How YAFFS Works

Charles Manning

2007-2010

Table of Contents

1 Purpose

The purpose of this document is to give a reasonable explanation of most of the core mechanisms that make yaffs work. As of the time of writing this, the yaffs2 code base includes approximately 19,000 lines of code making a very detailed discussion impractical.

This document should serve as a first port of call for understanding yaffs. Further detailed questions and discussions are welcomed on the yaffs discussion list.

This document is updated periodically. Yaffs mechanism design changes from time to time and this document will not always reflect the most up to date code.

2 What Is YAFFS?

YAFFS stands for *Yet Another Flash File System*, a term coined by Charles Manning in 2001 when suggesting that the already cluttered flash file system space could do with yet another offering – this time a flash file system designed from the ground up to work with NAND flash.

The first release of YAFFS was developed during the end of 2001 and the beginning of 2002. By mid-2002, YAFFS was being trialled in various products. In May 2002, YAFFS was announced and a few more projects started playing with it. In June, porting to other OSs started. In September 2002, YAFFS was announced on linuxdevices.com which started a far wider uptake of YAFFS.

YAFFS1, the original version of YAFFS, supported 512-byte page NAND devices with a SmartMedia-like memory layout.

YAFFS2 is a follow-up to YAFFS1 extending YAFFS1 to support larger and different devices with different constraints.

YAFFSis highly portable and has been used in many different products and applications as varied as sewing machines, point of sale, phones and aerospace. YAFFS has been used with multiple different operating systems. Native support is available for Linux, WindowsCE and eCOS while yaffs Direct Interface provides a layer that can be used in other applications such as RTOSs. YAFFS has been used with multiple different CPUs and compilers.

YAFFS has also been used as a NOR file system and even as a RAM file system.

3 Design and coding strategies

YAFFS is designed to work in multiple environments which drives the need for portability. Portability also improves the debugging and testing since it is easier to develop and debug code in an application environment than within an OS kernel. This generally allows YAFFS to progress faster with less programming resources. Primary strategies that improve portability include:

- No OS-specific features used in main code body.
- No compiler-specific features used in the main code body.
- Abstract types and functions used to allow Unicode or ASCII operation.

Simplicity is another key goal. Complexity is limited to those situations that provide significant improvements to performance or utility. Simplicity improves robustness and the ease of integration and development. Primary strategies that improve simplicity are:

- Single threaded model. YAFFS is locked on a per-partition basis at a high level. This is simpler than tracking lower-level locking. Yaffs Direct Interface uses a single lock for all partitions.
- Log structure makes for a simpler garbage collector and allocation methodology.
- Abstractions that build layered code which is easier to understand and debug.

4 Terminology: YAFFS vs YAFFS1 vs YAFFS2

Before we embark on our quest to understand what YAFFS is, a terminology explanation is in order.

There are two versions of YAFFS code: the original yaffs code-base and the current yaffs2 code-base. The yaffs2 code-base supports functionality provided by the yaffs code-base as well as being extended with new functionality to provide extra modes of operation, most importantly those required to work with larger and more modern NAND parts. The yaffs2 code-base supports the yaffs1 mode of operation through a backward compatibility mode.

Unless explicitly mentioned, this text refers to the yaffs2 code-base.

This text uses three terms YAFFS, YAFFS1 and YAFFS2 depending on various modes of operation.

•YAFFS1 is a simpler mode of operation that uses deletion markers to track state.

•YAFFS2 is a more complex mode of operating that was developed to support larger flash types that cannot used deletion markers.

•YAFFS means operations common to both.

YAFFS1 originally only worked with 512-byte page devices but has been extended to support larger flash pages such as Intel M18 flash with 1k pages.

YAFFS2 was originally designed for 1k or larger page sizes but can be used with smaller pages by using in-band tags.

Since the YAFFS1 mode of operation is simpler to understand than the YAFFS2 mode of operation, YAFFS1 will be described first and readers interested in YAFFS2 operation should first fully acquaint themselves with the YAFFS1 mode of operation. It is suggested that this document be read sequentially rather than browsing it.

5 Objects

In YAFFS-speak, an Object is anything that is stored in the file system. These are:

- Regular data files
- Directories
- Hard-links
- Symbolic links

- Special objects (pipes, devices etc).

All objects are identified by a unique integer ***objectId***.

In the standard POSIX model, as used by Linux, there are inodes and directory entries (dentries).

An inode may be a regular file, directory, or special file.

Directory entries provide the naming mechanism that is used to locate inodes.

Under POSIX, each inode may have zero, one or many dentries:

- One dentry per inode is commonly understood.
- Multiple dentries per inode allow the same inode to be accessed via multiple names. This is achieved through hard links.
- Zero dentries per inode is less often understood. This happens when an inode is unlinked while there is still a handle open to the inode. The unlinking deletes the dentry but the inode still exists. As soon as the handle is closed the inode is deleted too.

6 The YAFFS NAND Model

The memory in NAND flash is arranged in ***pages.*** A page is the unit of allocation and programming. In YAFFS, the unit of allocation is the ***chunk***. Typically a chunk will be the same as a NAND page, but there is flexibility to use chunks which map to multiple pages (eg. A system may have two NAND chips in parallel requiring 2x512 = 1024 byte chunks) . This distinction gives a lot of flexibility in how the system can be configured. In this text the term page and chunk are synonymous unless stated otherwise.

Many, typically 32 to 128 but as many as a few hundred, chunks form a ***block***. A block is the unit of erasure.

NAND flash may be shipped with bad blocks and further blocks may go bad during the operation of the device. Thus, YAFFS is aware of bad blocks and needs to be able to detect and mark bad blocks.

NAND flash also typically requires the use of some sort of error detection and correction code (ECC). YAFFS can either use existing ECC logic or provide its own.

7 How YAFFS1 Stores Files

YAFFS1 has a modified log structure, while YAFFS2 (see below) has a true log structure. A true log structured file system only ever writes sequentially. YAFFS1 uses deletion markers, which breaks the sequential write rule. YAFFS2 does not use deletion markers and is thus a true log structured file system.

Instead of writing data in locations specific to the files, the file system data is written in the form of a sequential log, The entries in the log are all one chunk in size and can hold one of two types of chunk:

- ***Data chunk***: A chunk holding regular data file contents.
- ***Object Header***: A descriptor for an object (directory,regular data file, hard link, soft link, special descriptor,...). This holds details such as the identifier for the parent directory, object name, etc.

Each chunk has ***tags*** associated with it. The tags comprise the following important fields (others being ignored for now):

- ***ObjectId***: Identifies which object the chunk belongs to.
- ***ChunkId***: Identifies where in the file this chunk belongs. A chunkId of zero signifies that this chunk contains an objectHeader. ChunkId==1 signifies the first chunk in the file (ie. At file offset 0), chunkId==2 is the next chunk and so on.
- ***Deletion Marker:*** (YAFFS1 only) Shows that this chunk is no longer in use.
- ***Byte Count***: Number of bytes of data if this is a data chunk.
- ***Serial Number***: (YAFFS1 only) Serial number used to differentiate chunks with the same objectId and chunkId.

Now let's bring that all together in an explanation. This is a simplified explanation and skips over issues such as Soft Deletion (mentioned later in the document). For explanation purposes, we shall consider usage of fictitious NAND type that has 4 chunks per block, starting with an empty (erased) flash.

First we'll create a file.

Block	Chunk	ObjectId	ChunkId	Deletion	Comment
1	0	500	0	Live	Object Header for this file (length 0)

Next we write a few chunks of data to the file.

Block	Chunk	ObjectId	ChunkId	Deletion	Comment
1	0	500	0	Live	Object Header for this file (length 0)
1	1	500	1	Live	First chunk of data
1	2	500	2	Live	Second chunk of data
1	3	500	3	Live	Third chunk of data

Next we close the file. This writers a new object header for the file. Notice how the previous object header is deleted.

Block	Chunk	ObjectId	ChunkId	Deletion	Comment
1	0	500	0	Deleted	Obsoleted object header (length 0)
1	1	500	1	Live	First chunk of data
1	2	500	2	Live	Second chunk of data
1	3	500	3	Live	Third chunk of data

2	0	500	0	Live	New object header (length n)

Let's now open the file for read/write, overwrite part of the first chunk in the file and close the file. The replaced data and object header chunks become deleted.

Block	Chunk	ObjectId	ChunkId	Deletion	Comment
1	0	500	0	Deleted	Obsolete object header.
1	1	500	1	Deleted	Obsolete first chunk of data
1	2	500	2	Live	Second chunk of data
1	3	500	3	Live	Third chunk of data
2	0	500	0	Deleted	Obsolete object header
2	1	500	1	Live	New first chunk of file
2	2	500	0	Live	New object header

Now let's resize the file to zero by opening the file with O_TRUNC and closing the file. This writes a new object header with length 0 and the contents are pruned making the data chunks deleted.

Block	Chunk	ObjectId	ChunkId	Deletion	Comment
1	0	500	0	Deleted	Obsolete object header.
1	1	500	1	Deleted	Obsolete first chunk of data
1	2	500	2	Deleted	Second chunk of data
1	3	500	3	Deleted	Third chunk of data
2	0	500	0	Deleted	Obsolete object header
2	1	500	1	Deleted	Deleted first chunk of file
2	2	500	0	Delted	Obsoleted object header
2	3	500	0	Live	New object header (length 0).

Note how all the pages in block 1 are now marked as deleted. This means that block 1 no longer contains useful information so we can now erase block 1 and reuse the space.

We will now rename the file. To do this we write a new object header for the file

Block	Chunk	ObjectId	ChunkId	Deleted	Comment
1	0				Erased
1	1				Erased
1	2				Erased
1	3				Erased
2	0	500	0	Deleted	Obsolete object header
2	1	500	1	Deleted	Deleted first chunk of file
2	2	500	2	Deleted	Obsoleted object header
2	3	500	0	Deleted	Obsolete object header
3	0	500	0	Live	New object header showing new name

Block 2 now contains only deleted chunks and can be erased and reused.

Note how the tags tell us:

•which chunks belong to which object,

•the position of the chunks within a file,

•which chunks are current

With that information we can recreate the state of the files regardless of the placement of the chunks in NAND. This means that should the system fail (crash, power fail,...) at any point during the sequences above, we are still able to recover the file system state up to that point.

Note that there are no file allocation tables or similar structures stored in the flash. This reduces the amount of writing and erasing (improving write performance) while also improving robustness. Corruption of file allocation tables and similar structures is a common failure mechanism for embedded file systems. The lack of such structures makes yaffs more robust. Or, to put it another way, you can't corrupt what you don't store.

8 Garbage collection

As we have shown so far, when a block is made up only of deleted chunks, that block can be erased and reused. However, consider what will happen if we have many blocks which each have a few current chunks in them. Clearly we cannot erase any of the current blocks or we will destroy data. What we need to do is copy the useful data chunks off a block, deleting the originals and allowing the block to be erased and reused. This process is referred to as ***garbage collection***.

The YAFFS garbage collection process works as follows:

1. Find a block worth collecting (according the the heuristics below). If none found, then exit.
2. Iterate through the chunks in the block. If the chunk is in use then create a new copy and delete the original. Fix up the RAM data structures to reflect the change.

Once all chunks in this block have been deleted, the block can be erased ready for reuse.

Note that although step 2 deletes the original chunk when it gets copied there is not much point in actually programming the deletion marker in flash since we're about to erase the block anyway. Thus yaffs does not program the deletion marker in these cases. If power is lost part way through the operation we can still tell the difference between the original and the copy by looking at the serial number (not that it really matters anyway since the two have the same contents so either can be used).

The heuristics to determine whether a block is worth collecting are as follows:

1. If there are many erased blocks available, then YAFFS does not need to work hard, but will instead only attempt to perform garbage collection of blocks with very few chunks in use. This is termed ***passive*** garbage collection.
2. If, however, there are very few erased blocks, YAFFS will work much harder to recover more space and will garbage collect blocks with more chunks in use. This is termed ***aggressive*** garbage collection.

OK, the passive/aggressive names are a bit silly, but they were contrived around midnight... give me a break!

If garbage collection is aggressive, the whole block is collected in a single garbage collection cycle. If the collection is passive then the number of copies is reduced thus spreading the effort over many garbage collection cycles. This is done to reduce garbage collection load and improve responsiveness.

The rationale behind the above heuristics is to delay garbage collection when possible to reduce the amount of collection that needs to be performed, thus increasing average system performance. Yet there is a conflicting goal of trying to spread the garbage collection so that it does not all happen at the same causing fluctuations in file system throughput. These conflicting goals make garbage tuning quite challenging.

All flash file systems need some sort of garbage collector to reclaim space. The garbage collector is often a determining factor in the performance of the flash file system. Many file systems have to do a significant amount of effort in a single collection sweep. YAFFS only deals with one block at a time which constrains the amount of effort in a garbage collection cycle, thus reducing the “stall time”.

The YAFFS garbage collection algorithm is under review with the goal of reducing “stall time” and increasing throughput.

9 YAFFS1 Serial numbers

In YAFFS1, each chunk is marked with a 2-bit serial number that is incremented every time a chunk with the same tags value is replaced (eg. When a chunk is overwritten or when copied by garbage collection).

Now let's suppose first that we did not have a serial number.

Let's consider a rename where an object header chunk must be replaced with a new one with the new name. YAFFS could do the following:

•Delete the old chunk.

•Write the new chunk.

If the system fails after the delete, the system could end up with no chunks matching the relevant tags values. This could cause files to be lost (end up in lost+found). Thus, we have to write the new chunk before deleting the old chunk.

But now consider what might happen if we did this:

•Write new chunk.

•Delete old chunk.

If the system failed after the write we now have two chunks that match the tags. How can we tell which one is current?

The current chunk is determined by inspecting the serial number. Since the serial number is only incremented by one before the old chunk is deleted, a 2-bit serial number is sufficient. Valid pairs are:

Old	New
00	01
01	10
10	11
11	00

Thus, by inspecting the serial number attached to the old and new tags the current chunk can be determined and the old chunk can be deleted.

10 YAFFS2 NAND model

YAFFS2 is an extension of YAFFS designed to fulfill a new set of objectives to work with newer NAND types including:

- Zero overwrites. YAFFS1 only ever overwrites a single byte in the spare area of the chunks to set the deletion marker. More modern NAND devices are less tolerant of overwrites and YAFFS2 performs no overwrites at all.
- Sequential chunk writing within a block. More modern NAND reliability specifications tend to assume sequential writing. Since YAFFS2 does not write deletion markers, the writing of chunks within a block is strictly sequential.

Since YAFFS2 performs less writes, there is potential for YAFFS2 to have improved write performance.

YAFFS2 cannot use a deletion marker because that would violate the zero overwrites mandate and alternate mechanisms

are provided to determine which chunks are current and which are deleted. These mechanisms are:

- ***Sequence Number***: As each block is allocated, the file system's sequence number is incremented and each chunk in the block is marked with that sequence number. The sequence number thus provides a way of organising the log in chronological order.
- ***Shrink Header Marker***: The shrink header marker is used to mark object headers that are written to shrink the size of a data file. This is explained more in the accompanying text.

Note: The YAFFS2 sequence number is not the same as the YAFFS1 serial number!

Note: We still refer to chunk deletion in yaffs2. The chunks are marked as deleted in the RAM data structures used by garbage collection etc, but there is no deletion marker written to flash.

The sequence number allows the chronological order to be determined which allows YAFFS to determine the sequence of events and restore the file system state. This is achieved by scanning ***backwards*** chronologically (ie. From the highest sequence number backwards to the lowest sequence number). Thus:

- Since we're scanning backwards, the most recently written – and thus current – chunk matching an objectId:chunkId pair will be encountered first and all subsequent matching chunks must be obsolete and treated as deleted.
- The file length in object headers are used to trim back resized files. If an object header is encountered, then subsequent data chunks that lie beyond that file length are clearly obsolete and treated as deleted. Note that both current and obsolete object header sizes must be used to obtain an accurate reconstruction.

Explaining the purpose of the shrink header marker is a bit convoluted. The purpose of a shrink header marker is to show that this object header indicates that a file has shrunken in size, and to prevent those object headers from being deleted by garbage collection. To understand this, it is easiest to consider what would happen if the shrink header markers did not exist.

For simplification we'll assume no garbage collection happens during this part of the exercise. It is safe to make that assumption since it does not alter the reasoning.

Consider a file that has been through the following:

```
h = open(“foo”,O_CREAT| O_RDWR, S_IREAD | S_IWRITE); /* create file /
write(h,data,51024*1024); /* write 5 MB of data /
truncate(h,1024*1024);  /* truncate to 1MB */
lseek(h,2*1024*1024, SEEK_SET); /* set the file access position to 2MB */
write(h,data,1024*1024); /* write 1MB of data */
close(h);
```

The file will now be 3MB in length, but there will be a “hole” between 1MB and 2MB. According to POSIX (and to aid security etc) that “hole” should always read back as zeros.

YAFFS2 would represent that in NAND with the following sequence of chunks:

1. Object header for file creation (file length = 0)
2. 5MB worth of data chunks (0 to 5MB)

3. Object header for truncation (file length = 1MB)
4. 1MB worth of data chunks (2MB to 3MB)
5. Object header for close (file length = 3MB)

At this stage, only the the following data chunks are current:

- The first 1MB of data created in step 2 above.
- The 1MB of data created in step 4 above.
- The object header created in step 5 above.

But YAFFS must remember the truncation that happened in step 3, otherwise YAFFS will “forget” that there is a hole, which would cause the old data to be visible again. Thus, if yaffs creates a hole it first writes a shrink header to indicate the start of the hole and a regular object header to mark the end of the hole.

The shrink header modifies the behavior of the garbage collector to ensure that these shrink headers are not erased until it is safe to do so. This unfortunately has a negative impact on garbage collection in systems which make extensive use of files with holes.

Shrink headers are also used to indicate that a file has been deleted and that the record of the file deletion is not lost.

10.1. YAFFS2 Extended tags

YAFFS2 extends the tags in object headers with extra fields to improve mount scanning performance. The operation of extended tags is described later in this document.

11 Bad block handling NAND error handling

NAND flash is designed to be very low cost and very high density. In order to get high yields, and thus keep costs down, NAND flash is typically shipped with a few bad blocks and some further blocks are expected to go bad during the use of the device. Thus, any flash file system without an effective bad block handling policy is unsuitable for NAND flash.

YAFFS1 uses the Smart-Media style of bad block marking. This checks the sixth byte (byte 5) of the spare area. In a good block this should be 0xFF. A factory-marked bad block should be 0x00. If YAFFS1 determines that a block has gone bad it is marked bad with 0x59 ('Y'). Using a distinctive marker allows YAFFS-marked bad blocks to be differentiated from factory-marked bad blocks.

YAFFS2 mode is designed to support a wider range of devices and spare-area layouts. Thus, YAFFS2 does not decide which bytes to mark but instead calls driver functions to determine whether the block is bad, or to mark it bad.

So, when does YAFFS mark a block bad? YAFFS will mark a block bad if a read or write operation fails or if three ECC errors are detected. A block that is marked bad is retired from use, thus improving the robustness of the file system. This policy is suitable for SLC flash. MLC policies are under development.

Further, NAND flash cells can sometimes be disturbed (corrupted) by NAND activity and charge loss. These errors are corrected by error correction codes (ECC) which may be implemented in hardware, software drivers, or within YAFFS

itself. Again, any flash file system lacking effective ECC handling is unsuitable for NAND flash.

YAFFS1 mode can use either the built in ECC or use ECC provided by the driver or hardware.

Since YAFFS2 mode is designed for a wider range of devices, it does not provide ECC internally but requires that the driver provide the ECC.

The ECC code supplied with YAFFS (yaffs_ecc.c) is the fastest C code implementation of a Smart Media compatible ECC algorithm that we are aware of. This code will correct any single bit error within a 256-byte data block and detect 2 errors per 256-byte data block. This is sufficient error correction to provide very high reliability on SLC-type NAND flash.

For a more complete description of failure handling, please refer to the YAFFS Error mitigation document.

12 RAM structures

While it is theoretically possible to use a log structured file system with very few RAM structures, this would provide reduce performance. Thus, significant RAM data structures are required to store enough information to provide sufficiently high performance.

Before we jump in, it is a good idea to first figure out what purpose the RAM structures serve. The main structures are defined in yaffs_guts.h and their main purposes, are :

- Device/partition: This is named ***yaffs_Device***. This is required to hold information relating to a yaffs "partition" or "mount point". Using these, rather than globals, allows YAFFS to simultaneously support multiple partitions and partitions of different types. In fact, almost other data structures mentioned here are part of, or accessed via, the structure.
- NAND Block Information: This is named ***yaffs_BlockInfo*** and holds the current state of the NAND blocks. Each yaffs_Device has an array of these.
- NAND Chunk Information: This is a bitfield attached to the yaffs_Device that holds the current in-use state of each chunk in the system. There is one bit per chunk in the partition.
- Object: This is named ***yaffs_Object*** and holds the state of an object (see section 3).There is one of these for each object in the file system, where an object is one of a regular file, directory, hard link, symbolic link or special link. There are different variants of yaffs_Objects to reflect the different data required for these different object types. Each object is uniquely identified by an objectId.
- File structure: For each file object, YAFFS holds a tree which provides the mechanism to find the data chunks in a file. The tree consists of nodes called ***yaffs_Tnodes*** (tree nodes).
- Directory structure: The directory structure allows objects to be found by name. The directory structure is built from a doubly-linked list that is rooted in the directory and binds together the sibling objects within a directory.
- Object number hash table: The object number hash table provides a mechanism to find an object from its objectId. The objectId is hashed to select a hash bucket. Each hash bucket has a doubly-linked list of objects that belong in this hash bucket, as well as a count of the objects in the bucket.
- Cache: YAFFS provides a read/write cache that significantly improves performance for short operations. The size of the cache can be set at run-time.

The usage of these structures is described in more detail below.

YAFFS makes significant use of doubly-linked lists. These data structures are very useful because they provide rapid insertion, removal and traversal. They do however come at the expense of requiring two pointers rather than just one.

12.1. yaffs_Object

Each object (file, directory,...) in the system is represented by a yaffs_Object. The yaffs_Object's main function is to store most of the object metadata and type-specific information. This means that almost all the information about an object can be accessed immediately without reading flash.

The metadata includes, amongst others:

•objectId: The number used to identify the object.

•parent: A pointer to the parent directory. Not all objects have a parent. The root director and special directories for unlinked and deleted files do not have parents so this is NULL.

•short name: If the name is short enough to fit in a small fixed-sized array (default 16 charaacters) then it is stored here otherwise it must be fetched from flash each time it is requrested.

•type: Type of object.

•permissions, time, overship and other attributes

Depending on the object type, yaffs_Object also stores:

•a tnode tree, file extents etc for data files

•a directory chain for directories

•an equivalent object for hard links

•a string for symbolic links

12.2. ObjectId look-up

The purpose of the objectId look-up mechanism is to rapidly access an object by its device and object Id. This is required during garbage collection, scanning and similar operations.

Each yaffs_Device uses a hash table for this purpose. The hash table has 256 “buckets” (tunable), each holding a doubly-linked list of the yaffs_Objects in that bucket.

The hash table's hash function is just masking the least significant bits of the object Id. This is a very cheap function that provides reasonable “spread”.

YAFFS makes an attempt to assign objectIds in a way that keeps the number of objects in each hash bucket quite low, thus preventing any one hash entry getting too long.

12.3. Directory structure

The purpose of the directory structure is to rapidly access an object by name. This is required to perform file operations such as opening, renaming etc.

The YAFFS directory structure is formed by a tree of yaffs_Objects of type YAFFS_OBJECT_TYPE_DIRECTORY. These objects have a doubly linked node of children.

Each yaffs_Object also has a doubly linked list node called sibling which chain together the siblings in a directory.

Each YAFFS partition has a root directory.

Each YAFFS partition also has some special purpose “fake” directories which are not stored in NAND but are instead created on mounting:

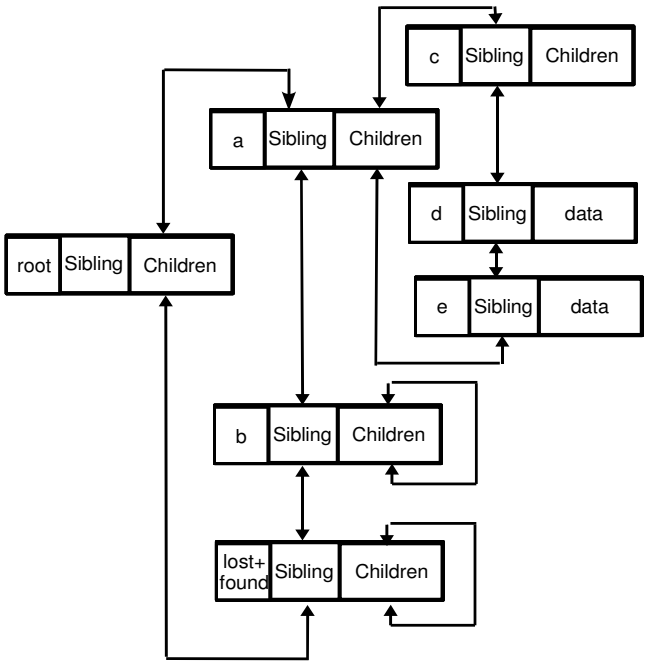
- lost+found: This directory is used as a place to store any lost file parts that cannot be placed in the normal directory tree.
- Unlinked and deleted: These are pseudo-directories that are not in the directory tree. Placing objects in these directories gives them a status of being unlinked or deleted.

Each object has a name that is used to look up the object within the directory, Thus, each name within a directory must be unique. This rule does not apply to objects in the unlinked and deleted pseudo-directories since we never use names to look up unlinked/deleted files and there may be many files of the same name that have been deleted.

Name resolution is sped up by two mechanisms:

•Short names are stored directly in the yaffs_Object soi that they don't have to be loaded from flash.

•Each object has a “name sum”. This is a simple hashing of the name which speeds up the matching process because it is cheaper to compare te hash than compare the whole name string.



The above directory structure represents the following directory tree

/	Root directory
/a	Directory containing c,d,e
/b	Empty directory
/lost+found	Empty directory
/a/c	Directory
/a/d	Data file
/a/e	Data file

If a YAFFS partition has been somehow corrupted then it is possible for an object to be created during scanning but with no object header (which contains the object name) being found. The object would then not have a name. In these cases, YAFFS creates a pseudonym from the object number of the form objnnnn so that the object can be placed in the directory structure.

12.4. Hard links

In POSIX file systems, the objects (inodes in Linux terminology, vnode in BSD-speak) and the object names (dentry in Linux terminology) are independent in that an object can have zero, one or more names.

Having files with zero links is caused by constructs such as:

```
h = open(“foo”,O_CREAT | O_RDWR, S_IREAD | S_IWRITE); /* create file “foo” */
unlink(“foo”); /* unlink it so it can't be found in the directory */
/* we still have a handle to the file so can still access it via the handle */
read(h,buffer,100);
write(h,buffer,100);
close(h); /* as soon as the handle is closed the file is deleted */
```

YAFFS does not store separate object and name records because that would waste space and incur extra reading/writing. Instead, YAFFS is optimised for the most typical case of one name per object. That is, YAFFS stores one name per yaffs_Object and uses “cheats” to achieve other behavior:

- Objects that have zero names (ie unlinked objects) are handled by renaming the object into a fake directory that is used to designate unlinked objects.
- Multiple names object are handled by designating one of the objects the main object and having multiple hard link pseudo-objects that refer to it.
- The inode number (vnid in BSD-speak) is a per-file system unique number that identifies the object. The YAFFS object id of the main object is presented to the user. That is, if the inode number is queried from a hardlink object then the object id of the equivalent object is returned.

The linkage between hard link pseudo-objects and their equivalent object is through two fields:

- Each object has a hardLinks doubly linked list node that links the list of hard links to that object.
- Each hard link has a pointer to the “equivalent object”.

The usage of this structure is straight forward except for one interesting case when hard links are being deleted. Consider the following sequence of events:

1	# touch a	Create file a

2	# ln a b	Create a hard link b with equivalent object a.
3	# rm a	Remove a. The object must still exist under the name b

In step 3, YAFFS cannot just delete the equivalent object since that would leave a “hanging” hard link with no equivalent object.

We also cannot just change the type of one of the hardlink objects because that would cause the object's object id to change. This would be unacceptable since it would break any mechanism that use the object id (of which there are many within and outside of YAFFS).

Thus, if an object is deleted and it has hard links to it then we do the following:

- Select one of the object's hard links.
- Rename the object to the name in the hard link.
- Delete the hard link pseudo-object.

12.5. Symbolic link and special object

These object types just store values that are needed to provide a full POSIX file system.

A symbolic link object saves a symbolic link string that is used to provide the POSIX symbolic string mechanism.

A special object is used to store device numbers, pipes and other similar special objects.

12.6. File object

The yaffs_Objects have a type and a variant portion which is a union of different data required to hold the state of different object types.

The file objects have type YAFFS_OBJECT_TYPE_FILE, and an associated file variant. This stores the following main values:

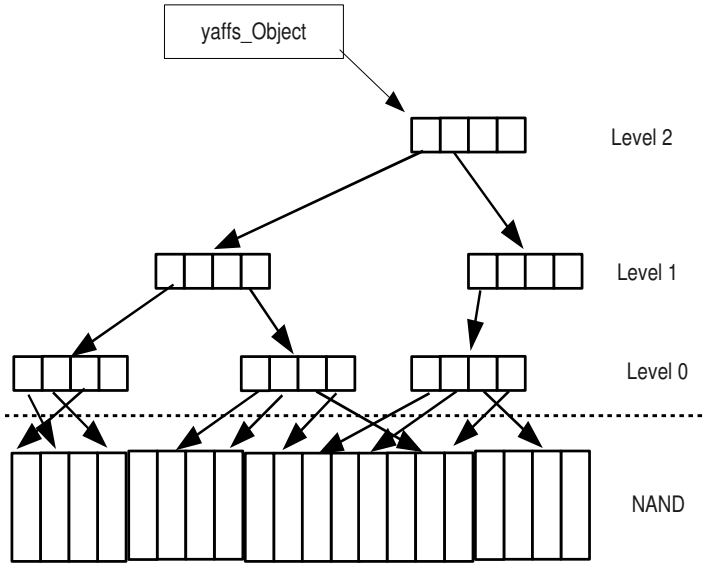
- fileSize: file size
- topLevel: depth of the Tnode tree
- top: pointer to top of Tnode tree

The meaning of the file size is quite obvious. This is the current length of the file. Note that a file might have holes in it caused by seeking forward past the extents of the file.

The top and topLevel work together to control the Tnode trees.

12.6.1 Tnode tree

Each file has a Tnode tree to provide the mapping from file position to actual NAND chunk address. top is a pointer to the top of the Tnode tree



The Tnode tree is made up of Tnodes (tree nodes) arranged in levels, each of which holds either:

- At levels greater than 0, a Tnode has 8 pointers to other Tnodes in the next level down.
- At level 0, a Tnode has 16 NAND chunk Ids which identify the chunk's location in RAM.

These numbers are powers of 2 which makes for simple look-ups by just applying bitmasks. The code that traverses this tree is in yaffs_FindLevel0Tnode().

As the file size grows, more Tnodes are added and as the levels fill, more levels are added. If the file is truncated or resized downwards, then the tree is pruned. If a chunk in a file is replaced (eg. data is overwritten or copied by garbage collection) then the Tnode tree must be updated to reflect the change.

Tnodes typically make up the bulk of the YAFFS' RAM usage.

13 How various mechanisms work

Having looked at the YAFFS data structures, we now consider how some of the mechanisms in YAFFS operates. This is most easily done by walking through some operations that YAFFS performs.

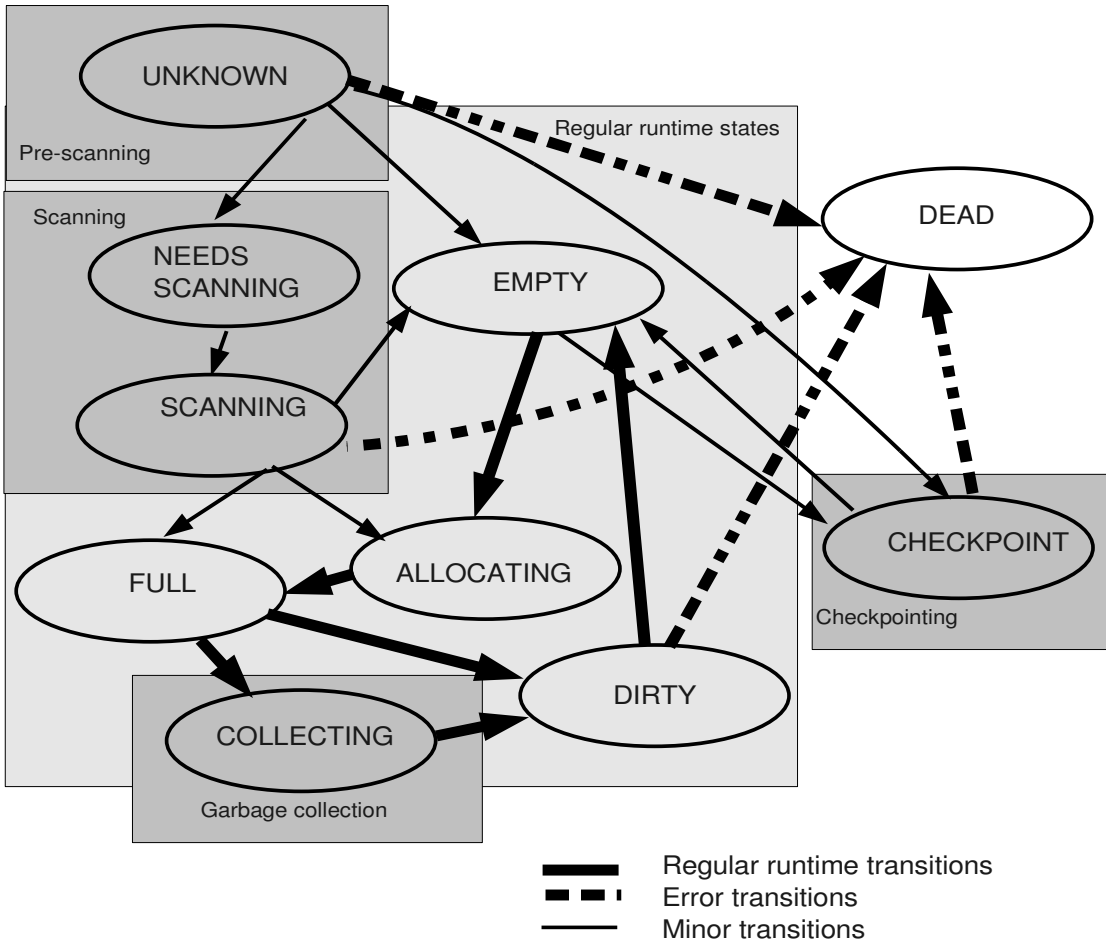
13.1. Block and Chunk Management

13.1.1 Block States

YAFFS tracks the state of each block and chunk in the partition. This information is first built up during scanning (or recovered from a checkpoint). A block may be in one of the following states (see definition of yaffs_BlockState):

UNKNOWN	The block state is not yet known.

NEEDS_SCANNING	During pre-scanning it has been determined that this block has something on it and needs scanning.
SCANNING	This block is currently being scanned.
EMPTY	This block has nothing in it (is erased) and may be used for allocation. The block can get to this state by being erased.
ALLOCATING	This is the block currently being used for allocation by the chunk allocator.
FULL	All the chunks in this block have been allocated and at least one chunk contains useful information that has not yet been deleted.
DIRTY	All the chunks in this block have been allocated and all have been deleted. The block might have been in the FULL or COLLECTING state before this. This block can now be erased and returned to EMPTY state.
CHECKPOINT	This block has checkpoint data in it. When the checkpoint is invalidated this block will be erased and returned to the EMPTY state.
COLLECTING	This block is being garbage collected. As soon as all live data has been inspected this block becomes DIRTY.
DEAD	This block has been retired or was marked bad. This is a terminal state.



The regular runtime states are; EMPTY, ALLOCATING, FULL, COLLECTING and DIRTY.

The checkpoint state is used for checkpoint blocks.

The DEAD state is the terminal state for blocks that were retired because they were either marked bad or an error was detected.

13.1.2 Block and Chunk Allocation

An available chunk must be allocated before it can be written. The chunk allocation mechanism is provided by yaffs_AllocateChunk() and is very simple. Each partition has a current block that it is allocating from. This is termed the ***allocation block***. Chunks are allocated sequentially from the allocation block. As the chunks are allocated, block and chunk management information is updated, including the block's pagesInUse count and the chunk use bitmap.

When the block is fully allocated, another empty block is selected to become the allocation block. The blocks are selected by searching upwards from the previous allocation block.

Previous to yaffs_guts.c V1.99, yaffs scanning would identify the allocation block being used when the file system was last shut down and would continue allocation from that point. This had the benefit of continuing to use the same block and slightly reduce garbage creation . However, doing this could potentially cause problems due to attempting to write a block that had been in the middle of a write when power was lost. Thus, from V1.99 onwards, yaffs now always starts allocation on a new block after a mount. This increases the amount of garbage (typically by an insignificant amount) but improves robustness.

13.1.3 A word about wear leveling

Since flash memory blocks can only endure a limited number of writes and erases (termed the flash endurance), a flash file system needs to ensure that the same few blocks are not overwritten continuously causing those blocks to wear out while the other blocks are hardly used at all.. Instead it is important that the file system spreads out the wear.

This is extremely important in a file system such as FAT which uses a limited set of logical blocks to store the file allocation table entries which get modified frequently. If FAT is used with a static logical to physical block mapping then the file allocation table blocks will wear out far before other blocks causing premature device failure.Thus, for FAT, it is very important that the logical to physical mapping gets stirred up a bit to cause the file allocation tables to be written to different physical blocks, thus levelling out the wear.

Wear levelling is far less of a concern in a log structured file system such as YAFFS since we always write to the end of the log and thus don't keep on overwriting the same blocks.

There are basically two ways to achieve wear leveling:

•use a specifically written set of functions that actively perform wear leveling.

•allow a degree of wear leveling to happen as a side effect of other activity.

YAFFS takes the second approach.

Firstly, being log structured, yaffs inherently spreads out the wear by performing all writes in sequence. That means that we don't single out a few blocks for a specific purpose and write them more than others.

Secondly, blocks are allocated serially from the erased blocks in the partition. Thus the process of erasing (after deletion of garbage collection) and then allocating blocks does tend to spread the block use and provides a level of wear levelling.

Thus, even though there is no code to specifically perform wear leveling it does happen as a side effect of other activity.

This strategy performs well and has survived many accelerated lifetime tests in both simulations and on real NAND-based products.

13.1.4 yaffs_Tnode and yaffs_Object Management

yaffs_Tnodes and yaffs_Objects are dynamically created and released as they are needed.

The allocation and management could be done by just using the standard memory allocator (malloc, kmalloc etc), but this could cause performance problems, particularly in embedded systems which tend to have rather simplistic memory allocators.

The allocation and deletion of many small structures can cause memory fragmentation causing out-of-memory and slowdown problems.

To prevent these problems, YAFFS manages these objects internally. YAFFS creates many structures (100 or so) at a time

and then allocates them singly from a free list. When structures are released they are placed on the free list for reuse.

13.2. Internal Cache

This describes the standard YAFFS internal cache implementation. Alternative caches have been devised for other purposes and at some future point YAFFS might be extended to support pluggable cache policies.

The main purpose of the internal cache is to reduce NAND access for badly behaved applications which perform many small reads and writes. Believe it or not, but there are applications out there that read or write files a few bytes at a time:

```
while(...)
      write(h,buf,1);
```

This sort of behaviour would be terrible for performance and flash memory life if each read or write operation resulted in a write to flash. YAFFS provides an internal cache to mitigate against this sort of behaviour.

The YAFFS internal cache is called a “Short Op Cache” (short operation cache) and only comes in to play for operations which are not chunk aligned. The caching mechanism is very simple and was primarily written for less sophisticated OSs that lack their own file caching layer. The cache is still beneficial in OSs like Linux, which provide a caching layer, because it caches both the read and write path. Thus, while short reads will be serviced by the Linux page cache, the shortOpCache does address non-chunk aligned writes.

Page aligned reads/writes bypass the cache unless that chunk is already in the cache.

The cache is stored in an array of cache entries. The number of cache entries is set in the yaffs_Device configuration by dev->nShortOpCaches and the cache is configured during initialisation. A cache size of zero disables the cache. The cache management algorithms are simplistic and probably don't scale well to very large numbers. This means that it is probably best to stick with around 5 to 20 cache entries.

Each cache entry holds:

- object id, chunk id, nbytes (the tag info of the chunk being cached)
- the actual data bytes
- cache slot status (last used counter, dirty,...)

Finding an unused cache entry is a simple operation. We just iterate through the cache entries searching for one that is not busy. If all cache entries are busy then a cache push-out is performed and then we retry the find.

Whenever a cache entry is accessed (read or written) the last-use counter is is set to an incrementing number. This counter value is used to drive the least-recently-used (LRU) algorithm.

Cache entries which have been modified and not yet written back to flash are marked with a dirty flag. This is tells us whether we need to write out the data during a flush or push-out.

A push-out happens whenever the cache is full and we need to push an entry out of the cache to make space to load a new entry. To do this we use the LRU algorithm to select the LRU cache entry. We then write out all dirty entries for that file, in ascending order, and mark these entries as free.

Whenever a file is flushed (file flush or close), the file's dirty entries are written out. At certain times (eg. before an unmount or at sync()) the entire cache is flushed.

This caching policy is far from optimal but gives a useful performance improvement with very simple algorithms (good bang-for-bucks).

Apart from its normal caching function, the shortOpCache also does much of the heavy lifting for inband tags support since all the inband tags accesses are not chunk aligned, they must all go through the shortOpCache.

13.3. Scanning

Scanning is the process whereby file system state is rebuilt from scratch. This happens on mounting a YAFFS partition if no checkpoint data is available or if the mounting was told to ignore the checkpoint data.

Scanning involves reading the tags from all the active chunks in the system so can take quite a while, but probably less time than you'd expect.

The scanning process is different for YAFFS1 vs YAFFS2 because of the differences mentioned before.

13.3.1 YAFFS1 Scanning

YAFFS1 scanning is rather straight forward because YAFFS1 uses deletion markers.

All we really need to do is read the tags off all the chunks in the partition and determine if the chunk is active. Order is not important and blocks could be read in any order. If the deletion marker has not been set then it is active and we need to add the chunk to the file system state as follows:

If the chunk is a data chunk (chunkId > 0) then the chunk must be added to the relevant file's tnode tree. If the object does not exist, then we create it in lost+found, although it has no name and other file details yet.

If the chunk is an object header (chunkId == 0) then the object must be created (or it might already exist in lost + found and we need to fix up its details and slot it into the correct part of the directory tree.

If at any time during scanning we find that we are reading a second chunk with the same objectId:chunkId as a previously encountered chunk then the 2-bit serial number is used to resolve the dispute.

Once the scanning is complete we still need to fix up a few things:

Hard links must be connected up properly (it is more efficient to defer this until after the scanning has completed).

Also, objects which were marked as unlinked must be deleted since there are no longer handles open to these objects.

13.3.2 YAFFS2 Scanning

YAFFS2 has no deletion markers which makes the scanning significantly more complex.

The first thing that needs to be done is to pre-scan the blocks to determine the sequence number for the blocks. The list of blocks and sequence numbers is then sorted to make a chronologically ordered list.

We now scan ***backwards*** through these blocks (ie in reverse chronological order). Thus, the first occurrence of any objectId:chunkId is the most current one and it is easy to determine which is chunks are out of date.

> We read the tags for each chunk in each block.
>
> If it is a data chunk (ie. chunkId > 0) then:
>
> - If a chunk with the same objectId:chunkId has already been found then this is not the current chunk. Delete this one.
> - Else if the object is marked as deleted, or corrupted, then delete it.
> - Else if the object does not yet exist then this chunk was written since the last object header write for this object so this chunk must be the last written chunk in the file and must have happened just before an

unclean shutdown with the file still open.We can create the file and use the chunkId and nbytes to set the file extents.

- Else if the object does exist and the chunk lies beyond the object's file scan size (extents) then it is a truncated chunk and can be deleted.
- Else it is placed into the file's tnode tree.

If it is an object header (ie. chunkId == 0) then:

- If the object is in the deleted directory then mark the object as deleted and set the object's shrink size to zero.
- Else if an object header has not yet been found for this object, then this is the most current object header and holds the most current name etc. If no data chunks have yet been found for this chunk then the size in the object header is also the file extents. The file size is used to set the file scan size since at the time it was written the file did not extend past that point and we can safely ignore further data chunks for that file that extend past that point.
- Else just use the file size as the scan size if it is smaller than the current file scan size.
- Else this an obsolete object header. It is deleted.

A bit more explanation is probably needed to describe the file scan size. The whole purpose of this is to determine data chunks that should be ignored due to the file being truncated.

13.4. Checkpoint

Mount scanning takes quite a lot of time and slows mounting, though perhaps by less than you'd think. Checkpointing is a mechanism to speed the mounting by taking a snapshot of the YAFFS runtime state at unmount or sync() and then reconstituting the runtime state on re-mounting. The speed up is dramatic.

The actual checkpoint mechanism is quite simple. A “stream” of data is written to a set of blocks which are marked as holding checkpoint data and the important runtime state is written to the stream.

Not all state needs to be saved, only the state needed to reconstitute the structure of the runtime data. For example, file metadata need not be saved since it is very easy to load that through lazy-loading.

The checkpoint stores data in the following order:

•start marker, including checkpoint format id.

•yaffs_Device information

•block information

•chunk flags

•objects, including file structure

•end marker

•checksum

The checkpoint validity is ensured by the following mechanisms:

•the data is stored using whatever ECC mechanisms are in place.

•the start marker includes a checkpoint version so that an obsolete checkpoint will not be read if the checkpoint code changes.

•data is written to the stream as structured records, each with a type and size.

•the end marker must be read when expected.

•two checksums are maintained across the whole checkpoint data set.

If any checks fail, the checkpoint is ignored and the state is re-scanned.

The astute reader might notice that block state changes as we're writing checkpoint blocks, so how do we manage to re-constitute the block states correctly? This is achieved by the following:

•As we write the checkpoint, the checkpoint block allocation selects EMPTY blocks. We don't change the state as we're wrtitng the checkpoint so their state is written to the checkpoint as EMPTY or CHECKPOINT, it does not really matter which.

•After reading the checkpoint, we know which blocks have been used to store checkpoint data. We update those blocks to reflect the CHECKPOINT state.

A checkpoint is invalidated by any modifications (writes, erases,...) thus any modification path also checks to see if there is a checkpoint and erases it.

The regular yaffs block allocator and garbage collection logic also needs to be aware of the approximate size of the checkpoint so that these can ensure enough spare space is reserved to store a checkpoint. There is a calculation that updates the value whenever the number of objects etc changes.

13.5. Extended Tags and Packed Tags

Extended tags is an abstraction of a way to store more information in the tags, allowing for faster scan times. Packed Tags is a physical implementation of the storage of extended tags.

When we're scanning we really only need to build up the structure of the file system. In other words, rebuild the directory relationships, tnode trees and such. Much of the detail such as file names, permissions etc can be lazy loaded.

This required storing extra information, such as the parent directory and object type, in the tags of object headers. But how do we do this without enlarging the tags size?

Object headers always have a chunkId of zero and nbytes has no meaning because it is not a data chunk. By using a single bit to identify object headers we suddenly open up a whole lot of space that we can stuff full of useful data. By being clever as to what data we use, we can pack in the important data in most circumstances.

Sometimes, for example very large files, the packing will not work because there are insufficient bits to hold the field so we just write regular tags instead. That's OK since there can only be few such files in the system so the performance reduction only impacts very few files and the system level impact is unnoticable. A simple, cheap, strategy that gives speed up 99% of the time is a very handy one.

Other details are lazy-loaded on demand. This will happen when the object is looked up (eg. by a file open or searching for the file in the directory).

13.6. Inband tags

Thus far we have considered chunks where there are sufficient unallocated bytes in the spare (or out of band) area to store the tags. This means that the data part of the page is a power of 2 in size. Powers of 2 are very fast to deal with which is why most operating systems use various powers of 2 in their internals.

But what happens if we don't have enough space in the spare area to store the tags? Inband tags to the rescue! With inband tags we store the tags in the flash data area alongside the data. This gives us far more flexibility in the types of storage that yaffs can support, but breaks the power-of-2 alignment. There are three major penalties with inband tags:

- Due to the loss of chunk alignment, all transactions must pass through the shortOpCache. This costs an extra memcpy().
- Offset to chunk, free space and other calculations are no longer just shift and mask operations but instead require

multiplications and divisions which can relatively expensive.

- Reading tags during scanning etc can involve reading the whole page, making this slower. The impact is minimised by using checkpointing.

Thus, it is better, performance wise, to only use inband tags when regular tags will not work.

13.7. Soft Deletion

Soft deletion is a mechanism that was added to speed up file deletion in YAFFS1 mode.

Looking back to the description of how files are stored in YAFFS1, you can see that deleting a file involves writing the deletion marker of every data chunk in the file. This can take quite some time for large files.

The soft deletion mechanism was added to avoid writing all those deletion markers. Instead of writing the deletion markers, soft deletion instead tracks the number of chunks in each block that belong to deleted files and leaves the clean up to the garbage collector.

When the garbage collector looks at a block it can see what current chunks (ie chunks not yet deleted) belong to deleted files and does not copy those, thus effectively deleting the chunk. Once all the chunks associated with a deleted file have cleaned up by the garbage collector the file header object can also be deleted.

YAFFS2 mode does not write deletion markers and thus does not use soft deletion.