



Yaffs Robustness and Testing

Charles Manning

2017-06-07

Many embedded systems need to store critical data, making reliable file systems an important part of modern embedded system design. That robustness is not achieved through chance. Robustness is only achieved through design and extensive testing to verify that the file system functions correctly and is resistant to power failure.

This document describes some of the important design criteria and design features used to achieve a robust design. The test methodologies are also described.

Table of Contents

1 Background.....	2
2 Flash File System considerations.....	3
3 How Yaffs achieves robustness and performance.....	5
4 Testing.....	8
4.1 Power Fail Stress Testing.....	8
4.2 Testing the Yaffs Direct Interface API.....	9
4.3 Linux Testing.....	9
4.4 Fuzz Testing.....	9
4.5 Software checking.....	9
4.6 Open Source Usage.....	10
4.7 Universities.....	10
5 Conclusions.....	10
6 References.....	11

1 Background

Cheaper CPUs and flash have driven up embedded system functionality. These increased functions often require file system storage.

The original flash file storage mechanisms were the use of a flash translation layer (FTL) – a driver model which makes flash memory appear as a disk drive – in conjunction with a disk-oriented file system such as FAT. This storage methodology has various performance and robustness limitations, leading to the development of file systems designed specifically for flash memory. These are known as flash file systems.

Flash memory has various limitations when compared with a disk. For example, flash memory pages cannot be individually re-written but instead the whole block must be erased and rewritten with the modified page. Typical FTL achieve this with various logical to physical mapping schemes. Thus emulating disk-like behaviour on flash memory adds an extra layer of software which slows down write performance. Worse still, it adds extra state which is prone to corruption due to power failure.

A typical FTL-based solution will use a file system such as FAT. FAT always stores tables in the same disk blocks and must thus change the same blocks often. Since flash memory blocks can endure a limited number of write/erasure cycles, the FTL must move the physical location of the block around to prevent the high traffic blocks from wearing out. This requires wear levelling – a further burden.

Some systems try to mitigate against the performance penalties by caching a lot of data to reduce writes. While this can go some way towards reducing the apparent write time, it does make the solution prone to data loss in the event of a power failure.

As well as limitations, flash memory also has certain advantages when compared to rotating media. There is no spin up time, making it more responsive. There is no read head so there is



no read penalty if data is fragmented. Flash file systems can be designed to exploit these advantages.

True flash file systems¹, on the other hand, are a single code body designed to take the features and limitations of flash into consideration. Flash file systems can thus be far more “flash friendly”. Well designed flash file systems will avoid using “flash unfriendly” techniques such as allocation tables and in-place rewriting.

Yaffs is a flash file system, originally designed specifically for NAND flash though also used with great success on NOR flash too. Yaffs development started in 2001 when 32Mbytes of flash was considered large. Since then, Yaffs has been expanded to work properly with different flash types and far larger memory arrays (many GBytes).

Yaffs was originally deployed in Linux, but was written in an OS neutral way, mainly to facilitate modular development and testing. This turned out to be useful when we had requests to port Yaffs to other operating systems. Since then, Yaffs has been used in many different products in a wide variety of applications from consumer products such as cell phones and sewing machines to aerospace, industrial and medical applications including process control, building management, communications infrastructure and many others.

Yaffs is not tied to any endian ordering and has been used on a wide variety of architectures including ARM, MIPS, PowerPC, x86, some DSPs. Yaffs has been deployed using a wide variety of compilers from many different vendors.

Nor is Yaffs tied to any particular operating system. Yaffs has been used with Linux, eCOS, Windows Embedded Compact (previously Windows CE), VxWorks and others.

2 Flash File System considerations

There are many criteria that are important when selecting a flash file system. These are some that our customers have found to be most important:

- **Robustness:** It really is not worth having a file system that loses or corrupts data and prevents the system from working properly. Some flash file systems are only robust when synced (flushed) after files have been written. That requires extra code, slows the system and leaves the file system in a non-deterministic state until the sync has been completed. Some file systems need to perform disk repair operations (eg. check disk) in the event of a power failure. This can add considerable time to the start up, preventing the system from being operational.
- **Performance:** In many embedded system designs, slow read/write performance can delay tasks and cause system degradation or even failure. It is thus important that read/write performance be acceptable. Note that many flash file systems need to perform extra actions such as garbage collection. In some cases these actions can cause the file system to stall for a long time. It is thus important that these factors be taken into consideration. Performance can sometimes be improved by adding a write

1 Note that flash translation layers (FTLs) are erroneously referred to as flash file systems. For example, despite the name, M-Systems TrueFFS is not a flash file system but a flash translation layer (FTL).



caching layer. These caching layers make the file system calls (write etc) proceed quickly, without actually writing the data to flash. This is a two-edged sword as data in the cache will be lost on a power failure and syncing the file system will take a long time. Thus, most flash file systems offer either speed or robustness – not both.

- **Proven:** It is very easy for a file system vendor to provide a list of claims saying “100% power safe” or similar, but what evidence do they have to back this up? All file systems, including flash file systems, are complex bodies of software with extremely complex state. They need very significant testing to prove that they work correctly.
- **Portable:** Software is a huge investment and it is increasingly important to be able to port the software between OSs and CPU architectures. Most flash file systems are limited to a single operating system which makes it difficult to migrate software between different platforms. In Yaffs' case portability extends to being able to seamlessly migrate the core Yaffs code into a test framework which enhances testing.

3 How Yaffs achieves robustness and performance

Yaffs was designed from the ground up specifically to work well with NAND flash. The author had already developed another flash file system, so was already familiar with the design challenges of flash.

As Yaffs is not just adapting a disk-oriented file system to work with flash it is not surprising that Yaffs is very different in design to most other file systems. The key difference is that Yaffs is a log-structured. file system².

In essence, a log structured file system writes file system changes as a sequential log. The log structure is particularly suitable for various reasons including:

- All writes are at the end of the log. When a file is modified, there is no need to erase and rewrite a part of the flash. Changes are just appended to the log. Thus there is no need to erase and copy old data just to change some existing pages in a file. This tends to make writes much faster. There is no need to do a lot of caching to achieve performance. Data can be written to the log immediately meaning that sync time is very low. Data robustness is improved dramatically.
- There are no allocation tables or similar. This reduces the amount of data being written to the flash, and that there are no tables to corrupt.
- Since all writes are to the end of the log there are no “high traffic” blocks. This means Yaffs only needs a simpler wear leveling strategy.
- In the event of a power failure, the file system state is easily re-created from the log. This means there is no need to perform disk repair operations to correct for an unclean shutdown.

2 This document only provides a brief overview of the Yaffs architecture and design. The “How Yaffs Works” document provides an in-depth explanation of the Yaffs design and architecture.



So why don't we see many log structured file systems for disk file systems? The answer to this is that log file systems tend to spread the writes around on the media. That is typically very problematic for mechanical storage which must physically move the read/write head from one location to another making the access time very slow. Of course that does not apply for flash memory. By designing specifically for flash, Yaffs can ignore this issue and write fast.

Many flash file systems need make a compromise between robustness against performance. Thanks to the log structure, Yaffs does not need to. Yaffs can provide high performance without giving up on robustness.

Those familiar with log structured file system design will know that some log structured file systems have problems with garbage collection³. Garbage collection is used to clean up the log and make more free space available. Some log structured file system designs did not pay enough attention to garbage collection and can stall for a considerable time while garbage collection happens. Yaffs was designed differently. The potential impact of garbage collection was considered throughout the Yaffs design process. As a result, Yaffs has a very simple garbage collection model that allows a lot of flexibility in garbage collection scheduling. This prevents the garbage collection from making the file system unresponsive.

Most file systems are developed within the context, and framework of, a single operating system. This typically makes the code difficult to explore, debug and port. Yaffs on the other hand was developed from the start inside a portable development/testing framework. Yaffs is then ported to various operating systems through the addition of glue code.

3 Refer to the "How Yaffs Works" document to understand the need for garbage collection and how Yaffs does garbage collection.



There are many benefits to this approach:

- A development framework runs in user space which provides a richer development environment than inside an OS kernel. It is much easier to attach debuggers and perform logging and create reproducible testing.
- The code is structured to be portable. That makes it relatively easy to port to a new OS or RTOS with a high level of confidence.
- There is a choice of porting interfaces allowing more flexibility to determine which porting method will be simpler. For example, the Linux wrapper accesses the Yaffs core directly while the Windows CE wrapper access the Yaffs Direct Interface⁴.
- Each different environment provides different test tools. For example, Linux provides a raft of file system test tools that do different kinds of testing to the power fail test

⁴ For more information on Yaffs Direct Interface, please read the various documents on the subject.



harness. Since the vast majority of the code is in the portable Yaffs core, that code gets the advantage of all test methods combined. The many millions of Android phone users that use Yaffs-based devices are helping test the core code that is also used on a wide variety of different operating systems and a wide range of different device types.

- Its Open Source status means any interested person can take a version from git and test it to make sure that it meets their needs.

4 Testing

It is all very well to design software to be robust, but it needs thorough testing to build confidence in the software and in the code authors and supplier.

File systems have incredibly complex state so it is not at all surprising that operating system designers find file systems some of the most challenging code to test.

There is no single test methodology that can test all cases, and exhaustive state testing is impossible. The best we can do is to have a suite of test methodologies that combine to provide thorough testing.

Approximately 60% of Yaffs development effort is directed towards testing and improving the test environment. We are constantly researching new methodologies.

The following are descriptions of the most important test strategies used in Yaffs.

4.1 Power Fail Stress Testing

By far the most important consideration in Yaffs testing is that it is robust to corruptions caused by power failure. We want Yaffs to provide the basis for reliable products in typical embedded system environments.

The first power fail testing was done by companies integrating Yaffs into their products. Some of these built test jigs which would automatically interrupt power while running a real hardware device. This approach is better than nothing, but has some limitations:

- It is relatively costly to set up. Flash has a limited lifetime meaning that test devices wear out with time and must be replaced.
- It is relatively slow since each cycle takes many seconds to boot, run the application for a while and then reboot.
- Many, and perhaps most, of the power interruptions will happen at times when the file system is inactive and thus be wasted cycles.

In 2008 we developed a simulated power-fail-test environment. This simulates power failures using simulated flash memory while running various consistency checks.

The benefits of a software-based test harness are numerous:



- The test harness controls the point at which the simulated power failures happen. That means every test cycle counts.
- No special hardware is required.
- Test loops run faster. A quad-core computer can simulate ten to twenty power failures per second. That means about a million power failure simulations per day.

The effectiveness of the new software based testing is dramatic.. A few times we have tested some user-suggested changes that had passed a week or more of real-world power fail testing only to have problems caught by the software simulation in a matter of minutes.

We regularly run the power fail simulation test over a weekend, testing millions of power fail cycles.

4.2 Testing the Yaffs Direct Interface API

Yaffs Direct Interface (YDI) is a POSIX-like wrapper around the Yaffs core. This provides a set of function calls, `yaffs_open()`, `yaffs_write()` and the like.

Each function can return numerous error codes and has to handle numerous different parameters.

Although there were already significant tests for the YDI, an extensive test harness was developed in 2010 to comprehensively test each of the error paths and generate all of the required errors.

This provides a high level of confidence that the test interfaces do what they should and that POSIX-like functionality provides relatively clean porting of existing code.

4.3 Linux Testing

The main purpose of the Linux testing is to test the Linux wrapper code. It does, however, still exercise the Yaffs core code differently to other testing and thus adds to the test fabric.

The Linux testing does not do power fail testing but instead tests other aspects such as clean shutdowns, cache integrity and write performance. This gives a useful way to assess the effects of modifications.

Linux testing is done on both real hardware and on simulated hardware using the Linux mtd nand simulator.

As with the Power Fail Stress Testing described above, the simulated testing is done using a scripted environment which is typically run for a few days at a time.

4.4 Fuzz Testing

Fuzz testing deliberately corrupts the flash image and ensures that Yaffs still mounts. This is done to ensure that Yaffs does not crash on corrupted data.



Clearly fuzz testing can corrupt the contents of individual files (that's its job), but the file system integrity should not be compromised.

4.5 Software checking

Although code checking is not actual testing per se, it does help to verify code changes.

The Yaffs code has been checked with Coverity⁵ – a market leader in code checking.

We are investigating the use of KLEE and other tools.

4.6 Open Source Usage

Open Source usage of Yaffs opens it up to testing by the many different projects. Many of these participate in some way with the Yaffs mailing list, helping to uncover issues and generally provide feedback.

Each different project uses Yaffs in a different way, meaning that a wider range of operation sequences are tested.

4.7 Universities

Over its lifetime, Yaffs has been used as a teaching framework for various university course in many different countries around the world.

For example, Oregon State and Utah State Universities have used Yaffs as a test bench for their post-graduate software testing courses. They essentially test the test tools by injecting errors into the Yaffs code and seeing if these are discovered by the test tools they are testing. Of course this has the side effect of running these tests over the original Yaffs code too.

Those interested in test methods are encouraged to read some of the university papers in the references of this document.

We expect to adopt and extend some of the Oregon State ideas and use these to extend our in-house test suite.

5 Conclusions

Software development is an iterative process of design and testing. Robust software solutions are only possible with extensive testing.

Embedded systems increasingly need to store data in file systems as a critical part of their operation. It is thus important to have a file system that is predictable and robust to power failure and similar interruptions.

Yaffs has been designed from scratch to be robust to power failure and that robustness has been verified with a multi-pronged test strategy that is continuously being improved.

⁵ <http://www.coverity.com>



6 References

Charles Manning, “How Yaffs Works”.

<http://users.actrix.co.nz/manningc/yaffs-docs/HowYaffsWorks.pdf>

A. Groce et al, “Swarm Testing” from Oregon State University, School of EE and Computer Science and Utah State University, School of Computing, 2011.

<http://www.ct.utah.edu/~regehr/papers/swarm12.pdf>

Wu CH, “Efficient Initialization and Crash Recovery for. Log-based File Systems over Flash Memory” from National Taiwan University, Dept of Computer Science, 2006.

http://www.cis.nctu.edu.tw/~lpchang/papers/SAC_wu_sac06.pdf

