POFS

Plain-Old Flash Filesystem

Embedded Device Filesystem

Requirements

Table of Contents

1Introduction....2
1.1Why Flash....2
1.2What format....2
1.3What density, what outlook....2
1.4Constraints....2
1.4.1Erase concept....2
1.4.2Limited number of erases....3
1.4.3Fixed erase block size....3
1.4.4Dead blocks....3
1.5The State-of-the-Art....4
2Main features....4
2.1UFS concepts....4
2.2Crash proof....4
2.3Journaling....4
2.4Allocation strategy in function of erase cycles experienced....4
2.5Moving allocation table....4
2.6Block descriptor....4
2.7File states cycles....4
2.8Compression....5

1 Introduction

Embedded systems are traditionally devices with limited resources and cost-driven specifications. Their feature set is usually targeted at a simple function, or at least a reduced set. In many cases, the form factor is also a significant criterion. This leads to carefully thought designs to keep space, electrical power and price under control.

This arena has long been reserved to dedicated operating systems, when at all. With the emergence of low-power, high-performance processors, the pictures has changed significantly and desktop operating systems like Linux or *BSD are commonly found. The main design challenge resides in the hardware support. One of these is the filesystem that supports the application. This document draws the guidelines of a simple filesystem compatible with “plain-old Flash”, i.e. the type of device that usually holds the BIOS of a PC.

1.1 Why Flash

Flash memories are now common in the market place. Their technology is derived from the old EPROMs, that required UV to reset its whole content to 1s, what does not permit a very flexible management. With time, these memories were replaced by electrically-erasable models, called EEPROMs. Although this was an improvement, the memories were fairly expensive so a trade-off was designed, called Flash. These are lower priced and pemit to erase (set content to 1s) blocks of predefined size. Their management is somewhat more complex than EEPROMs but usually offer a shorter write time.

Flash are reasonlably low-cost, at least affordable for high-volume consumer appliances, and lately offer sizes that can hold a complete filesystem. Their main interest is a compromise between fast erase, block-oriented management and in-system self-reprogramming.

1.2 What format

Flash devices are used in a variety of appliances: USB pendrives, digital cameras Flashdisks, PC BIOS ROM, digital answering machines, etc. Each has its own target applications and own features. A pendive has a PC-like filesystem with a USB bus. Flashdisks have a PC-like filesystem with an IDE controller. PC BIOS ROM holds a plain application, digital answering machines hold a very basic (single directory, no filename) filesystem, etc.

To reduce cost in an embedded system, it may be of advantage to take profit of the boot ROM to hold a filesystem: although this binds the Flash to the device (the “disk” cannot be swapped for another one), it may help drive the cost down by having a single ROM device instead of 2. In this case, the boot sector can be used to store a mini BIOS.

1.3 What density, what outlook

Flash memories can be obtained from various manufacturers: Intel, AMD, Samsung, Micron, just to name a few. Todays densities commonly amount to 1GB and the outlook is to exceed 10GB by 2010, what could create a need for more-than-32bits filesystem from day 1.

1.4 Constraints

Running a filesystem on such a device poses severe restrictions, which will be summarized here.

1.4.1 Erase concept

An erased device is filled with 1s. In other words, you can write a 0 on a 1 but not a 1 on a 0.

To write a 0 on a 1, you can simply read the byte, modify the content and write it back.

To write a 1 on a 0, you must read the whole block (typically several KB), modify the bit, erase the whole block and write back – or omit the erase and write elsewhere but then you have to keep track of where you've put it. In other words, you will have to play the musical chairs.

The conclusion for the filesystem architecture is that, as much as possible, you should preserve areas that may be modified later.

Regarding the file content, there is not much you can do: the application rules the game.

1.4.2 Limited number of erases

Flash devices are not eternal: every write and erase cycle causes stress on the silicon junctions and, eventually, the bits will start sticking. The only way to preserve this is to initiate write and erase cycle only when necessary, delay them as much as possible. A typical figure for the maximum number of erases is between 100k and 1M.

The maximum number of write/erase cycles is defined per cell: with a careless design, you could perfectly get to a situation where you end up selling the device on eBay with the tag: “99% brand new” - meaning “the 10 bytes I needed are stone dead”.

The first candidate for these “10 bytes” is the file/inode allocation table – which I would recommend to have cruising through the device on each update since it will be the area suffering most changes. Another hint is to have some space for “updates”: when you modify a file, you will have to update the “modified” field in the file descriptor. Erasing a whole table just to update 1 record may not be a very viable route. If your allocation table resides in memory (which it should do to optimize performance), there is no harm in having a base table and a list of updates, apart from the wasted space.

To ensure homogeneous stress, the filesystem should keep track of the number of erase cycles of each block.

The filesystem designer will have to decide which information is important and which can be left aside: UNIX filesystems traditionally offer the “last accessed” time, which is easy to manage on a hard-disk but causes constraints ona Flash-based system.

1.4.3 Fixed erase block size

Erase blocks usually have prohibitive sizes to use them as single storage entities – for instance, 128KB on the Micron MT29F2G08AABW. On the other hand, common allocation units on spinning disks is in the order of 512 bytes to 4KB.

This means that more than 1 file will have to be allocated to a block. It also means that erasing a file will leave a hole in the middle. After some time, the filesystem will look like Gruyere cheese (the one with the holes all over the place) and data will have to be consolidated. This process is commonly called “garbage collection”.

Garbage collection should be issued at “quiet” times of the system as the device cannot be read while it is being written or erased.

Another constraint of garbage collection is the need for a free block: the memory have to report itself as 100% full before allocating the last block otherwise the garbage collection may fail to free space and cause the appliance to crash.

A strategy will have to be defined to group files that are likely to never be touched – or all together.

1.4.4 Dead blocks

Cannot be written or erased properly.

1.5 The State-of-the-Art

This initiative is not the first one: several projects started with the same goal in mind. Some names: LFS, JFFS/JFFS2, YAFFS.

This section needs to be expanded to describe their features, strengths, weaknesses, license, etc.

2 Main features

2.1 UFS concepts

- file descriptors
- Allocation

2.2 Crash proof

Can recover from powerfail without pain

2.3 Journaling

2.4 Allocation strategy in function of erase cycles experienced

2.5 Moving allocation table

2.6 Block descriptor

- Free
- Allocation table
- Files/Directories
- Boot sector

2.7 File states cycles

To be finalized: free, initialized (directory not yet written), active (directory up to date), hanging (content has been moved and directory points to new location), dirty (flagged for garbage

collection).

2.8 Compression

To save space and help increase capacity.