RIFFS: Reverse Indirect Flash File System

Marcelo Trierveiler Pereira and Antônio Augusto Fröhlich and Hugo Marcondes

Federal University of Santa Catarina, PO Box 476
88049-900, Florianpolis - SC, Brazil
{trier,guto,hugom}@lisha.ufsc.br,
http://www.lisha.ufsc.br~{trier,guto,hugom}

Abstract. This project presents a new technique for flash storage management called a *Reverse-Indirect Flash File System* (RIFFS). However, flash memories have a drawback: its data cannot be updated-in-place. To solve this limitation, the data is stored inside of the proper file. The solution was to construct a reverse-tree. This would be impracticable with the current systems, because it would not be possible to locate a file directly, from root directory. This schema would break the navigability of the system, and then a direct tree need to be constructed in RAM memory. This article shows the reverse-tree management schema to solve the limitations of flash memories. This solution helped to minimize extreme updates and write operations, increasing flash life-time.

Keywords: operating systems, embedded systems, flash memory, file systems.

1 Introduction

The current embedded systems are used in a large number of applications, due to microprocessors technology advancement, the integrated circuits, and these devices control system. Usually, when these circuits are referred, we first remember of our personal computers and its processors chips (also called CPU), forgetting about other equipments around us that also use them. According to Tennenhouse [2], only 2% of 8 billions processors made in 2000 were used in personal computers (PC). The great majority of them were used in embedded systems.

Each kind of dedicated system needs the use of specific software to its own control and configuration. This software may have simple functions (executing just routines) or complex ones (executing memory, task management functions). This software is also called *firmware*.

With the appearance of new embedded systems, there was the necessity of flash memories with enough storage capacity to accommodate control softwares, configuration files, user data, files that are constantly updated, etc. To achieve all these requirements a file management system was required. This system implementation in flash memories is not so trivial, due to its peculiarities. This project presents a different storage structure from current flash memory file systems. The directory system is based on *reverse trees* and the file system is based on a new and different structure, called *file context*.

Next section shows the disadvantages of flash memories and some concepts to manage them efficiently. Section 3 presents the RIFFS system, its architecture, project and modules implementation. Section 4 describes some tests realized with the first prototype and comment the test results. Section 5 is the conclusion of this article.

2 Flash File Systems

This section presents flash memory features, and the most common ways to manage them efficiently. Among its characteristics related to file management [1], there are two that can be emphasized as more relevant: *actualization* and *deletion numbers*. Its possible to write a byte, in any flash *clean* position, but this data can only be rewritten after an eraser operation. The eraser operations, in turn, are achieved in entire section, and not in individual bytes. A sector in a flash is a contiguous bytes block, and the manufacturer defines its size and its position inside of the memory. Each sector has a maximum deletion number, and since the maximum number is exceeded, the manufacturer does not guarantee the data integrity.

Flash Management: The files systems are usually implemented in two different ways: *entirely developed* [3], or *constructed inside of a software layer* [4] to access the device (also know as *driver*), keeping compatibility with the existing file systems superior layers. One very important thing in the flash file system characteristics is the concern in preventing deletion in each updating, due to the fact that it consumes a long time and reduces flash lifetime. The *remapping* mechanism [5] is used to supply this expectation. This remapping method causes fragmentation inside the sectors due to invalid data, thus needing a *cleaning procedure* [6, 7] to delete them later afterwards.

Sector Cleaning: The sector erasing strategy, reorganizing its valid data in somewhere else, is called *sector cleaning*, and its procedure is known as *garbage collector*. According to Chiang [8], the choice of cleaning policies has a great impact in the system performance, being able to reduce application efficiency until 50%. The cleaning policies have three different basic aspects: the *segment selection* that is going to be cleaned, the *data reorganization*, and the *garbage collector start routine*. This last one may be in three different ways: by *determined time*, by *flash utilization percentage*, or by a *system priority low routine*, that is always doing this work.

3 Design Rational of RIFFS

Inside this project, the file concept is defined as a data set. This data can be used for internal control, such as directory tree control, or can be used as a simple stored data. According to these project requirements, it was necessary

to keep all control information regarding the file inside it. To obtain this characteristic, the RIFFS project created a special structure called *file context*. The file management, the storage device management and the directory management are shown.

File Management Within the file management there are basically three structures: the *file* (itself), the *file context* and the *logical data blocks*.

File: Each file contains a *list of blocks*, a *type* and a *file context* structure. The file context is responsible for adding control information, as for example the file name. The second attribute stores all blocks of data belonging to the file (in case they exist), and their respective sizes. The type attribute classifies the file in the system as: *user file* or *system file* (directory). Figure 1-b shows an example of how the file is seen when loaded in the RAM memory. While in the figure 1-c it is possible to see the blocks of the file spreaded over the flash memory. In 1-a, a view of the file as a *data set* is shown. In this figure, the circle represents the file name `file1.txt` which contains blocks of data identified in the example as `f_1`, `f_2`, and `f_3`.

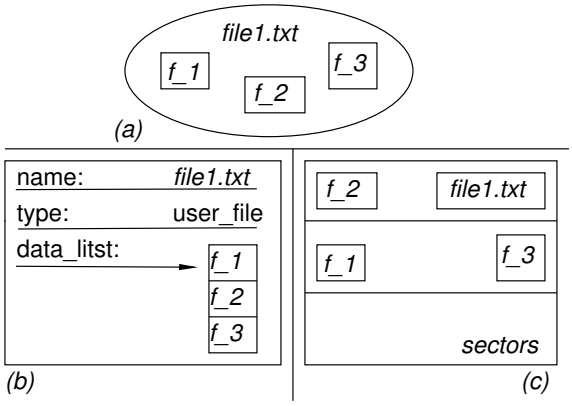

Fig. 1. (a) Logic view. (b) RAM view. (c) Flash view

File context: All control information relevant to the file is within its context. The context is implemented as a block inside the flash, and for that reason, it contains the same logical block attributes, described in 3. It contains the following information: a reference to the *father context*, and the name of the file. The context also could contain the file attributes to which it belongs such as: *creation date*, *users control*, etc[1]. The reference for the father context is used in

[1] It is important to emphasize that it was decided not to implement these controls in this project, being left for future implementations

the management of directories, and they are explained next. The name of the file is an array of characters and stores the name chosen by the user.

File blocks organization: Each logical block of a file contains a version that identifies the various parts of a file. Thus, sorting the list of logical blocks ascending, we have the organized data of the file (more details on the field *version* of each logical block - section 3). This was necessary to allow file blocks that, are written randomly in any part of the device, guaranteeing its reconstitution in the start up of the system. The figure 1-a shows two types of file blocks `file1.txt`: *user blocks*, and a *context type* block. As user blocks, there are: `f_1`, `f_2`, and `f_3`, and the context type block is represented by the file name: `file1.txt`.

Storage device management As a storage device an AMD flash memory (AM29D163CxT) was used, with *2Mbytes* of storage capacity. This device does not have *physical blocks*, since it manipulates bytes in any position of the memory. That is an advantage since its data is accessed randomly, always within the same time (pre-defined by the manufacturer), which does not occur with the hard disks (because of their nature).

Internal Architecture: The physical architecture of the device was focused in the sector, and not inside the flash. In this work it contains a structure shown in figure 2. Inside it, it's possible to visualize three basic structures: *data structure* (found in the data area), *control structure* (found in the control area) and a *header* (found in the beginning of each sector).

The first structure can be understood as the files own data, and represents the logical blocks of variable size. They are recorded from the beginning to the end of the sector, as shown in figure 2. These structures are named `raw data`. As foreseen, the size of this data can vary as much as one might wish, as long as it does not exceed the maximum value of the sector. In case this occurs, the file system records the data remaining in another place in the flash. The second structure relates to the control of the `raw data`, called `data control`, and is recorded from the end to the beginning of the sector (figure 2). The `control structures` have a fixed size, so that the initial reading method of the system can be fast and efficient. The third structure, the `header`, contains control information to the system as for example, the number of sector deletions. Each one of these structures is shown in details next:

`Data control`: They are structures used in the volume mount to improve the system loading performance. They are always found in the end of the sectors, and are recorded from the end to the beginning. These structures contain references to the `raw data` inside the sector. The fields of this structure are: *size*, `offset`, `version`, `identification` and `type`. The `size field` indicates the amount of bytes that the `raw data` occupies. The `identification field` is the logical number of the file, also called `file id`, which the `raw data` belongs. This number is unique for each file inside the file system. The `field offset` is

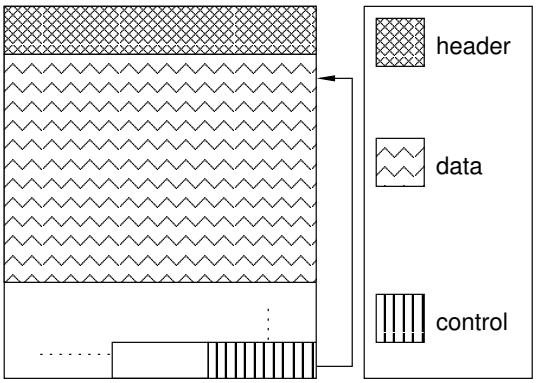

Fig. 2. Sector Architecture.

the place inside of the sector where the `raw data` is stored. The `version` is the corresponding number to the part inside the `raw data` which the file belongs. This field is necessary, to distinguish the diverse parts of a file, in case they exist. In case the file is physically in different parts of the flash, these parts will have the same `file id`, but `different versions`. The `field type` classifies the `raw data` to which this structure represents. It can classify the `raw data` in three types: `user data`, `log context` and `context`, as it can be seen in the next item.

`Raw data:` Structure that contains its data in accordance with the field `type` of the `data control` (to which it belongs). The types can be: `user data`, `log context` and `context`. The `user data` is the data itself, recorded from the user. The `log context` is used in case the data is updated. The `context` can be related to the union of a directory entry with a file descriptor. This last structure contains the fields: `file name`, and a `logical identifier` of the superior branch, called `father id`. This father identifier is responsible for the characteristic of the *reverse tree*. It's important to emphasize that this `raw data` structure corresponds to the logical block of variable size data.

`Sector header:` Structure found in the beginning of each flash sector, that contains the fields: a `magic number`, an *identifier* (called `sector id`), and the *sector deletions number* (called `erased no`). The `sector id` field is a logical number attributed to the sector when formatting the memory, and is responsible for differentiating the sectors of a file system operating with more than one flash memory. The `deletions number` is used by the garbage collector and by the allocation method, so that all sectors are equally worn out. It's important to highlight that a sector is said empty when it contains only the header. These structures are manipulated in accordance with the software modules, shown in section 3.1.

Logical blocks: The concept of variable size logical blocks was adopted for this project. As the concept of physical blocks are non-existent on flash memories, the implementation of variable size blocks become simpler. The logical block size is limited by the sector size where it is inserted. In this project the name given to the logical block was `raw data`.

Free blocks management: Because flash memories work with the deletion by sector, the free blocks management also follows this approach. A list of empty sectors is kept in the main storage area. The first item of this list corresponds to the sector with the smallest number of deletions to guarantee that the sectors are equally consumed. This list is sorted by the deletions number. When a request arrives, the amount of space required is allocated inside the first sector. If there isn't enough space, only the free existent space is returned. Details of free blocks management implementation can be found in subsection 3.1.

Directory management The architecture of directly linked tree is proposed by the conventional file systems for a link of the structures recorded in the flash. Inside this architecture, the directories contain information about the files and subdirectories (in case they exist). For example, in the *Journaling Flash File System* (JFFS2) [9], this concept is so used, that to erase a file, invalidating its reference in the directory is enough. This also occurs in conventional file systems, such as *Unix Fast File System* (UFFS) [10]. Following this storage method, the directories are created with a size pre-defined by the system. If on one hand some file belonging to the directory does not exist, a space waste exists, and on the other hand, if some file exists, and the update rate is high, there is also a waste of invalid references.

This project proposes one directory system stored in a *reversed linked tree*. The structures belonging to this tree contains all the information concerning itself, eliminating direct references in the system. However, the navigability of a reversed tree is not adequate for a file system, and for this reason, it is necessary to construct it (conventionally - not reversed) in the RAM memory.

The directory is implemented as a file, and thus it contains all its attributes, such as: a `context`, one `list of files`, and a `type` (more details on file, see section **??**). Because the directory is implemented as a special type of file, it contains all attributes of a normal file. Its context is identical to the one explained previously, with the same attributes. This way, each directory has a reference to its father, and thus successively, characterizing a reverse tree.

Figure 3 shows three types of directories views. Figure 3-b shows an example of how the directory named `dir1` is seen when loaded in the RAM memory. Figure 3-a shows a logical view of the reverse tree of context. It is possible to notice that the context of the files `file1.txt` and `file2.txt` has as *father context* the directory `dir1`. Figure 3-c shows the content of the three contexts presented, organized randomly in the flash memory.

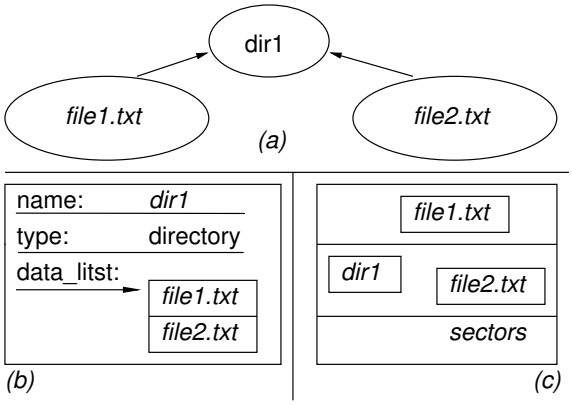

Fig. 3. (a) Logic view. (b) RAM view. (c) Flash view.

3.1 Implementation

This project contains seven modules for the file system, see figure 4. These modules are: *Flash Controller*, *Device Manager*, *Allocator*, *Garbage Collector*, *Scanner*, *File Manager*, and *Directory Manager*, explained next.

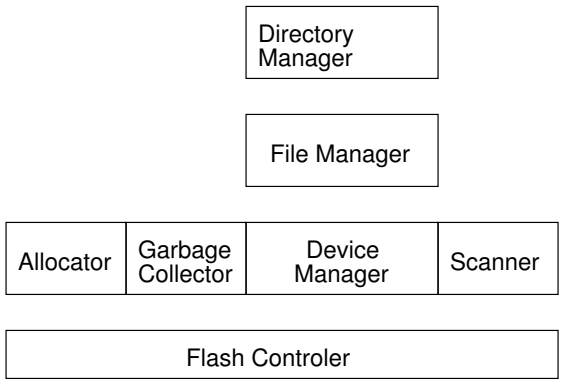

Fig. 4. RIFFS module architecture.

Flash Controller: This module is responsible for the implementation of the basic routines for flash memory manipulation. This way it is possible to make all the management code compatible with the other modules of the file system for several memories available in the market, requiring only the implementation of this layer.

Scanner: This layer is used in the initialization of the system, and later destroyed. Its main function is to construct a list of Data_Control that will give support to the other modules in their initialization.

Device Manager: This module is responsible for implementing data reading and data writing in accordance with the RIFFS semantics. This component can also be understood as the connection between the physical structures architecture (found in the flash) and the logical structures architecture (found in the RAM).

Allocator: Layer responsible for the management of the free space in the flash. It contains a list of clean sectors, ordered by the number of deletions. The algorithm used for implementing space allocation is *first-fit*. This way, in case there is a space request, this module always allocates the first free space inside of the first sector of the list.

Garbage Collector: Because of the flash characteristics, when some data needs to be updated, it is invalidated and rewritten in another place. Thus, inside of the device there might be residues of updated data which, later needs to be extinguished. This module is the one responsible for the management of the invalid data for the system. The garbage collector has got two lists: `erased list` and `invalid list`. The first list is composed of objects that represent the header of the sector, and ordered by the field `erased number`. The second list is composed of objects that represent the data of a sector, and ordered by the amount of invalid data. The list `erased list` is used when it is desired to discover the destination of valid data of a sector, which is being recycled. The list `invalid list` is used to select the sector that has the largest amount of invalid data that can then be recycled. The first element of the list `invalid list` is the sector that will be chosen to be recycled, when the method `start` of garbage collector is invoked. According to the literature the method for choosing the sectors *origin* and *used destination* in this work is called *Greedy*.

File Manager: This Module is responsible for grouping the several parts of a file (in case they exist) in structures of the type `file`. After its initialization a list of files is constructed and it will serve for `directory manager module` initialization, explained next.

Directory Manager: This module is responsible for constructing and manipulating the direct tree in the main memory of the system. In the initialization of the system, this module runs through the list of `file manager` files and then constructs a *direct tree* in the RAM. The directory is implemented as a file of the type `system file`. The first directory is called root and its name is the character ‘‘/’’. This directory is not recorded in the flash, but is build on RAM memory on file system startup. It has a standard identifier equal to one. This way, all files that contain the field `father id` equal to one, belong to the root directory, and then they are added to the list of files of the directory. The same occurs with the subdirectories, and thus successively.

4 System Results

For each file system a different platform was used. This was the solution found because a JFFS2 version for the RIFFS platform could not be found and vice versa. Also for that reason it was necessary to find some way to make the tests of these two systems in different platforms compatible. The solution was to find a *percentage* or a *time ratio* of data writing in a flash (through simple functions given by the manufacturer), by the time of data writing through the file system. Next, the tests platforms and calculations are shown in detail.

RIFFS platform: The test platform used in this project is named KMB01. The KMB01 has a *PowerPC 405GP of 200MHz* processor, by IBM. In the processor bus the following itens are connected: the flash memory, the RAM memory, besides other components not mentioned for not being part of this work objective. The flash memory is connected to the processor through a bus of 16 bits and its size is 2MB. The RAM memory size in the system is 16MB.

JFFS2 platform: The platform used to make the JFFS2 system tests was one Compaq *iPaq H3600*. The H3600 has a 206MHz *StrongArm SA-1110* processor, by Intel. The RAM memory and the flash memory are connected in the processor bus. The flash memory is connected to the processor through a bus of 32 bits, and its size is 16MB. The RAM memory size is 32MB. The JFFS2 tests were run on Linux operating system, and for this reason, it is known that there is a scheduler and some processes of kernel that cannot be disactivated. This way the tests will not reflect with *accuracy*, these two systems comparison, but even so, it will serve as base for a more advanced analysis.

Data writing tests: The accomplishment of these tests is based on the percentage of lost system performance in relation to the simple flash operations. This scheme has a simple formula explained in the following: it assumes the Writing Time in the flash performed by Simple functions, without intervention from system modules, called SWT, and suppose the Writing Time of the same amount of bytes through a file system, called FWT. Dividing the *File Writing Time* (FWT) by *Simple Writing Time* (SWT) a percentage of the consumption of additional processing imposed by the system is obtained. This percentage of performance was necessary because the tests were done in two different platforms. The times results are described as follows:

RIFFS tests time: The *Simple Writing Time* (SWT) for the KMB01 platform flash was acquired in the following way: firstly, 1MB buffers were written one hundred times and its respective times were calculated, then it was found SWT = 7,307456 seconds as time average for 1MB buffers.So, to find the average time spent to write 1kB buffers the result was divided per 1024. It was found SWT = 7,13619 mili seconds as time average for 1kB buffers.

The *File Writing Time* (FWT) for RIFFS system, has six values. Each value is calculated in accordance with the tests with different buffers sizes for writing.

The sizes chosen were: *16kB, 32kB, 64kB, 128kB, 256kB* and *512kB*. The time for each buffer size was obtained in the following way: firstly, perform the function `append` one hundred times for each buffer size (for each time the plataform was reseted) and then, calculate the average time spent on these tests. The calculated times for each buffer, are shown in the table 1.

JFFS2 tests time: The *Simple Writing Time* (SWT) for the iPAQ H3600 platform flash was acquired in the following way: firstly, 12MB buffers were written one hundred times and its respective times were calculated and then, it was found SWT = 53,557377 seconds as time average for 12MB buffers. So to find the average time spent to write 1kB buffers the result was divided per 12288 (12kB). It was found SWT = 4,43989 mili seconds as time average for 1kB buffers. The FWT for the JFFS2 was obtained in the same way as the FWT for the RIFFS (shown above), except that it was run in the iPAQ H3600 platform. The table 1 shows the results:

Buffer size	**RIFFS FWT (msec)**	**JFFS2 FWT (msec)**
16KB	114	120
32KB	236	220
64KB	501	450
128KB	965	860
256KB	1930	1600
512KB	3739	2900

Table 1. RIFFS File Writing Time (FWT)

Comparing times: After the measurement tests, the formula FWT/SWT was used to calculate the percentage, shown in the table 2.

Buffer size	**RIFFS FWT/SWT**	**JFFS2 FWT/SWT**
16KB	0,46%	68,92%
32KB	3,66%	54,85%
64KB	9,70%	58,37%
128KB	5,70%	51,33%
256KB	5,67%	40,77%
512KB	2,35%	27,57%

Table 2. RIFFS and JFFS2 comparing times.

5 Conclusions

This article presented the objectives of RIFFS (Reverse Indirect Flash File System), a file system for flash memories, focused mainly in the characteristics of these memories. The efforts are concentred in simple structures written in flash memories.

The first system prototype was builded as a functions library. Because this library has not external dependences, this project is easily portable for other platforms. In other hand, there is a *negative* point: the file management module does not support *hard links*.

The results of write performance are very satisfactory. The tests on garbage collector could not have been made in the practical one. It happens because there was not tools enough to isolate the JFFS2 garbage collector from entire system. Theoretically RIFFS is faster, due to this operations simplicity. In the file write test for 16kB buffers, RIFFS obtained the best result in comparison to JFFS2.

References

1. Grossman, S.: Future trends in flash memories. In: Proceedings of the 1996 IEEE International Workshop on Memory Technology, Design and Testing (MTDT'96), Singapore, IEEE (1996)
2. Tennenhouse, D.: Proactive Computing. Communications of the ACM **43** (2000) 43–50
3. Wu, M., Zwaenepoel, W.: envy: a non-volatile, main memory storage system. In: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems, ACM Press (1994) 86–97
4. Kawaguchi, A., Nishioka, S., Motoda, H.: A flash-memory based file system. In: USENIX Technical Conference, New Orleans, LA, USENIX Assoc. (1995) 155–164
5. Rosenblum, M., Ousterhout, J.K.: The design and implementation of a log-structured file system. ACM Transactions on Computer Systems (TOCS) **10** (1992) 26–52
6. Chang, L.P., Kuo, T.W.: A real-time garbage collection mechanism for flash memory storage system in embedded systems. In: The 8th International Conference on Real-Time Computing Systems and Applications (RTCSA 2002), Tokyo, Japan, Keio University, Mita Campus (2002)
7. Mei-Ling Chiang, Paul C.H. Lee, R.C.C.: Cleaning policies in mobile computers using flash memory. The Journal of Systems and Software **48** (1999) 213–231
8. Chiang, M.L., Lee, P.C.H., Chang, R.C.: Managing flash memory in personal communication devices. In: IEEE International Symposium on Consumer Electronics (ISCE'97), Singapore (1997) 177–182
9. David Woodhouse Red Hat, I.: JFFS: The Journalling Flash File System. Red Hat. (1998)
10. McKusick, M.K., Joy, W.N., Leffler, S.J., Fabry, R.S.: A fast file system for UNIX. ACM Transactions on Computer Systems (TOCS) (1984)