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

Wade Penson
Department of Computer Science
University of British Columbia
wpenson@alumni.ubc.ca

Scott Fazackerley
Department of Computer Science
University of British Columbia
scott.fazackerley@alumni.ubc.ca

Ramon Lawrence
Department of Computer Science
University of British Columbia
ramon.lawrence@ubc.ca

***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 simplified 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.**

I. Introduction

Embedded devices typically utilize flash based storage for persistent data [1]. 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 [2]. 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 [3]. File systems such as Ext2 [4] and FAT [5] 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 [6], YAFFS [7], NANDFS [2], and Coffee [8]. 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 [9]. 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 [8].

This work presents TEFS, a **T**iny **E**mbedded **F**ile **S**ystem, 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 FAT [10] [11] 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.

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 Section II, different types of flash memory and the FAT file system are introduced. Section III covers the design of TEFS, its functionality, and the properties that distinguish it from FAT. Section IV presents experimental results, and Section V is a discussion of the trade-offs of TEFS and FAT. The paper closes with future work and conclusions.

II. 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 [12]. Another part is the directory, and it is usually a list of file names and inode address pairs. 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 [5]. 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 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.

NAND flash devices are page addressable and not byte addressable [13]. 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 [13], 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 [5]. However, a subset of embedded devices and micro-controllers have a limited amount of RAM [13] and are not capable of caching the complete index for a file in memory. TEFS was designed to perform well on these such devices.

Fig. 1. Representation of a 500MB File by the FAT32 File System

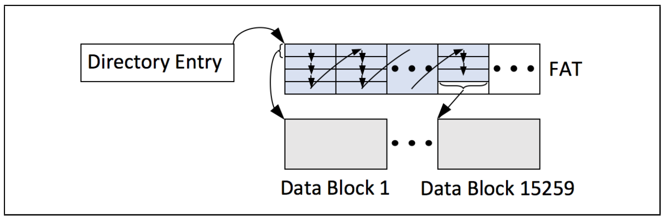
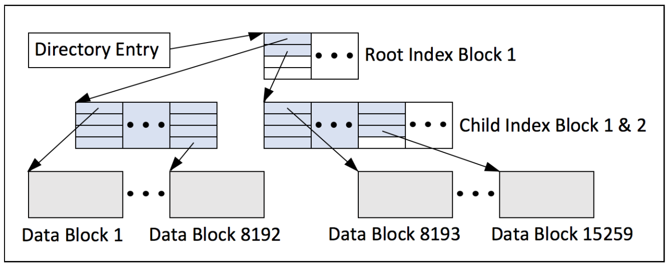


Fig. 2. Representation of a 500MB File by TEFS



III. 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 four 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. A bit of value 1 indicates that a cluster is free and a bit of 0 indicates that a cluster is in use. The location of a bit correlates to the location of a cluster on the device. The size of this bit vector is determined by the number of data clusters that the device has. The third section is the directory for the files. TEFS currently only supports a single root directory with a static size that is specified during formatting. Each file has a directory entry that contains information about the file including the file name, the size of the file, and the pointer to the root index cluster for the file. The file name is consistent with the 8.3 convention; the name is 8 characters and the extension is 3 characters. The last part of the file system is the tree index structure for each file. It consists of a single root index cluster, child index clusters, and data clusters (Figure 2). The root contains a sequence of addresses that either point directly to data clusters (for smaller files) or child index clusters that point to data clusters. Addresses can either be 2 or 4 bytes. 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.

A. Open, Close, and Remove Operations

When opening a file, the directory is scanned for the file. If the file is found, the existing metadata will be read from the directory entry. If the file is not found, it will be created. In this process, a cluster for the root index is reserved. A directory entry is also created.

B. 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 is created. As new child index clusters are 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.

When reading or writing to a file, the data cluster address is found either in the root index cluster or a child index cluster. In either case, the correct entry is calculated directly based on the file offset location. TEFS will detect if the page being read or written is the same as the previous read/write request. 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 tracked in the file's directory entry. The max file size is limited by

$$\frac{c^3}{a^2} \tag{1}$$

bytes where c is the cluster size in bytes and a is the address size in bytes.
Using equation 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.

C. 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

$$\left\lceil \frac{n}{(\frac{p}{a})c} \right\rceil \tag{2}$$

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

$$\left\lceil \frac{n}{c} \right\rceil \tag{3}$$

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 read (if there are any child indexes) before finding the data cluster. In Figure 1, the cluster chain requires at least 120 page reads to traverse to the end of the file, but in Figure 2 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 [14] 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

$$\left\lceil \frac{n}{c} a \right\rceil \tag{4}$$

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

IV. Experimental Results

Experiments were done on an Arduino Uno [15] 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 storage. The comparison was done between the TEFS page interface, the TEFS C file interface, and two popular FAT libraries - the Arduino SdFat library [16] and FatFs [17]. 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.

A. 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 I 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 I, for sequential writing since it maintains the cluster state bit vector.

Table II 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 as Arduino SdFat only supports single byte reads at a time.

Table III demonstrates having a staggered cluster chain such that each address in the cluster chain is on a different page in

TABLE I
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 II
SEQUENTIALLY READ 1000 PAGES

File System	Page Reads
TEFS	1015
Arduino SdFat	1016
FatFs	1015

TABLE III
RANDOMLY READ 1000 BYTES FROM 10MB FILE WITH STAGGERED CLUSTER CHAIN

File System	Page Reads
TEFS	1999
Arduino SdFat	96069
FatFs	96069

TABLE IV
FILE OPERATIONS

	Page Reads			Page Writes		
File System	Open	Create	Close	Open	Create	Close
TEFS	3	9	5	0	7	4
Arduino SdFat	0	0	0	0	1	1
FatFs	0	0	0	0	1	1

the FAT for a file 10MB in size. When traversing the cluster chain, a page is read for each data cluster address.

Figure 3 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.

When opening, creating, and closing a file, TEFS reads and writes more pages than FAT. This is due to the root index cluster that has to be created and traversed. The results can be seen in Table IV.

Fig. 3. Read 1000 Bytes at Random Locations

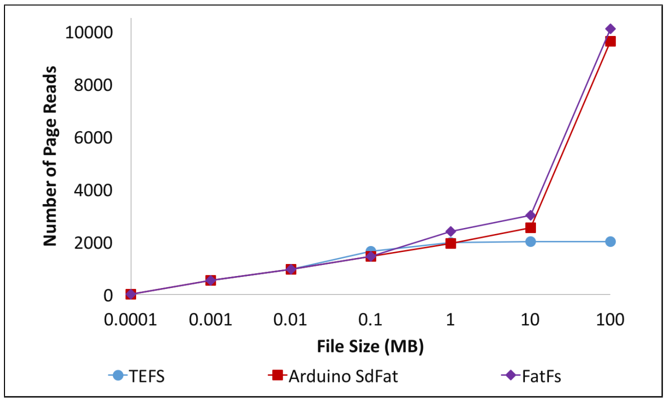


TABLE V
SEQUENTIALLY READ 1000 PAGES OF DATA WITH A RECORD SIZE OF 1 BYTE

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

TABLE VI
LIBRARY SIZES IN BYTES

File System	Text Size	Dynamic Memory	Memory per File
TEFS	9884	559	28
TEFS C file interface	14259	559	35
Arduino SdFat	14752	608	27
FatFs	14879	584	36

B. 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 and Table V 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 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 5 shows the times for reading bytes at random locations.

C. 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 VI), and it uses less RAM.

Fig. 4. Sequentially Write 1000 Pages With Varying Record Sizes

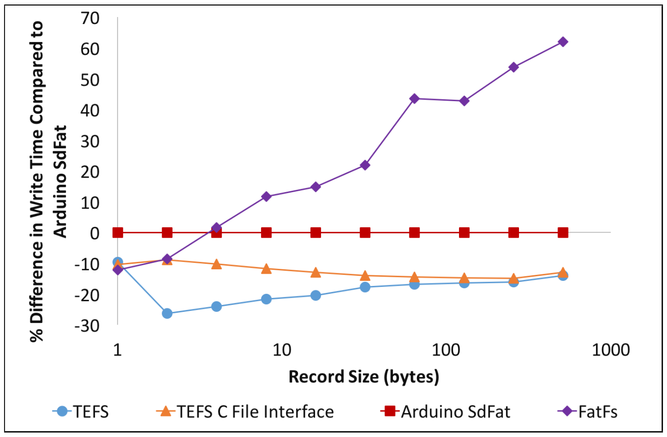
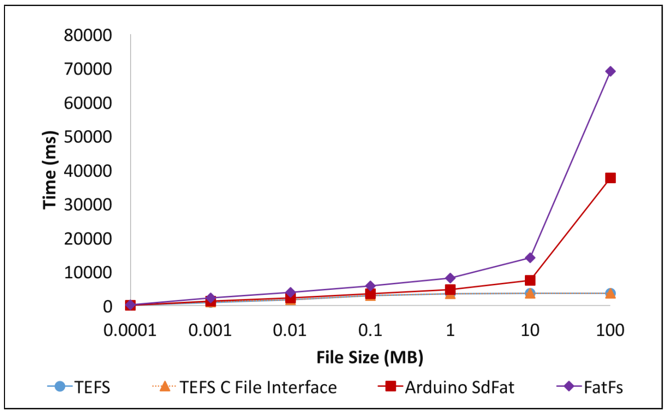


Fig. 5. Read 1000 Bytes at Random Locations



V. 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) will be relatively empty. In comparison, this file would be represented by a single address entry in FAT. If there are a large number of small files, more space is used by TEFS than FAT. With TEFS, there is an overhead to manage the index clusters leading to more reads and write when creating, opening, and 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. Otherwise, if an application creates, opens, and closes files frequently, this is to be considered when choosing to use TEFS.

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.

VI. Conclusion

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.

Further improvements can be made to TEFS to include additional features. Future work would be to create an improved directory and support long file names in a manner that would produce the smallest code footprint in an efficient way for embedded devices. Also supporting reading and writing to the storage device on common operating systems would be beneficial for the user so that they can read the data from the storage device directly. We are planning to use TEFS as the underlying file system for the LittleD relational database [18] and IonDB key-value store [19] for embedded systems.

Acknowledgment

The authors thank NSERC for supporting this research.

References

[1] Micron Technology Inc., "NAND Flash 101: An Introduction to NAND Flash and How to Design It In to Your Next Product," Micron Technology Inc., Tech. Rep., 2006.

[2] A. Zuck, O. Barzilay, and S. Toledo, "NANDFS: A Flexible Flash File System for RAM-constrained Systems," in *Proceedings of the Seventh ACM International Conference on Embedded Software*, ser. EMSOFT '09. New York, NY, USA: ACM, 2009, pp. 285–294.

[3] MultiMedia LLC. Using the Memory Technology Device (MTD). [Online]. Available: http://www.stlinux.com/howto/Flash/MTD

[4] R. Card, T. Ts'o, and S. Tweedie. (2015, nov) Design and Implementation of the Second Extended Filesystem. [Online]. Available: http://e2fsprogs.sourceforge.net/ext2intro.html

[5] Microsoft Corporation, "Microsoft EFI FAT32 File System Specification," Whitepaper, dec 2000, retrieved from https://msdn.microsoft.com/en-us/windows/hardware/gg463080.aspx October 2015.

[6] D. Woodhouse, "The Journalling Flash File System," in *Proceeding of Ottawa Linux Symposium*, vol. 200, no. 1, 2001.

[7] S.-H. Lim and K.-H. Park, "An efficient NAND flash file system for flash memory storage," *IEEE Transactions on Computers*, vol. 55, no. 7, pp. 906–912, July 2006.

[8] N. Tsiftes, A. Dunkels, Z. He, and T. 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*. IEEE Computer Society, 2009, pp. 349–360.

[9] H. Dai, M. Neufeld, and R. 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*, ser. SenSys '04. ACM, 2004, pp. 176–187. [Online]. Available: http://doi.acm.org.ezproxy.library.ubc.ca/10.1145/1031495.1031516

[10] Technical Committee: SD Card Association, "SD Specifications Part 1: Physical Layer Simplified Specification," SD Group, Tech. Rep. 4.10, 2013.

[11] K. Munegowda, G. Raju, and V. M. Raju, "Directory Compaction Techniques for Space Optimizations in ExFAT and FAT File Systems for Embedded Storage Devices," 2014.

[12] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 0th ed. Arpaci-Dusseau Books, May 2015.

[13] S. Fazackerley and R. Lawrence, "A flash resident file system for embedded sensor networks," in *IEEE 24th Canadian Conference on Electrical and Computer Engineering*, May 2011, pp. 1400–1405.

[14] Adesto Technologies. (2015, dec) DataFlash. [Online]. Available: {http://www.adestotech.com/products/data-flash/}

[15] Arduino. Arduino UNO & Genuino UNO. [Online]. Available: {https://www.arduino.cc/en/Main/ArduinoBoardUno}

[16] W. Greiman and SparkFun Electronics. SD Library for Arduino. [Online]. Available: https://github.com/arduino/Arduino/tree/master/libraries/SD

[17] ChaN, "FatFs - Generic FAT File System Module," 2011.

[18] G. Douglas and R. Lawrence, "LittleD: a SQL database for sensor nodes and embedded applications," in *Symposium on Applied Computing*, 2014, pp. 827–832. [Online]. Available: http://doi.acm.org/10.1145/2554850.2554891

[19] S. Fazackerley, E. Huang, G. Douglas, R. Kudlac, and R. Lawrence, "Key-value store implementations for Arduino microcontrollers," in *IEEE 28th Canadian Conference on Electrical and Computer Engineering*, 2015, pp. 158–164. [Online]. Available: http://dx.doi.org/10.1109/CCECE.2015.7129178