TEFS: A Flash File System for Use on Memory Constrained Devices

by

Wade Harvey Penson

B.SC. COMPUTER SCIENCE HONOURS THESIS

THE UNIVERSITY OF BRITISH COLUMBIA

(Okanagan)

April 2016

© Wade Harvey Penson, 2016

Abstract

A file system is used to manage data on storage media. The FAT (File Allocation Table) file system was originally designed for floppy drives that were less than 500KB in size, and these drives were not capable of fast random reads and writes. FAT has been adapted to work on other types of storage devices since, and it is still widely used today. It is the standard file system used by microprocessors and embedded devices with constrained resources. Micro-controllers, like the Arduino, only officially support the FAT file system when interacting with a SD card. FAT performs well when data is read or written sequentially, but when data is read or written randomly, there is an impact on performance for large files on page based flash devices that cannot utilize caching strategies. Applications that perform random reading and writing are impacted by this architectural issue. For example, flash data structures, like a B-Tree, will have poor performance since random reading is utilized to look up values. TEFS (Tiny Embedded File System) uses a tree indexing structure to take advantage of the fast random reads and writes of flash storage and guarantees that the number of page reads and writes will stay constant as the file size increases when randomly reading or writing. Experimental results show that TEFS has significantly better performance than FAT on the Arduino for random I/Os, and the more efficient TEFS page interface is even slightly faster than FAT for sequential reading and writing.

Preface

During the time working on my honours research, my conference paper on TEFS, which is a succinct version of this paper, has been accepted by CCECE 2016.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank Ramon Lawrence, my supervisor, for his guidance and for all of his effort of helping me throughout this project. I would also like to thank Scott Fazackerley, a PhD student, for helping me with the initial design of TEFS and for giving me helpful advice throughout. Finally, I would like to thank NSERC for supporting my research on this project during the summer of 2015.

Chapter 1

Introduction

Embedded devices typically utilize flash based storage for persistent data [Mic06]. File systems are used on storage to organize the data such that the files have associated metadata and are indexed by a name. Devices, like micro-controllers, that have constrained program storage and constrained RAM need this file system to utilize the least amount of resources as possible.

NAND flash devices such as SD, MMC, USB flash drive, and CompactFlash have a page-emulation layer (a page is the smallest physical unit of addressable memory). The next layer above that is a Flash Translation Layer (FTL) that provides an interface which allows logical pages to be read from and written to, hides bad erase units, and performs wear-leveling [ZBT09]. These flash storage devices are also known as Memory Technology Devices (MTD), and in this paper, only this type of storage device will be considered [Mul14]. File systems such as Ext2 [CTT15] and FAT [Mic00] are designed to work on these devices. NAND flash devices that do not have a FTL and have their erase clusters exposed utilize file systems such as JFFS [Woo01], YAFFS [LP06], NANDFS [ZBT09], and Coffee [TDHV09]. However, these log-structured file systems are too resource intensive, and the code and structure is too complex for use on memory constrained embedded devices [DNH04]. Coffee is an exception to this, but it assumes that files are fixed in size. Coffee allows for files to expand beyond the fixed size, but the whole file must be copied to a different location [TDHV09].

This work presents TEFS, a Tiny Embedded File System, that offers better performance, a smaller code footprint, and less RAM utilization compared to the industry standard file systems that exist for micro-controllers with constrained resources. The standard file system used for these devices is typically FAT [Tec13][MRR14] or a derivative of FAT. TEFS provides faster random reads and writes compared to the FAT file system for large files on memory constrained devices.

There is another file system that is in current development for embedded devices, which is not a derivative of FAT. The file system is called lwext4 and is a port of Ext [Kos16]. There are benefits to this file system compared

to TEFS such as caching, journalling and transactions, metadata checksum, and limited compatibility with existing Ext implementations on Linux and other environments. This file system will not work the devices that TEFS is targeting. It requires 8KB of SRAM at minimum and is intended for use on devices such as the ARM Cortex-M series [Kos16].

TEFS offers two APIs. The first is a page interface, and the second is a C file interface. The page interface has better performance since it does not need to map the specified byte location in the file to the corresponding logical page and byte within that logical page on the device. The C file interface is convenient for applications that are already adapted to use this interface as the C file interface is included in a standard library within the C language.

The paper organization is as follows. In Chapter 2, different types of flash memory and the FAT file system are introduced. Chapter 3 covers the design of TEFS, its functionality, and the properties that distinguish it from FAT. Chapter 4 presents experimental results, and Chapter 5 is a discussion of the trade-offs of TEFS and FAT. The paper closes with future work and conclusions in Chapter 6.

Chapter 2

Background

Without a file system on the flash device, data is written to addressable logical or physical pages on the device. For specific applications, this may be all that is needed. It is the fastest way to read and write data as it does not have the overhead of the data structures to manage files. This may be sufficient for fixed size records. Otherwise, in most cases, a file system is needed.

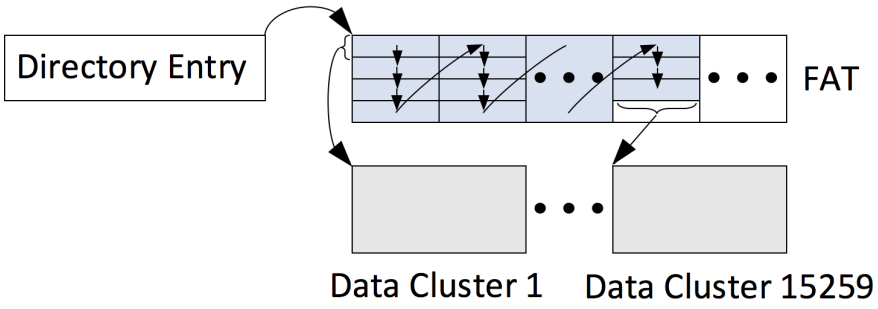
There are various structures for different parts of a file system. There needs to be a way to store metadata about a file, and in UNIX terminology this is typically done with a structure called an inode or index node [ADAD15]. Another part is the directory, and it has entries of file names and inode address pairs in a structure for lookup. The simplest inodes have direct pointers where each pointer points to a data cluster (a cluster is a sequence of a fixed number of pages). However, this limits the max file size significantly so a structure like indirect pointers or linked list is used instead to allow for larger files. FAT takes the linked list approach whereas TEFS uses indirect pointers.

FAT12, the initial FAT file system, was designed for floppy drives that were less than 500KB in size [Mic00]. It was later extended by FAT16 and FAT32 to support larger storage devices. The FAT file system is made up of a boot record, the File Allocation Table (FAT), the root directory, and data clusters. The boot record contains information that pertains to the file system, the storage device, and the partitions. The root directory is a fixed size for FAT12 and FAT16, but for FAT32 it allows for chained clusters. A directory entry for a file in FAT stores the metadata for the file and has a pointer to the first address for the file's cluster chain in the File Allocation Table. The File Allocation Table is a single array of addresses shared by all files, and the size of an address is 12, 16, or 32 bits depending on the version of FAT file system. The FAT is fixed in size and depends on the size of the device. Each address in the FAT points to another address in the FAT to form a linked list (or cluster chain). The location of where the address is in the FAT determines the location of the corresponding data cluster on disk. The last address in the cluster chain for a file is set to a large value

to indicate that it is the last data cluster in the file. If an address is 0, the data cluster is free and can be reserved by a file.

Figure 2.1 shows a file represented by the FAT file system. Ellipses indicates that there is a series of pointers, clusters, or reserved space on disk and arrows are pointers to clusters on disk or a pointer to a location in the FAT. The directory entry for the 500MB file points to the first address in the chain. The first address points to the second address in the chain and so on until the last address of the cluster chain. The position of an address in the chain correlates to the position of the data cluster. For example, the first address in the cluster chain for the 500MB just happens to be the first address in the FAT so it correlates to the first data cluster. Since the FAT is shared by all of the files, the addresses for a file's cluster chain in the FAT may not be adjacent to one another depending on the write pattern of data for the files. This may require more page reads in the FAT to reach the target data cluster in a file.

Figure 2.1: Representation of a 500MB File by the FAT32 File System



The directory structure of Ext3 and Ext4 uses a H-Tree which is a tree structure that uses the hash of a file name as keys [Phi01]. The tree structure makes the lookup of files quite fast. Other file systems such as UFS use a B-Tree or a variation of a B-Tree for the directory structure [Phi01]. The issue with using a data structure like these is that they are complex structures that consume a significant amount of code space [Phi01]. An improved version of the linear search, as seen in FAT, is used for the directory of TEFS to minimize the code size.

NAND flash devices are page addressable and not byte addressable [FL11]. This requires that a complete page be read or written even if only a single byte is to be read from or written to that page. The implication of this is

that if data in an existing page is to be modified, the complete page must be read into memory followed by the complete page being written back to the device. There are NOR flash devices that are capable of byte reads and writes [FL11], which may be utilized to reduce page buffering in memory. This allows for faster reading and writing and smaller RAM utilization.

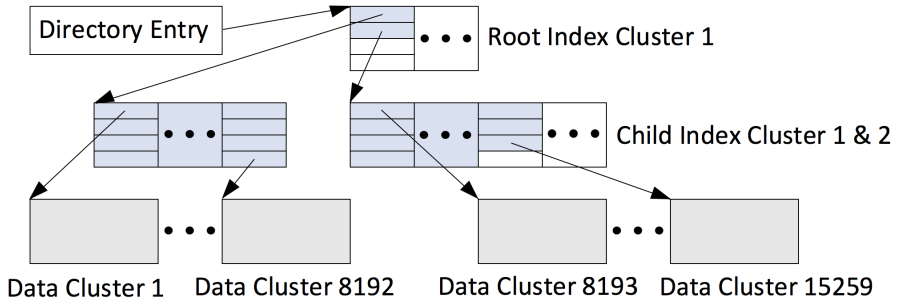
Computers typically have a significant amount of Random Access Memory (RAM). Depending on the amount of RAM, file cluster chains, directory entries, or the whole FAT and directory could be cached [Mic00]. However, a subset of embedded devices and micro-controllers have a limited amount of RAM [FL11] and are not capable of caching the complete index for a file in memory. TEFS was designed to perform well on these devices.

Chapter 3

Structure and Operation of TEFS

TEFS provides an interface for opening, closing, and removing a file; reading and writing records to and from pages; and formatting the storage device. The layout of TEFS consists of three essential parts. The first part of the file system is the information section. It is located at logical page 0 and contains information about the storage device and other information provided by the user when formatting. The second section is the cluster state section. It is a bit vector where each bit indicates the status of a cluster on the storage device. The last part of the file system contains the tree index structure for each file. Each file consists of a single root index cluster, child index clusters, and data clusters. Figure 3.1 demonstrates this. For smaller files, the root contains a sequence of addresses that point directly to data clusters. Larger files have a root index cluster that points to child index clusters and the child index clusters point to data clusters. The directory structure is contained within two files. The first file contains hashes for file names and the second contains the metadata pertaining to each file.

Figure 3.1: Representation of a 500MB File by TEFS



3.1 Information Page

This page is created on format and its contents are static with the exception of the sizes for the files used for the directory structure. It contains the values needed for the operation of the file system. Here are all of the fields with a description of each:

Byte 0 to 3: Check flag

This flag 0xFCFCFCFC is used to verify that the storage device has been formatted.

Byte 4 to 7: Number of pages[2]

The number of pages that the storage device has.

Byte 8: Physical page size[1,2,3]

The size of a page in bytes.

Byte 9: Cluster size[1,2,3]

The size of a cluster in terms of pages (not bytes).

Byte 10: Address size[1]

The size of an address in bytes. The address is the physical location of a page on the storage device. It is either 2 or 4 bytes.

Byte 11: Hash value size[2,3]

The size of a hash in bytes. The size can be either 2 or 4 bytes.

Byte 12 to 13: Metadata entry size[2,3]

The size of each metadata entry associated with a file. This includes the size of the file name.

Byte 14 to 15: Max file name size[2]

This is the max size of a file name for each file and it is fixed. It must be less than (metadata entry size - 10) bytes.

Byte 16 to 19: State section size

The number of pages that the state section is comprised of.

Byte 20 to 29: File information for the hash entries file

Bytes 20 to 25 is the size of the file and bytes 26 to 29 is the address of the root index cluster.

Byte 30 to 39: File information for the metadata file

Bytes 30 to 35 is the size of the file and bytes 36 to 39 is the address of the root index cluster.

3.2 State Section

A bit in the state section indicate the status of a cluster. A bit with the value of 1 indicates that a cluster is free and a bit with the value of 0 indicates that a cluster is in use. The location of a bit correlates to the location of a cluster on the storage device. The first free bit is always the next cluster to be allocated. The size of this bit vector is determined by the number of clusters that the storage device has.

3.3 Directory Structure

The directory manages the metadata for files along with the structure for looking up files. TEFS only supports a single directory - the root directory. The first file that is part of the directory structure contains hashes of the file names. The second file that makes of the directory is the file that contains the metadata for files. The location of a hash in the hash file corresponds to the location of the metadata entry in the metadata file. For example, if it is the 9th hash in the array of hashes in the hash file, that corresponds the the 9th metadata entry in the metadata file. A metadata entry for a file contains the following fields:

Byte 0: Status flag

The status indicates if the file exists or has been deleted.

Byte 1 to 4: File size - end of file page

[1]The value is stored as the exponent where the base is 2. For example, if the value is 8 (2^3), the value 3 is stored.

[2]This is a value that is specified by the user during formatting.

[3]The value must be a power of two.

The last page in the file.

Byte 5 to 6: File size - byte in end of file page

To have the file size in bytes instead of just pages, the byte within the last page of the file is also stored.

Byte 7 to 10: Root index cluster address

The is the pointer to the root index cluster of the file.

Byte 11 to N: File name and user metadata

Contains the name of the file padded until the max file size is reached. The remaining is optional user metadata.

3.3.1 Hash Algorithm

The algorithm used for hashing is the DJB2a hashing algorithm [Las16]. The distribution of the hashes are poor compared to a hashing algorithm such as Murmur2 or Murmur3 but collisions are rare [Boy12]. Since randomness is irrelevant to the purpose of the hashes in the directory structure, DJB2a was chosen because the code size is much smaller than most hash functions and it is fast [Boy12]. A hash can be either 2 or 4 bytes. The DJB2a algorithm is designed to return a 4 byte hash so a modification had to be implemented to be able to return a 2 byte hash; the remainder of the 4 byte hash is divided by a large prime number just less than 2^{16}.

3.4 Formatting Operation

The format function is currently only available on device. The function required the user to specify the fields as shown in Section 3.1.

Any device less than 2TB (assuming that the page size is 512 bytes or greater) is compatible with TEFS since a 4 byte integer stores the page addresses (2^{32}-1 pages are allowed). On format, depending on the number of pages that the device has, the size of an address is set to either 2 or 4 bytes automatically. That is, if the storage device has less than 2^{16} pages, the size of an address is 2 bytes, otherwise the size of an address is 4 bytes.

The size of a hash can be set to either 2 or 4 bytes. A hash size of two bytes is beneficial if only a small number of files will be created. This speeds up the lookup for files in the directory since less page reads are needed.

The user also specifies the metadata entry size for files as well as the max size of the file names that will be stored in the metadata. The remaining space in the metadata entry can be used by the user to store custom metadata such as timestamps.

The remaining fields are the cluster size and page size. They are specified just like with FAT.

It is important to note that most fields must be a power of two since the file system was optimized around the values being a power of two.

The format function stores these user defined values in the information page. Based on the size of the device, page size, and cluster size, the size of the state section is determined and created. Lastly, the two files for the directory structure are created.

3.5 Open, Close, and Remove Operations

When opening or removing a file, the file name passed into the open function is hashed. The hash entries file is scanned linearly for the same hash. If the hash is found, the file name in the metadata entry that corresponds to the position of the hash is checked against the file name passed into the open function. The file is found if the file names match and then the file is opened or removed. If the hashes matched and the file names did not match, the rest of the hashes are scanned and this goes on until the file is found or the end of the hash entries file is reached.

For creating a file, it is similar to opening a file. The user passes in the file name to the open function and the open function will create the file if the file is not found. If the user does not wish to create a file on open, they can check if the file exists using the exists function which checks the directory for the file. When creating the file, the first deleted hash entry is kept in memory if found during scanning the hashes and if it is a new file being created (no other files exist with the same name), the hash for the new file will be inserted into that spot and the corresponding metadata in the metadata file is created. Otherwise, the hash is appended to the the end of the hash entries file and the corresponding metadata is appended to the end of the metadata file.

When creating a file, there are a few things that have to be done. The status in the metadata entry is set to in-use, the end of file page is set to 0, the byte in the end of file page is set to 0, the root index cluster address is written to the entry, and the file name and metadata is also written to the entry. Before the root index cluster address is written, the root index

cluster has to be reserved in the state section. After it is reserved, the first data cluster is reserved and the address of the data cluster is written to the first byte in the root index cluster.

If the file already exists and is opened, the end of file page and the end of file byte within that page is loaded into memory along with the root index cluster address. If the file has child index clusters, the address of the first child index cluster address is read from the root index cluster into memory. Finally, the first data cluster address is read into memory from the root index cluster or the first child index cluster if the child index cluster exists.

When removing a file, the file is first found in the directory otherwise an error is returned. The hash entry is marked as deleted and the status in the metadata entry is also marked as deleted. Next, all of the index and data clusters in the file have to be released. This can be a long process for large files.

3.6 Read and Write Operations

On creation, a file only requires a root index cluster. As data is appended to the file, new data clusters are reserved and their addresses are added to the root index cluster. When the root index cluster is filled up, it becomes the first child index cluster and a new root index is reserved. When a new child index clusters is needed, a new cluster is reserved and its address is added to the root index cluster. The clusters do not need to be pre-erased when being reserved because the end of the file is kept track of so garbage data is overwritten.

When reading or writing to a file, the data cluster address is found in either the root index cluster or a child index cluster. In either case, the correct location for reading or writing is calculated directly based on the file offset location. TEFS will also detect if the page being read or written is the same as the previous read/write request to prevent unnecessary page reads. Page data is inserted into a page buffer in memory and is not flushed until a different page is read, the file is closed, or a flush forced.

The file size is written to the device when a flush or close occurs. It is optional to write the file size out after a new page is being written out in the file or any new data has been appended to the file. The size of a file is tracked in the file's directory entry by keeping track of the last page and last byte within the page. The max file size is limited by

$$\frac{c^3}{a^2} \tag{3.1}$$

bytes where c is the cluster size in bytes and a is the address size in bytes. Using equation 3.1, a device that has a page size of 512 bytes, 4 byte addresses, and a cluster size of 32KiB has a max file size of 2TiB.

3.7 File Allocation Table Versus TEFS Index Structure

For random reads and writes, the FAT file system requires traversing a linked list of data cluster addresses to find the cluster for a specified location in a file. In the case where the addresses of a cluster chain are in sequential order, the number of page reads is

$$\left\lceil \frac{n}{(\frac{p}{a})c} \right\rceil \tag{3.2}$$

where n is the location, in bytes, to read or write to in a file, and p is the page size in bytes. The worst case is when there are multiple files and fragmentation occurs in a way that causes each cluster address for a file to be on a different page in the FAT. In this case, the number of page reads is

$$\left\lceil \frac{n}{c} \right\rceil \tag{3.3}$$

when the FAT is not cached in memory.

Suppose there is a 10MB file stored on a device with the FAT file system, the device does not cache the FAT, the size of a cluster is 32768 bytes, the size of a page is 512 bytes, the address size is 4 bytes, and the file location pointer is at the beginning of the file. If the last byte is to be read from the file, the best case is 3 page reads and the worst case is 306 page reads.

TEFS will always guarantee at most 2 index page reads for a read or a write to any location in a file because a page in the root index may be read and then a page in the child index may be read (if there are any child indexes) before finding the data cluster. In Figure 2.1, the cluster chain requires at least 120 page reads to traverse to the end of the file, but in Figure 3.1 only 2 page reads are needed. The number of page reads for FAT will increase as a file grows in size.

TEFS has been adapted to work on NOR Serial Dataflash [Ade15] that can read bytes directly. This allows for faster traversing of the indexes for both FAT and TEFS. The number of byte reads required for FAT will be

$$\left\lceil \frac{n}{c}a \right\rceil \tag{3.4}$$

to find the address for the data cluster. As for TEFS, it will be at most $2a$ byte reads.

3.8 TEFS C File Interface

The C file interface makes it easy for existing applications to work with TEFS. Also, the user does not have to be concerned with page boundaries as the interface takes care of it. As a note, only a subset of the C interface is implemented. There are functions such as getc or ferror that are not implemented. All of the functions are prefixed with a "t_" to avoid conflicts with the functions defined in stdio.h. Here are the implemented functions with the corresponding C function definition below each:

```
int8_t t_fclose(T_FILE *fp);
int fclose(FILE *fp);

int8_t t_feof(T_FILE *fp);
int feof(FILE *fp);

int8_t t_fflush(T_FILE *fp);
int fflush(FILE *fp);

int8_t t_fgetpos(T_FILE *fp, fpos_t *pos);
int fgetpos FILE *fp, fpos_t *pos);

T_FILE *t_fopen(char *file_name, char *mode);
FILE *fopen(const char *file_name, const char *mode);

size_t t_fread(void *ptr, size_t size, size_t count, T_FILE *fp);
size_t fread(void *ptr, size_t size, size_t count, FILE *fp);

int8_t t_fseek(T_FILE *fp, uint32_t offset, int8_t whence);
int fseek(FILE *fp, long int offset, int whence);

int8_t t_fsetpos(T_FILE *fp, fpos_t *pos);
int fsetpos(FILE *fp, const fpos_t *pos);

uint32_t t_ftell(T_FILE *fp);
long int ftell(FILE *fp);

size_t t_fwrite(void *ptr, size_t size, size_t count, T_FILE *fp);
size_t fwrite(const void *ptr, size_t size, size_t count, FILE *fp
    );

int8_t t_remove(char *file_name);
int remove(const char *file_name);
```

```
void t_rewind(T_FILE *fp);
void rewind(FILE *fp);
```

Chapter 4

Experimental Results

Experiments were done on an Arduino Uno [Ard16] with a 16GB UHS I Sandisk Micro SD card. The Arduino Uno is an 8-bit micro-controller with 2KB of RAM and 32KB of code space. The comparison was done between the TEFS page interface, the TEFS C file interface, and two popular FAT libraries - the Arduino SdFat library [Bil16] and FatFs [Cha11]. The page size for the card was 512 bytes, and the cluster sizes were set to 64 pages (32KiB). The results were an average of 5 runs with the exception of the file create, open, and remove benchmarks for FAT. Each of these benchmarks required formatting between them so it would be too time consuming to repeat the benchmarks 5 times.

The results can vary depending on the size of a cluster but 64 pages is the default size on Windows when formatting a drive with FAT so it is the cluster size used in the tests. It is important to note that different SD cards cause different results based on their class and other factors. Even if the same card is used between benchmarks, the results for time might be slightly different because of the card's FTL. The first set of benchmarks considers the number of page reads and writes to avoid this issue; the page reads and writes are a large factor in the time that a file operation takes. The second set of benchmarks is the time it took to complete them in milliseconds.

4.1 Number of Page Reads and Writes

The first set of tests measured the number of page reads and writes to the storage device. The TEFS C file interface calls the underlying TEFS page interface; therefore, the number of reads and writes are the same for these two interfaces.

Table 4.1 shows the number of page reads and writes when 1000 pages of data were written out to a file at different record sizes. The results were different for 1 to 511 byte records and 512 byte records because when the record size was 512 bytes, the pages did not have to be buffered first since the record size was the same as the size of a page. TEFS required more

page reads and writes, as shown in Table 4.1, for sequential writing since it maintains the cluster state bit vector.

Table 4.1: Sequentially Write 1000 Pages: 1 to 511 Byte Records on the Left and 512 Byte Records on the Right

File System	**Page Reads**	**Page Writes**
TEFS	31	1030
Arduino SdFat	17	1017
FatFs	17	1019

File System	**Page Reads**	**Page Writes**
TEFS	31	1030
Arduino SdFat	2	1003
FatFs	2	1005

Table 4.2 shows the number of page reads required when sequentially reading 1000 pages of data. There were 16 page reads required to get the addresses for the data clusters for Arduino SdFat since there were 16 data clusters. This is similar for TEFS and FatFs, but they cache the first data cluster address on file open so TEFS would have 17 (due to the root index cluster) and FatFs would have 16. Only 1 byte records were used since Arduino SdFat only supports single byte reads at a time.

Table 4.3 demonstrates having a staggered cluster chain such that each address in the cluster chain is on a different page in the FAT for a file 10MB in size. When traversing the cluster chain, a page is read for each data cluster address. Figure 4.1 demonstrates the architectural issue of FAT when files grow in size. Once a file gets larger than 1MB, many page reads are needed to traverse to the correct data cluster in the file. TEFS levels off at 1 page read for the file sizes shown. Larger files may require 2 page reads as there are more child index clusters.

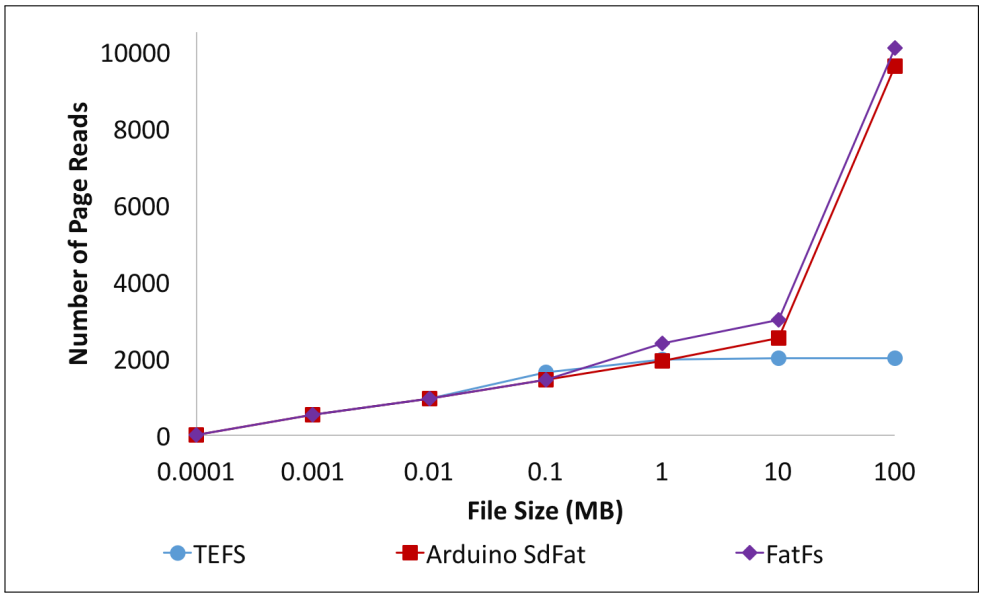
Table 4.2: Sequentially Read 1000 Pages

File System	**Page Reads**
TEFS	1015
Arduino SdFat	1016
FatFs	1015

Table 4.3: Randomly Read 1000 Bytes From 10MB File With Staggered Cluster Chain

File System	**Page Reads**
TEFS	1999
Arduino SdFat	96069
FatFs	96069

Figure 4.1: Read or Write 1000 Bytes at Random Locations



When opening, creating, or removing a file, TEFS reads and writes more pages than FAT. This is due to the root index cluster that has to be created and the index clusters that have to be traversed. These numbers can be seen in Tables 4.4, 4.5, and 4.6 for a single file.

TEFS Version 1 is TEFS with the old directory structure. It was a fixed sized directory that just did linear search on the directory entries. The sizes of the directory entries were 32 bytes. TEFS Version 2 has the current directory structure as explained earlier. In Tables 4.4, 4.5, and 4.6, the benefit of the new directory structure is that there are far less page reads as seen for 100 and 1000 files. There are slightly more page writes for creating and removing files because the hash entries and metadata files require clusters to be reserved and the file size is saved to the metadata entries of these two files periodically. The effect of the extra page reads and writes for creating or removing a file in TEFS makes creating or removing many files less efficient than FAT despite the improvements in the new directory structure of TEFS. Opening a file in TEFS starts to become more efficient than FAT as more files are in the directory due to less page reads when finding a file.

Table 4.4: Create Files

	Page Reads				**Page Writes**			
File System	**1 File**	**10 Files**	**100 Files**	**1000 Files**	**1 File**	**10 Files**	**100 Files**	**1000 Files**
TEFS Version 1	7	52	766	35754	6	51	501	5001
TEFS Version 2	9	63	597	9350	6	60	600	6000
Arduino SdFat	0	0	71	6359	1	2	6	189

Table 4.5: Open the Last File When There Are N Files

	Page Reads			
File System	**1 File**	**10 Files**	**100 Files**	**1000 Files**
TEFS Version 1	2	2	8	64
TEFS Version 2	3	3	3	10
Arduino SdFat	0	0	1	12

Table 4.6: Remove Files

	Page Reads				**Page Writes**			
File System	**1 File**	**10 Files**	**100 Files**	**1000 Files**	**1 File**	**10 Files**	**100 Files**	**1000 Files**
TEFS Version 1	4	31	648	34736	3	30	300	3000
TEFS Version 2	7	70	700	10416	4	40	400	4000
Arduino SdFat	0	0	71	6359	1	2	20	200

4.1.1 Time Benchmarks

The TEFS implementation is optimized to use bit shifts and bit masks instead of modulo and division operations. It also reduces the number of function calls as much as possible. This makes TEFS more CPU efficient which is important on embedded devices. Figure 4.2 and Table 4.7 show that sequential read and write times for TEFS are 10% to 20% faster than FAT implementations even though the page reads and writes were slightly more for TEFS. Figure 4.2 shows that FatFs takes more time to write records that are larger in size compared to Arduino SdFat. It also shows TEFS takes less time, for all record sizes, when writing as compared to Arduino SdFat. Figure 4.3 shows the times for reading 1000 bytes at random locations for different file sizes. The same property of the tree index structure is prominent as displayed by the times.

Table 4.7: Sequentially Read 1000 Pages of Data With a Record Size of 1 Byte

File System	**Time (ms)**
TEFS	13118
TEFS C file interface	17918
Arduino SdFat	19110
FatFs	18672

The times creating and removing files in Tables 4.8, 4.9, and 4.10 match the trends seen with the number of page input and output operation.

4.2 Library Sizes

The library sizes and memory usage of the file systems include the size of the file system and the code to communicate with the SD card. The text size for TEFS is marginally less than the FAT implementations (Table 4.11). Part of this is due to the simplification of features like the single directory. Slightly more SRAM is used since the metadata for the hash entries files and the metadata file are stored in memory. Also, slightly more memory is used by each file since they store pointers to the root index, child index, and data clusters to speed up multiple reads and writes in the same location of the file.

Figure 4.2: Sequentially Write 1000 Pages With Varying Record Sizes

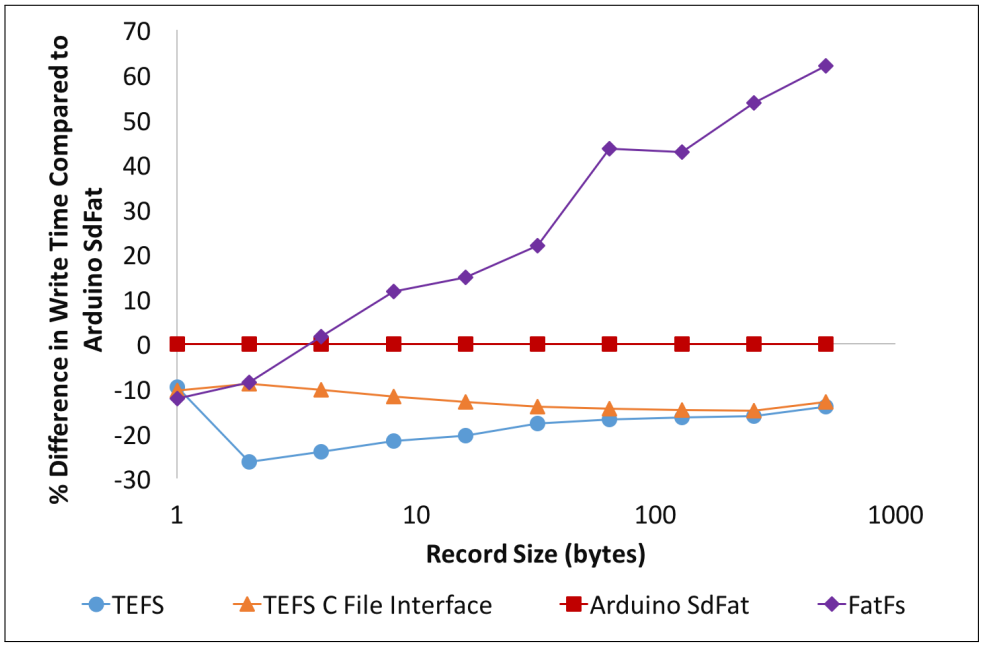


Table 4.8: Time in Milliseconds to Create Files

File System	1 File	10 Files	100 Files	1000 Files
TEFS Version 1	25	203	4052	93180
TEFS Version 2	29	251	2561	44703
Arduino SdFat	1	7	273	20058

Table 4.9: Time in Milliseconds to Open the Last File When There Are N Files

File System	1 File	10 Files	100 Files	1000 Files
TEFS Version 1	3	4	16	145
TEFS Version 2	5	5	8	44
Arduino SdFat	0	0	4	38

Table 4.10: Time in Milliseconds to Delete Files

File System	1 File	10 Files	100 Files	1000 Files
TEFS Version 1	18	120	2263	75760
TEFS Version 2	20	211	2251	40123
Arduino SdFat	0	6	264	19353

Figure 4.3: Read 1000 Bytes at Random Locations

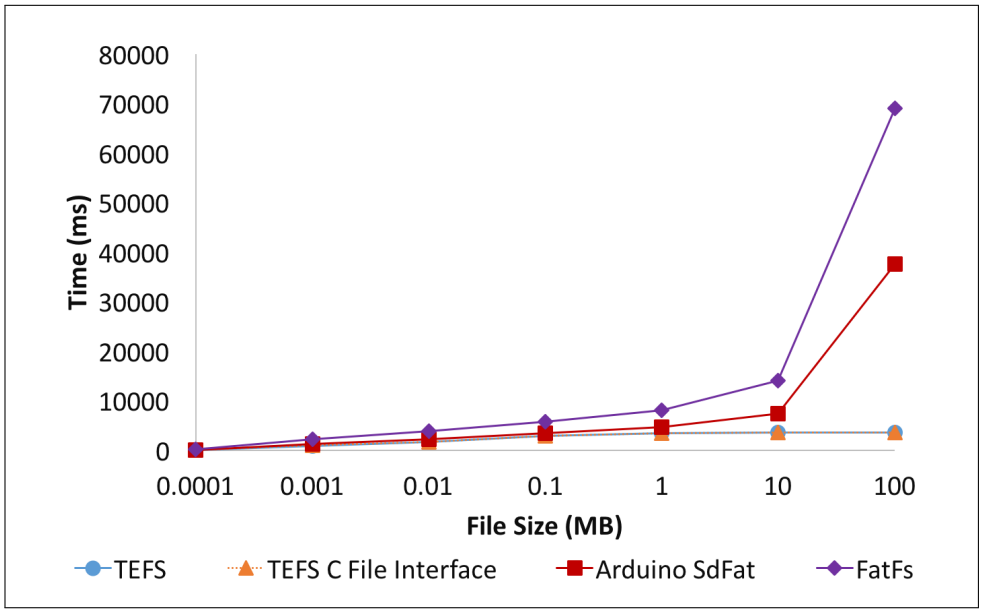


Table 4.11: Library Sizes in Bytes

File System	**Text Size**	**Dynamic Memory**	**Memory per File**
TEFS	10364	647	34
TEFS C file interface	12260	683	41
Arduino SdFat	14752	608	27
FatFs	14879	584	36

Chapter 5

Analysis of Trade-offs

The architectural advantage of TEFS over FAT is representing the index structure as a tree rather than a linked list of entries. Using a tree guarantees a small constant number of index page reads to find data in the file. This consistency is very important for embedded devices. From an implementation perspective, TEFS is optimized to minimize CPU usage, so even though it has a slight increase in page I/Os for sequential reads/writes, its time performance is better.

There are a few trade-offs for using TEFS compared to FAT. TEFS has a larger index overhead for each file. For small files, the root index cluster (32 KiB using the same cluster size as before) will be relatively empty. In comparison, this file would be represented by a single address entry in FAT. For example, if the size of a cluster is 32KiB, that would incur nearly a 64KiB overhead for a file because of the root index cluster and data cluster. For FAT, the overhead would be nearly 32KiB for the data cluster. If there are a large number of small files, more space is used by TEFS than FAT. The index overhead also leads to more reads and write when creating, opening, or removing files. However, seeking to the end of a large file with TEFS only takes at most 2 page reads as compared to FAT where it must seek the linked list to get the end of the file. In the case when opening existing files in append mode, it takes more page reads to do this for FAT. Also, TEFS is comparable on open times when there are lots of files because of the improved directory structure. Otherwise, if applications create and remove files frequently, this takes more time for TEFS and is something to consider when choosing TEFS. In summary, FAT performs better when the are few files, smaller files, and files are created and removed frequently. TEFS, however, performs better when there are more files, larger files, and files are opened and closed often but not created and removed as often.

FAT has the option to have multiple File Allocation Tables and this allows for some redundancy. Currently, TEFS has no method of redundancy to be able to recover from corrupted data or to prevent further data corruption. TEFS does allow file sizes to be written out often which is something that is not implemented in Arduino SdFat. Without this, the device could

crash and there could be data that is in the file but the size of the file does not reflect this data. To the file system, this means that the data does not exist and has not been written.

A disadvantage of FAT is that the largest file size allowed for FAT32 is 4GB. The reason for this is that the file size is stored as a four byte unsigned integer in the directory entry and thus can only represent $2^{32} - 1$ bytes. The limit for the max file size in TEFS is determined by the size of the clusters and pages. If the files need to be larger, the cluster size can be changed to allow for this. Both FAT and TEFS allow the size of the storage device to be in the terabyte range. For TEFS it is 2TB when the page size is 512 bytes and it is likely the same for FAT implementations.

TEFS has many options when formatting. It allows for the size of metadata in a directory entry to be as large as the size of a page as long as the size of the metadata entries are a power of 2. The file names can be (the size of the metadata - 10) bytes. FAT can support long file names and FatFs is an implementation that does but Arduino SdFat cannot. It only supports 8.3 (11 character) file names. TEFS also allows for custom metadata specified by the user.

FAT32 needs a storage device that is 32MB or larger. TEFS adapts to any size of storage as long as it has enough pages for the information page, state section, and index clusters. The page size also has to be at least 64 bytes in size.

FAT supports multiple directories but the directory TEFS only has a root directory and, therefore, folders are not allowed. The intention is that files on device can be organized by name and folders are not necessary for most intended applications. This was a trade-off to decrease the code size and the extra files needed for storing the extra directories.

Finally, due to FAT being ubiquitous, it is supported in the Microsoft Windows, Mac OS X, and Linux operating systems. TEFS currently does not support these platforms.

Chapter 6

Conclusion

6.1 Future Improvements

Further improvements can be made to TEFS to include additional features. FAT provides some redundancy with the multiple file allocation tables but TEFS does not have any method for redundancy. A way to prevent corruption of data is to have multiple information blocks that are the same and keep checksums of the critical data needed for the file system to make sure that the data is correct.

Both Arduino SdFat and TEFS support one buffer. FatFs allows for multiple buffers so for larger devices, this could be beneficial in speeding up some operations of the file system and would be a good future addition.

The TEFS C file interface does not have every feature of the regular C file operation set. Adding the rest would allow for wider audience to support TEFS.

Some of these improvements might defeat the purpose of having a tiny file system as they would increase the code size. However, they could be made optional at compile time and therefore be beneficial for devices with more memory but also be adapted to work on memory constrained devices.

Future work that does not add to the code size includes supporting reading and writing to the storage device on common operating systems. This would be beneficial for the user since they could easily get the data from the storage device as there is no easy way to do that currently. Another improvement would be to make a formatter script available on major operation systems as the only way to format a storage device is to use it the formatter on the micro-controller.

6.2 Summary

TEFS demonstrates that file systems with a linked list index structure, such as FAT, for micro-controllers with constrained resources are not efficient when randomly reading or writing to large files. TEFS implementation is

optimized for these types of devices to reduce the number of CPU cycles as this affects the time to read and write. It is a small, efficient file system that is faster than popular FAT library implementations designed for embedded devices and significantly better when randomly reading or writing in larger files.

As for an application of TEFS, TEFS will be used as the underlying file system for the LittleD relational database [DL14] and IonDB key-value store [FHD+15] for embedded systems. This key-value store has different data-structures that write to persistent storage and for some of these structures, there is a benefit of having fast random reads and writes.

Bibliography

[ADAD15] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015. → pages 3

[Ade15] Adesto Technologies. DataFlash [online]. Dec 2015. → pages 12

[Ard16] Arduino. Arduino UNO & Genuino UNO [online]. 2016. → pages 15

[Bil16] Bill Greiman and SparkFun Electronics. SD Library for Arduino [online]. 2016. → pages 15

[Boy12] Boyd, Ian. Which hashing algorithm is best for uniqueness and speed? [online]. 2012. → pages 9

[Cha11] ChaN. FatFs - Generic FAT File System Module, 2011. → pages 15

[CTT15] Rmy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem [online]. Nov 2015. → pages 1

[DL14] Graeme Douglas and Ramon Lawrence. LittleD: a SQL database for sensor nodes and embedded applications. In *Symposium on Applied Computing*, pages 827–832, 2014. → pages 25

[DNH04] Hui Dai, Michael Neufeld, and Richard Han. ELF: An Efficient Log-structured Flash File System for Micro Sensor Nodes. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 176–187. ACM, 2004. → pages 1

[FHD+15] Scott Fazackerley, Eric Huang, Graeme Douglas, Raffi Kudlac, and Ramon Lawrence. Key-value store implementations for Arduino microcontrollers. In *IEEE 28th Canadian Conference on*

Electrical and Computer Engineering, pages 158–164, 2015. → pages 25

[FL11] S. Fazackerley and R. Lawrence. A flash resident file system for embedded sensor networks. In *IEEE 24th Canadian Conference on Electrical and Computer Engineering*, pages 1400–1405, May 2011. → pages 4, 5

[Kos16] Kostka, Grzegorz. lwext4 [online]. 2016. → pages 1, 2

[Las16] Lassonde School of Engineering. Hash Functions [online]. 2016. → pages 9

[LP06] Seung-Ho Lim and Kyu-Ho Park. An efficient NAND flash file system for flash memory storage. *IEEE Transactions on Computers*, 55(7):906–912, July 2006. → pages 1

[Mic00] Microsoft Corporation. Microsoft EFI FAT32 File System Specification. Whitepaper, Dec 2000. → pages 1, 3, 5

[Mic06] Micron Technology Inc. NAND Flash 101: An Introduction to NAND Flash and How to Design It In to Your Next Product. Technical report, Micron Technology Inc., 2006. → pages 1

[MRR14] Keshava Munegowda, GT Raju, and Veera Manikandan Raju. Directory Compaction Techniques for Space Optimizations in ExFAT and FAT File Systems for Embedded Storage Devices. 2014. → pages 1

[Mul14] MultiMedia LLC. Using the Memory Technology Device (MTD) [online]. 2014. → pages 1

[Phi01] Daniel Phillips. A directory index for ext2. In *Annual Linux Showcase & Conference*, 2001. → pages 4

[TDHV09] Nicolas Tsiftes, Adam Dunkels, Zhitao He, and Thiemo Voigt. Enabling large-scale storage in sensor networks with the coffee file system. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 349–360. IEEE Computer Society, 2009. → pages 1

[Tec13] Technical Committee: SD Card Association. SD Specifications Part 1: Physical Layer Simplified Specification. Technical Report 4.10, SD Group, 2013. → pages 1

[Woo01] David Woodhouse. The Journalling Flash File System. In *Proceeding of Ottawa Linux Symposium*, volume 200, 2001. → pages 1

[ZBT09] Aviad Zuck, Ohad Barzilay, and Sivan Toledo. NANDFS: A Flexible Flash File System for RAM-constrained Systems. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 285–294, New York, NY, USA, 2009. ACM. → pages 1