, access sectors on the drive, and poll the drive to see it it's present. Within a drive, there can be up to four partitions (volumes).

If multiple physical drives are handled, phybytes within the file structure (Ddir and Dbase) map to multiple physical drives.

The simplest implementation uses a single 512 byte buffer to hold sector data. When small amounts of data are read from a file, often it is read from this buffer. When small amounts of data are written, it is cached in the buffer until the buffer is full or another sector requires access. Large data transfers mostly access the drive directly instead of going through the buffer.

A Proposed Implementation

Some single-instance parameters include:

```
Paths           constant
PathLen         U8
Path            U64[Paths]
```

The path is tracked by a list of directory phybytes in **Path**. **PathLen** is the nesting level, with 0 being the root only. To display the path, you list the directory names pointed to by each element from first to last. A nonzero **PathLen** indicates that the current directory is not the root directory. There are **Paths** subdirectories allowed in the path.

```
HeadSpace       constant
```

When a file is growing and needs to be expanded, **HeadSpace** is the number of clusters that need to be added to the file's disk region. If it runs into used clusters, RELOCATE is invoked to move the file to a bigger space.

```
Handles         constant
Handle          handlestruct[Handles]
```

Handles is number of concurrently open files supported. **Handle** is a data structure containing access parameters for the file.

```
Drives          constant
```

The number of physical drives supported.

```
MaxVolumes      constant
```

The total number of volumes supported.

```
Volumes         U8
Volume          U8
```

Volumes is the number of volumes found at initialization. This includes all volumes on all drives. **Volume** is current volume number.

At startup, the sector of the MBR is computed from the partition table. At initialization, the drives are opened and the MBRs of each drive are scanned to count the **`Volumes`** and initialize their data structures. Depending on how many volumes are supported, there are one or more instances of the following variables:

`DriveID` **`U8`**

The ID of the drive being accessed.

`spc` **`U8`**

Log_2(sectors per cluster).

`VolSector` **`U32`**

The beginning sector of the current volume. This points to the MBR, which contains various FAT parameters. Things that are seldom used don't merit their own variables. When you need them, just read them from the MBR.

`DirSector` **`U32`**

The beginning sector of the current directory. When a directory is changed, a FAT scan ensures that the directory is monotonic and then updates **`DirSector`**.

`RootSector` **`U32`**
`RootSectors` **`U16`**

`RootSector` is the beginning sector of the root directory. The root directory doesn't use a FAT chain. It's just a fixed block of **`RootSectors`** sectors.

`FATsector` **`U32`**

The beginning sector of the FAT table.

`BaseSector` **`U32`**

The beginning sector of cluster 0.

`FreeSpace` **`array 2*U32*n`**

A list of runs of unused clusters.