FAT file systems for embedded systems and its optimization

Jan Horký[1]
[1]Department of Circuit Theory
Czech Technical University in Prague
Prague, Czech Republic
horkyja3@centrum.cz

Matouš Pokorný[1]
pokormat@fel.cvut.cz

Jan Havlík[1]
xhavlik@fel.cvut.cz

***Abstract* – Because of both limited memory size and low computational power, file systems in embedded systems are usually designed to have low memory requirements. For these reasons, one of FAT (File Allocation Table) file systems FAT16 or FAT32 is often chosen due to its good performance on small volumes and low overhead. When there is a need to perform data logging (perform write operations in high frequency, while storing each time the same amount of data), it is necessary to minimize amount of read and write accesses to the drive in order to reach desired frequency of writes. Due to limited memory on embedded system, it is also necessary to keep buffer sizes as low as possible, which severally limits possibilities of caching. Therefore, there is presented FAT-compatible file system library, allowing data logging in high frequency, and it is compared to other FAT file system libraries for embedded systems.**

File allocation table, FAT, data logging, embedded systems. (key words)

I. INTRODUCTION

File Allocation Table (FAT) is a file system with origins in late 1970s and early 1980s, originally developed for floppy disk drives. Although FAT is nowadays usually not used on hard drives anymore, being replaced by NTFS (New Technology File System), Ext4 (fourth extended file system) or others, it is still widely used on removable media and in embedded systems. Reasons for this are as follows: FAT offers relatively good performance even in very lightweight implementations and has little overhead compared to other file systems. These features make it suitable for small volumes and systems with limited computational power. Nowadays, there are three FAT file system types: FAT12, FAT16 and FAT32, the number representing size, in bits, of entries in FAT structure.

This article explains principles of FAT file system and focuses on current implementations of FAT for embedded systems – DosFs, FatFs and FAT SL, theirs performance and features. After that, there is presented own implementation of FAT file system, based on DosFs and designed in order to allow high frequency of write operations during data logging, while maintaining low memory requirements. For this reason, it is called LDFS – Logging disk file system.

In the end, all these implementations of FAT file system are compared and it is discussed, whether similar optimizations made in DosFs-based LDFS are possible in other implementations and whether they would result in performance gain.

II. PRINCIPLE OF FAT FILE SYSTEM

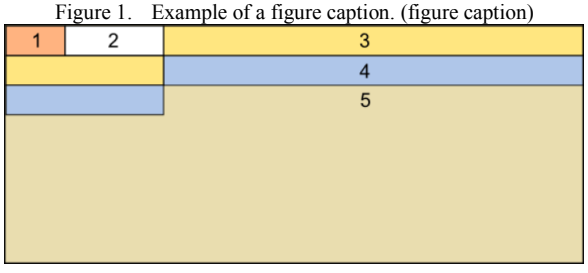
The difference between FAT12, FAT16 and FAT32 lies mostly in the size of entries in FAT structure, which is 12, 16 or 32 bits. Based on this, FAT12, 16 and 32 also have different maximum volume sizes. Further, we dismiss FAT12 and FAT16, and focus only on FAT32. For addressing, we use Logical block addressing (LBA).

When reading FAT32 volume, at first, it is necessary to read its first sector, which is called Volume ID. This sector contains among others number of bytes per sector (which is always 512), number of sectors per cluster, number of reserved sectors, number of FATs, sectors per FAT and number of first cluster of root directory.

The layout of FAT file system is as follows: 1 – Volume ID, 2 – reserved sectors, 3 – the first copy of FAT, 4 – the second copy of FAT, 5 – data area, see Fig. 1. There are two FATs in case one got damaged, so data can be retrieved from the other one. The data area is used to hold all files and directories and clusters begin their numbering at 2, see Fig. 2. Root directory entry (as well as subdirectory entries) contain list of files inside the directory, containing 8.3 short name, attributes, first cluster of this file and file size [2].

FAT is a table of 32bit values where each index represents one cluster, and contents of table at this index represents following cluster of file. If the file ends at current cluster, the index of this cluster has EOF marker value in FAT. End of file (EOF) marker is usually 0xFFFFFFFF, but it can be anything higher than 0x8FFFFFFF [1].

Figure 1. Example of a figure caption. (figure caption)



ISBN 978-80-261-0386-8,

Figure 2. Part of the FAT32 table – first two entries are unused, then there is root directory entry with size of one cluster (green), followed by file at clusters 3, 4, 6 (orange). Another file occupies clusters 5, 10, 11, 12 (yellow).

XXXXXXXX	XXXXXXXX	0xFFFFFFFF	0x00000004
0x00000006	0x0000000A	0xFFFFFFFF	0x00000000
0x00000000	0x00000000	0x0000000B	0x0000000C
0xFFFFFFFF	0x00000000	0x00000000	0x00000000
0x00000000	0x00000000	0x00000000	0x00000000

III. FAT FILE SYSTEMS FOR EMBEDDED SYSTEMS

In this section, there will be presented current FAT file systems used in embedded systems, their features and performance.

A. *DosFs*

DosFs is a FAT-compatible file system intended for low-end embedded applications and supports devices up to 2048 Gbytes in size. It is stateless, supports subdirectories and can be operated with a single global 512-byte sector buffer. It performs no memory allocation and has partial support for random-access files. DosFs sets file timestamps always to the same value and supports only 8.3 file names [4].

Although DosFs is fairly good solution for low-end applications, it suffers from frequent read and write accesses to the device. This is caused by not caching any parts of FAT in order to minimize memory requirements – the design requirement of DosFs is to operate at single 512-bytes scratch buffer. This can be noticed especially while setting clusters of FAT to new values, where read and write accesses are performed every time a single cluster is set. Because of this, one write operation can have high amount of read/write accesses to the device – read and write data, actualize FAT (read and write again) and finally actualize directory entry. Additional accesses are required, if FAT12 file system is used. On the other side, because everything is written directly to the device without any buffers for FAT sectors etc., it is not necessary to close the file before removing device. Close function does not exist (or rather, it is blank) and therefore no damage to the file can happen.

DosFs was used in operating system CircleOS, which is designed for STM32 Primer series, but it was replaced by FatFs in CircleOS 4.5.

B. *FatFs*

FatFs is generic FAT file system for small embedded systems and is completely separated from the disk I/O layer, meaning it does not depend on hardware architecture and therefore, it can be easily ported. It supports long file names, maximum file size is 4 GB [3].

Each file opened in FatFs has its own buffers, consisting of sector size buffer for data and sector size buffer for FAT or directory entry. One of the most significant differences between FatFs and DosFs is that FatFs uses buffer for parts of FAT and therefore, it has significantly lower amount of read and write accesses to the drive. This is particularly important on flash memory, because this memory has limited amount of read/write cycles and frequent accesses could result into damage.

There is also smaller version of FatFs, which also buffers FAT, but uses the same buffer for it as for the data.

C. *FAT SL*

FAT SL stands for Super Lean FAT File System, which is DOS compatible and supports FAT12, FAT16 and FAT32. It is designed to have minimal Flash and RAM footprint and can be used on wide range of 8, 16 and 32-bit MCUs.

FAT SL can work at one time with only one file and one volume, and it has buffer with size of sector, which is used for data. No buffering of FAT happens and file, volume and buffer are global variables.

The main advantage of FAT SL lies in low memory footprint and it is currently used in some versions of operating system FreeRTOS [5].

IV. IMPROVEMENTS OF DOSFS FOR DATA LOGGING IN EMBEDDED SYSTEMS

During another project, it became necessary to write measured data on microSD card at frequency of 200 Hz while maintaining low memory footprint, because this memory was needed for other purposes. At first, DosFs was tested for this purpose, as it was already included in operating system CircleOS, which was used. It was however found, that it is not capable of reaching desired frequency, ending up with only about 72 writes per second when writing data with size of 16 bytes. While FatFs, which was tested as next, provided better results due to buffering of FAT, it was still capable of reaching frequency only around 100 Hz, with occasional drops to 65 Hz – 70 Hz. These drops were mostly results of additional read/write accesses, as FAT buffer was getting filled from the device, or its contents was written to the device.

For these reasons, it was decided to design and implement own FAT-compatible file system library capable of desired tasks. For basis of this file system, DosFs was taken for its low memory requirements, relative simplicity and because it was a part of the CircleOS operating system. It will be referred to this system as LDFS – Logging Disk File System.

It was found, that insufficient performance of DosFs was caused by frequent accesses to FAT and directory entry, causing 4 – 12 read/write accesses per one write operation, depending on whether it was necessary to search for free space in FAT and update its contents. In order to reach desired frequency of write operations, it became necessary to get rid of these additional read/write accesses.

A. *LDFS principles*

As previously mentioned, for the basis of LDFS, DosFs was taken, and then modified severally. The

concept of DosFs with no global variables and no states was discarded. Instead, global variables for file info and volume info structures were introduced, together with scratch buffer in size of one sector for each file. In order to reduce read/write accesses, it was decided to handle files opened for data logging in the following way: Instead of accessing FAT and directory entry during write operations, it was decided to specify maximum allowed size of the file when the file is being created (open for write). After the maximum size of the file is specified, FAT is searched for continuous free space of this size. If such area of free space is found, directory entry is updated for this size and blank file of this size is created by updating FAT to make this free space into one file, see Fig. 3 and Fig. 4. The result of this is, that although there are no valid data written in the file yet, we already have FAT and directory entry set, and do not have to touch them during write operations. Later, when file is closed, directory entry is updated to resemble current file size and in FAT, cluster entries which were assigned to the file but not used, are marked as free space and last used cluster entry of the file is updated with EOF marker, see Fig. 5. It would be possible to set the FAT and directory entry only during closing the file, but it would have one significant drawback – if the file is not closed correctly (device accidentally turned off, data media removed), all written data would be lost, as there would not be any clusters belonging to this file in FAT. When we set the FAT at the time of creation of file and only update it when file is being closed, the only thing that can happen when file is not closed correctly is, that it will have incorrect size and apart from the written data, it will also contain random invalid data. Though, no data will be lost, except for current contents of scratch buffer.

This way, most of the additional read/write accesses during write operations were removed. As for the write operations themselves, they are implemented as follows: Upon each call to the write function, data are copied to a buffer with size of a sector (512 B) and when this buffer becomes full, it is written to the device and cleared. Because the file is initially empty, there is no need to perform any read access, as there are no data to be lost by rewriting. It should be noted that after last write operation, function for finishing last write has to be called in order to write not yet completely filled sector size buffer to the device. The same actions for finishing last write also have to be taken before sector size buffer is used for different purposes, otherwise data in it would be lost.

B. LDFS with free space bitmap

In previous subsection, there were explained principles of LDFS file system library. These changes brought desirable performance and allowed data logging in high frequency, with maximum frequency of write operations high above 200 Hz.

However, such implementation of LDFS has difficulties in creating new files, because it needs continuous free space of desired size. While this is not a problem with completely free disk / card, it becomes a serious problem when there are many files on the card and file system is fragmented. The result of this can be, that the file cannot be created despite that there is enough space on the disk, but not in one continuous chunk, see Fig. 6.

Figure 3. FAT before file is created, contains only directory entry at cluster 2.

XXXXXXXX	XXXXXXXX	0xFFFFFFFF	0x00000000
0x00000000	0x00000000	0x00000000	0x00000000
0x00000000	0x00000000	0x00000000	0x00000000
0x00000000	0x00000000	0x00000000	0x00000000

Figure 4. FAT after file is created with maximum allowed size of 7 clusters.

XXXXXXXX	XXXXXXXX	0xFFFFFFFF	0x00000004
0x00000005	0x00000006	0x00000007	0x00000008
0x00000009	0xFFFFFFFF	0x00000000	0x00000000
0x00000000	0x00000000	0x00000000	0x00000000

Figure 5. FAT after file was closed. Because only 4 clusters were written, the remaining cluster entries are marked as free space, last used cluster entry is updated with EOF marker.

XXXXXXXX	XXXXXXXX	0xFFFFFFFF	0x00000004
0x00000005	0x00000006	0xFFFFFFFF	0x00000000
0x00000000	0x00000000	0x00000000	0x00000000
0x00000000	0x00000000	0x00000000	0x00000000

Figure 6. Operation of creating new file with size of 7 clusters fails, because there is not enough continuous space in the FAT, as part of it is already occupied by another file.

XXXXXXXX	XXXXXXXX	0xFFFFFFFF	0x00000000
0x00000000	0x00000006	0x00000007	0x00000008
0x00000009	0x0000000A	0xFFFFFFFF	0x00000000
0x00000000	0x00000000	0x00000000	0x00000000

Figure 7. Example of FSB, allocation units from 2 to 5 are used, the rest is free (indexed from zero).

0	0	1	1	1	1	0	0

This issue was mitigated by the introduction of free space bitmap to LDFS. Free space bitmap (FSB) is an array, see Fig. 7, in which each bit represents allocation unit (sector, cluster, …). If the bit is set, it means that corresponding allocation unit is used. Otherwise, it is free. Although it is not possible to cache neither whole FAT, nor even its representation in FSB with each bit corresponding to one cluster, as it would take up too much memory, FSB can still be used in the following way: Instead of each bit matching one cluster, it is possible to match each bit to block of clusters. If such block is completely empty, its corresponding bit is clear, otherwise the bit is set. The size of the block is calculated from cluster size, FSB buffer size and total size of media. For example,

for microSD card with 4 GB size, cluster size 4096 KB and 256 B FSB buffer, each block has approximately 16 MB in size [6].

With the use of FSB, it is no longer needed to have continuous chunk of free space for the whole file. Instead, it is enough to have enough free blocks in which the file could fit. Since whole FSB resides in a buffer, no further accesses to FAT are needed and therefore, the performance of file system is not hampered in any way.

V. Comparison and results

For testing purposes, simple application which runs on CircleOS 4.6.1 was developed. This application tries to perform as much write operations per second as possible. For data logging purposes, DosFs, FatFs and LDFS were tested. FAT SL was not tested, because after browsing its code it became clear, that for data logging it did not provide any performance advantage over DosFs. Results of the test are presented in Tab. 1.

Testing was done on EvoPrimer STM32F103VE with 72 MHz Cortex M3 CPU, 512 KB Flash memory and 64 KB RAM with operating system CircleOS 4.6.1. [4]. Data were written to microSD card connected via SDIO interface and then viewed in PC.

The results of the test clearly show, that DosFs-based LDFS has significantly better performance during data logging than either DosFs or FatFs, with write frequency high above the desired limit of 200 Hz. However, it should be clear that functions for fast data logging in LDFS are of limited use for other purposes than data logging and therefore for these tasks, LDFS has to use functions taken directly from DosFs.

TABLE I. Results of data logging test

File System	Bytes per write	Write frequency	Speed
DosFs	16 B	72 Hz	1.152 KB/s
FatFs	16 B	< 100 Hz	< 1.6 KB/s
LDFS	16 B	2300 Hz	36.8 KB/s
	32 B	2060 Hz	65.92 KB/s
	64 B	1850 Hz	118.4 KB/s
	128 B	1240 Hz	158.72 KB/s
	256 B	590 Hz	151.04 KB/s
	512 B	380 Hz	194.56 KB/s

VI. Further possible development

Taken into account the results of data logging test and other FAT file system implementations, mostly FatFs, it is obvious, that new data logging functions in LDFS provide significant performance gain for such task. It was considered, whether there would be any further performance gain for FatFs-based LDFS because of overall better performance of FatFs over DosFs. However, it was found that no further speed-up of data logging was possible this way, because during data logging, there are already no additional read/write accesses except for writing logged data to the device. The main reason for better performance of FatFs over DosFs is buffering of parts of FAT in FatFs. However, this feature cannot be implemented into LDFS, because it already uses FSB to store information of which blocks of clusters are free/used. While this means that data logging functions performance can't be increased this way, it is possible to increase performance of standard file handling functions of LDFS in one of the following ways: 1. Create FatFs-based LDFS, which would use FAT buffer as FSB during data logging. Since FAT buffer is one sector in size, further modifications would be necessary in order to make it work with FAT buffer of various sizes. 2. Implement FAT buffering into current DosFs-based LDFS, bringing performance of standard functions on par with FatFs. For FAT buffering, buffer used for FSB would be used. Another improvements of LDFS would include functions for fast reading of file, using mechanisms similar to those used in data logging functions for fast write operations and support for FAT16 file system – currently, LDFS supports only FAT32.

Acknowledgment

Research described in the paper was supported by the Czech Technical University under the grant SGS14/191/OHK3/3T/13 – Advanced Algorithms of Digital Signal Processing and their Applications.

References

[1] MICROSOFT CORPORATION. (2015, Jun 29). Hardware White Paper – Designing Hardware for Microsoft Operating Systems: Microsoft Extensible Firmware Initiative FAT32 File System Specification. Microsoft Corporation [Online]. Available: http://download.microsoft.com

[2] P. Oppenheimer. (2015, Jun 29). File System Forensics FAT and NTFS [Online]. Available: http://www.priscilla.com

[3] E. Styger. (2015, Jun 29). FatFs: An Open Source File System [Online]. Available: http://www.embeddedcomputingconference.ch

[4] RAISONNANCE. (2015, Jun 29). CircleOS V4.6 Conception document [Online]. Available: http://stm32circle.com

[5] FreeRTOS. (2015, Jun 29) Super Lean FAT File System [Online]. Availbale: http://freertos.org.

[6] R. Hyde, *Writing Great Code: Understanding the Machine*. San Francisco, CA: No Starch Press, 2004.