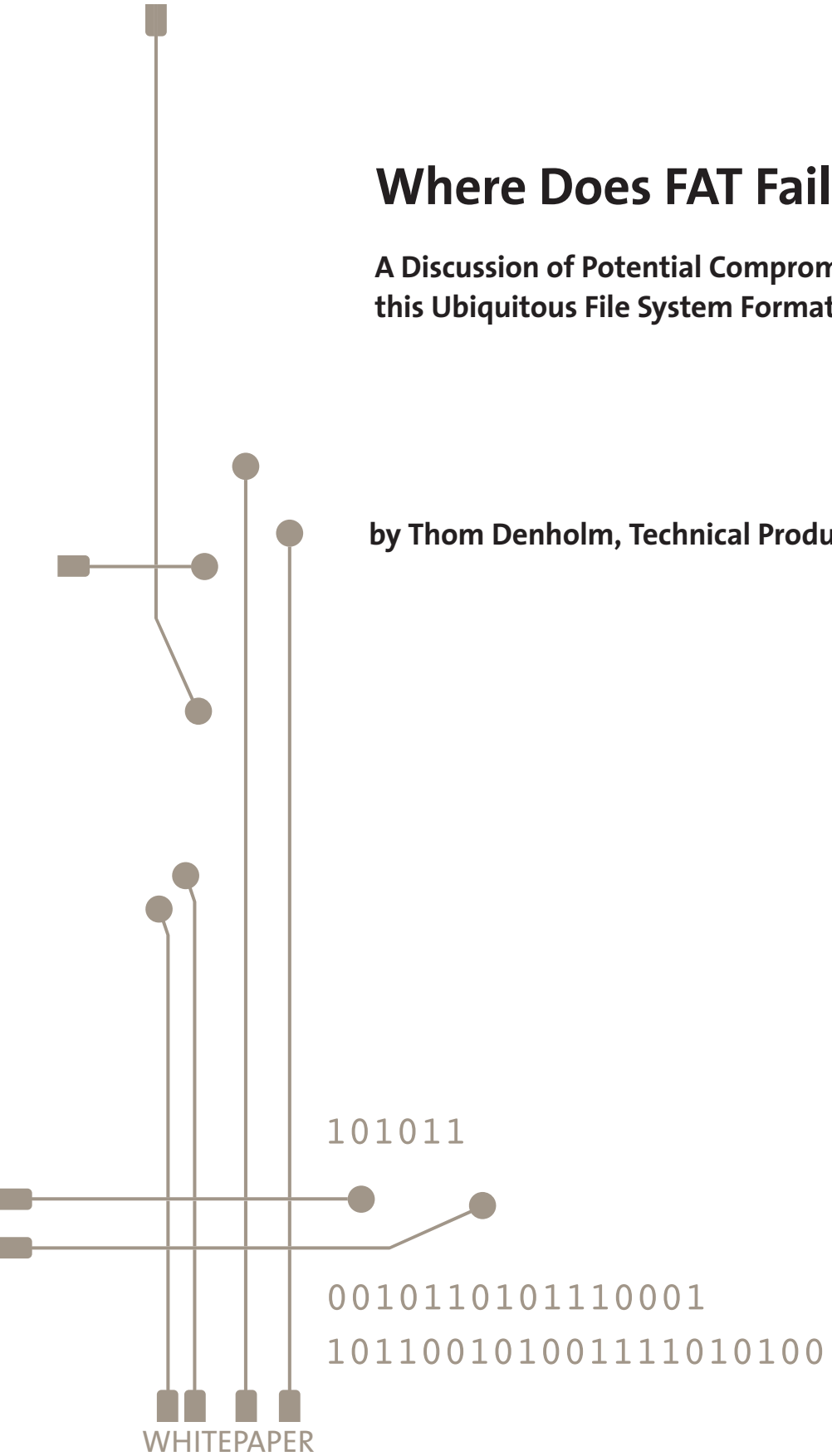
Datalight

Where Does FAT Fail?

A Discussion of Potential Compromises in this Ubiquitous File System Format

by Thom Denholm, Technical Product Manager



WHITEPAPER

Introduction

File systems have been used for organizing data in computers since data began to be stored in them. As an early entrant, the FAT file system has attained a certain ubiquity and almost all operating systems offer it in some form. Why? Because its simplicity makes it easy to understand and implement and its ubiquity makes it a common denominator presumed to provide interoperability among systems. In this document, we examine the underlying mechanisms used by the original DOS file system, some characteristics that leave it vulnerable to corruption, and some of the ways that vendors (including Microsoft) are trying to make it more reliable for today's users. Are they successful? Can reliability be achieved without sacrificing interoperability? Is the performance cost worth the benefit?

Contents

What is FAT?

The FAT (File Allocation Table) file system was originally designed for the BASIC interpreter, and was later incorporated into QDOS, which evolved into PC-DOS and MS-DOS. The target environment for this software was a desktop computer, and robust handling of power interruption was not a consideration as those machines were in their infancy (though it did create a large market for Uninterruptable Power Supplies).

Many embedded designs depend on battery power, often using a removable battery, and power loss occurs far more frequently than in a desktop situation. To properly handle unexpected power loss, operations must be planned for and contingencies made to clean up the mess afterwards.

In this article, we examine the internal structures of FAT and some retrofits attempted to add reliability. These include journaling (Samsung's RFS) and transactions (Microsoft's TFAT and TexFAT). All of these solutions are limited by a requirement to remain exchangeable across FAT environments.

Key Takeaways

FAT file systems, while simple to implement, offer no native protection against common failure scenarios. Presumption of interoperability and performance may be compromised by mechanisms added to provide failsafe operation.

FAT File System Basics

The FAT file system stores user data in a series of clusters on the media. Each file on the media resides in a directory or folder, either in the root directory or one of many subdirectories. Each of these directories also allocates one or more clusters to contain the directory entries, which contain much of the metadata associated with each file – items such as the file name, extension, creation date and time, and the file size. When the initial directory entry allocation is filled up, a new cluster is allocated, and it is often not contiguous with previous entries. This is why directories take progressively longer to read as more files are added.

The directory entry of each file or subdirectory contains a number which indicates the first cluster for that file or directory. A series of these pointers to clusters is stored towards the front of the media in what is known as a File Allocation Table. A zero entry in this table indicates the cluster is

available, while a non-zero entry either points to the next cluster for the file (a forward pointing chain of entries) or contains a flag indicating that this is the last cluster in a file.

By default, there are two copies of this File Allocation Table, which in the original implementation of FAT allowed a backup to handle media corruption and failed sectors – a common problem on floppy diskettes and early hard drives. Some recent implementations have replaced this mirroring with alternate uses, such as a transaction area. Microsoft's TFAT implementation stores allocations in the first FAT region, then commits the data to the media, then updates the second FAT region. Other non-reliable implementations merely zero or disable the mirrored FAT entries, giving faster performance but removing the possibility of file system error correction of media failures.

The number of FAT regions used is stored in the BPB (BIOS Parameter Block), a portion of the Boot sector used to track internal information. Not all implementations of FAT drivers will handle values other than 1 or 2 for the number of FAT regions. Media accessed by a simpler implementation would not update subsequent regions, and the original driver would have to deal with the table mismatch on first mount.

The Boot sector is usually unchanged by ordinary disk operations. In a FAT32 implementation, the sector following the boot sector is typically used for the File System Information block, which contains the number of free clusters on the volume. This can be updated frequently.

Journaling FAT

Initially designed for Samsung OneNAND flash media in Linux, Samsung's Robust File System (RFS) is a FAT16/FAT32 compatible which adds a transaction log for journaling (filename $rfs_log.lo$) in the root directory. This file system is designed for the Linux kernel, and does not run in other environments.

In addition to a journal, RFS does what it can to protect data overwrites by using an Extended Sector Remapper (XSR) to provide a logical to physical translation. This matches the write patterns of NAND flash, and the documentation suggests that a NAND Block Management Layer (BML) can be used to provide a FAT-based flash file system.

Samsung used this file system in a generation of Galaxy S smart phones. The next generation returned to using ext4, as the performance of RFS was found to be substandard.[1]

Transactional FAT

As mentioned above, the two copies of the FAT provide a backup in the case of media failure. Transactional FAT uses these two to reflect two on-media states. Depending on the size of media, TFAT12, TFAT16 and TFAT32 can be used. For larger media, the exFAT file system can be extended as TexFAT.

Each FAT represents one on-media state, initially identical. When an application modifies an ex-

Media Failure

Traditional rotating media has far more moving parts than NAND media, but each will fail over time. While failures on NAND media are usually individual blocks, rotating media failures can range from blocks to tracks to whole platters.

Data written on a failed block is usually lost, to the detriment of the application or file system. The File Allocation Table is so important to this file system that it was designed for two separate copies. Many of the FAT performance improvements remove this safety net, to the detriment of the overall design.

Modern failures are more block based, and thus less detectable to the file system software. For this reason, some newer file systems are adding CRC32 protection to their metadata.

isting metadata or directory entry, a free block is allocated and the first FAT is updated to reflect this new state. By default, only modifications to a directory and the FAT are backed up during a transaction using TFAT. On Windows Embedded Compact (and Windows CE), the registry key TransactData must be set to transact all user data modifications as well.

Perhaps the biggest penalty with this reliability approach is performance. As noted on Microsoft's website, "the additional copy operations inherent in TFAT handling cause a decrease in performance."[2]

TexFAT has been implemented in other RTOS environments, and works on many kinds of block-based media. Use of this file system in a commercial project requires a license from Microsoft, which aggressively protects the copyright of FAT and derivatives of this legacy product. Even the simple Linux VFAT file system (based on FAT, not exFAT) has not been truly tested in court – the closest was the case with embedded manufacturer Tom Tom, which was settled out-of-court.[3]

Failure Scenarios

Overwrites

While the methods above add some reliability to the file system structures, the user data is not protected by default. This is especially noticeable when modifying the data in a file, either by changing a block mid-file or by extending a file through append.

On an ordinary block device, these operations will overwrite the blocks in question. A power interruption while writing a block can leave that block in an indeterminate state – containing the original data, the new data, or a combination of both. While the metadata pointers may be correct, the resulting data blocks are effectively garbage.

In this failure situation, the CHKDSK utility may not detect anything wrong with the file system, allowing operating to proceed until some point when an application looks for the missing or corrupted data, possibly with unexpected results.

Subdirectories

For these sorts of operations, many parts of the file system metadata will be involved in the write. These include the directory entry blocks, the user data clusters, and associated directory entries in the case of subdirectory creation and deletion.

More writes lead to more chances for failure during a power interruption, and one new case – invalid files or directories. A power interruption here can cause the loss of a folder full of files, as the overwritten blocks are in an indeterminate state.

In this failure situation, the CHKDSK utility will often create a host of FILE####.CHK files to sort through – with the filenames and extensions lost.

Non Atomic Writes

Another potential problem has to do with the mechanism of updates in the FAT file system. Writes to the user data and the file system metadata should be performed at the same time. This is known as an *atomic operation*. None of the FAT implementations examined collect writes in this manner.

In many of the power interruption cases described above, the power interruption occurs while writing a block to the media. For a given atomic collection of writes, any power interruption in the middle of those writes results in a mismatch on the media. User data may be completely written, but none of the file system metadata has been updated. If the file contains all of the user data but the file system metadata indicates the length is shorter, the application will likely read only what it thinks is available. On subsequent writes, this additional data is lost, even though it was committed to the media.

FAT Damage

Perhaps the worst possible situation is where the FAT itself is damaged. Each FAT entry points to the next entry of the file, and these entries may be spread across multiple bocks. If power is interrupted after an incomplete set of writes, the FAT entries can become confused. The result can be *cross-linked files* (where two files point to the same FAT entry) or *lost clusters* (where no files point to a FAT entry).

Here the CHKDSK may have a file name, but the data within is incorrect. Lost clusters contain user data but are not associated with any file – a potential security situation when a new file is assigned those blocks on creation.

Conclusion

Considering the potential risks many developers draw a simple conclusion – the FAT file system is not designed for power interruption, whether on desktops or embedded devices. Critical data can be lost or require inspection before use. File system corruption, while unlikely to occur, can be fatal to the operation of the system. The non-atomic nature of FAT gives no guarantee that the user data matches the file system metadata, which can lead to failures whose root cause is difficult to troubleshoot.

Operations designed to improve the reliability of FAT media are somewhat successful in preventing the complete corruption of the file system data structures, but are rarely effective in protecting the user data. Microsoft's TexFAT solution can also protect the user data but the mechanisms used greatly reduce performance, which is a cost too high to bear for most embedded designs.

Requirements for simplicity and interoperability can be met while also achieving reliability and performance goals by using a more modern file system design, such as a transactional file system, like Datalight's Reliance, Reliance NItro or Reliance Edge.

References

1 http://www.myce.com/news/new-file-system-optimized-for-flash-memory-in-upcoming-linux-kernel-65429/
2 http://msdn.microsoft.com/en-us/library/aa915463.aspx
3 http://en.wikipedia.org/wiki/Microsoft_Corp._v._TomTom_Inc.

Find These Related Topics on www.Datalight.com

Whitepaper: Journaling vs Transactional File Systems

Whitepaper: Considerations For File System Selection for Flash Memory

Whitepaper: A Study of the Impact of File System Selection on Life Expectancy of Solid State Storage

Whitepaper: High-performance, Reliable Software for Critical Data Storage, Long-lived Devices

Documentation: Reference Manual for Reliance Edge, a tiny, power failsafe file system

About the Author

Thom Denholm, *Technical Product Manager*

Thom is an embedded software engineer with more than 20 years of experience, combining a strong focus on operating system and file system internals with a knowledge of modern flash devices. He holds a BS in Mathematics and Computer Science from Gonzaga University. His love for solving difficult technical problems has served him well in his fifteen years with Datalight.

In his spare time, he works as a professional baseball umpire and an internet librarian. Though he has lived in and around Seattle all his life, he has never had a cup of coffee.

About Datalight

Datalight products have delivered proven reliability in hundreds of millions of devices in demanding product categories like automotive, medical, retail, industrial automation and military/aerospace. When data integrity, device lifespan, and design flexibility matter, the world's leading device manufacturers invest in solutions from Datalight. Our product line includes:

- **Reliance Edge™**–a tiny, power failsafe file system for Internet of Things devices
- **Reliance Nitro™**–a high-performance, power failsafe file system
- **FlashFXe™**–software acceleration for managed flash
- **FlashFX® Tera**–comprehensive software management for raw flash.

For more information, visit
www.Datalight.com
or call 425.951.8086

Datalight, Inc.
22118 20th Avenue SE, Suite 135
Bothell, WA 98021 USA
1-800-221-6630
www.Datalight.com