Inferno Flash File System: NAND Flash Layout Specification: Release 1.0

Nigel Roles
Vita Nuova Holdings Ltd

Table Of Contents

Figures

Error! No table of figures entries found.

1 Introduction

This document describes how the Inferno Flash File System is mapped onto NAND flash, and how a partition, separate from the log structured file system, is created for the purposes of

1) storing the boot environment variables,
2) storing the kernel image,

and

3) simplifying the boot mechanism from the perspective of the boot loader

2 Usage Of Auxiliary Area

The Inferno Flash File System anticipates that pages of NAND flash have the following tags stored in the auxiliary area of each page:

1. error correcting codes for the main data, and auxiliary data
2. a *tag* or type which distinguishes log/data storage and free storage, from other usage
3. a further field which is tag specific, here called a *path*
4. an indication that the page is unusable

At the practical level there is a need to

1. distinguish between pages belonging to the Inferno Flash File System, and any other which might be implemented within the same device
2. automatically locate the start and extent of the Inferno Flash File System without any external guide
3. accommodate the restrictions on update of flash in general, and NAND flash in particular
4. accurately track the number of erases to facilitate load-wearing

2.1 Bad Blocks

NAND flash devices are designed such that if a page fails, the fault will not affect pages outside of the enclosing block. The suppliers recommend the use of ECCs to detect and correct a single fault, and this is used as a signal to copy the all the data in the block to another block, and to mark the block bad.

It is quite possible for bad blocks to be present in devices as they leave the factory; these are known as *early failure* blocks. The factory marks these blocks by setting byte 5 of the auxiliary area of each page of the block to 0. This is known as the *block status* byte. This is in accordance with the SmartMedia™ or SSFDC (Solid State Floppy Disk Card) Forum specification [2].

If, later, software detects a problem with a block, the block status byte of each page is set to `0xf0` to indicate a *late failure*.

In general, if two or more bits of this byte are set to 0, then the entire block is considered bad. This gives 2 bits margin to protect against stuck at problems[1].

As this mechanism is essentially written into the chip datasheets, we follow the same conventions.

[1] The specifications are unclear as to whether every page of every block should be consulted.

2.2 Error Correcting Codes for Main Data

The SSFDC Forum specify a particular algorithm for protecting the data in each page [3]. The algorithm is used twice, once for each half page, producing a 22 bit result, which is packed into 3 bytes, and stored in bytes 8,9,10 (second half page), and 13,14,15 (first half page) of the auxiliary area.

There is no particular evidence that this algorithm is tuned to the silicon. It has hamming-like qualities, with the ability to correct 1 bit and detect 2 bits in both the data and ECC itself. Some more space is taken than is optimal, but the extra space is traded against ease of implementation.

We might have selected a more compact algorithm if space had been tight in the auxiliary area, but as there is sufficient spare space, we have adopted this algorithm. This might well turn out to be a good decision if hardware implementations of the NAND flash interface and ECC algorithms appear on the market in the future.

2.3 Error Correcting Codes for Auxiliary Data

Omitted from the SSFDC Forum specification is really thorough protection for the auxiliary area. Some fields are protected, some are not. Some you can tell there is an error, but you cannot correct for it. There is nothing special about the auxiliary area; it is no more reliable than the data area, so this can be regarded as a weakness in SSFDC. The Inferno specification has a solution for all the fields.

2.4 Tags

There are two kinds of log structured file system blocks, *log* blocks, and *data* blocks. Additionally, there are *free* blocks. For the NAND flash file system, the tag of the first page of each block is the *block tag*.

For the purposes of creating and reserving space for the boot partition (see later), a 4th kind of block is implemented, the *boot* block.

Further, he design (at one stage) called for intermediate tags for detecting partial updates; i.e. a block would be tagged *log-copying* until it had been fully written, then it would be tagged *log*. If, at initialisation time a block was found to be tagged *log-copying*, a power interruption would be assumed, and the block reformatted. At the moment, this facility is not used, but the codes are defined here for future use.

So, the requirement is for 7 tags; *free*, *log-copying*, *log*, *data-copying*, *data*, *boot-copying*, and *boot*.

As tags can be changed independently of other fields in the auxiliary area, they must be *self protecting*. That is, the hamming distance between any two tags must exceed 2, or otherwise it is impossible to detect and correct a single error. It must be possible to convert the tags (by clearing bits only) along the following chains:

free → *log-copying* → *log*
free → *data-copying* → *data*
free → *boot-copying* → *boot*

A set of seven codes which fit this requirement has been found; these fit conveniently into a byte. There are of course many possible solutions; these are the chosen values. It is an exercise to the reader to verify that the minimum hamming distance is indeed >= 3:

Symbolic Name	Name In C header File	Value
Free	`LogFsTnone`	`0xff`
log-copying	Not used	`0x67`
log	`LogFsTlog`	`0x06`
data-copying	Not used	`0x79`
data	`LogFsTdata`	`0x18`
boot-copying	Not used	`0x1f`
boot	`LogFsTboot`	`0x01`

The *tag* is placed in byte 4 of the auxiliary area.

2.5 Tag Specific Data (a.k.a. path)

Each tag needs some tag specific data; this is referred to as the *path*, but any similarity to Styx Qid paths should not be relied upon. Design constraints suggested that this needed to fit into 32 bits. As with the tag, the path needs to be self-protecting so it can be updated independently from other parts of the auxiliary area. This strongly suggests a hamming(31, 26) code, that is 26 bits of data, 5 bits of protection, and one unused bit. The algorithm is specified as C; see section 5. The resultant 32 bit quantity is stored, big endian, in bytes 0 to 3 of the auxiliary area.

2.6 Magic and Erase count

There is a need for a magic constant to assist in distinguishing these file systems from any others which might be on the device. There is also a need for an erase count which can accommodate up to about 100000 erases for such devices. As it happens, there are 4 bytes remaining in the auxiliary area (though not contiguous), so the same protection defined for the path can be used, giving us 26 bits to play with. Fortuitously, if we use a byte as the magic, that leaves 18 bits for the erase count. This enables us to count 262142 erases which is plenty. Further, if we arrange for the magic to occupy the top 8 bits of the 26 bit data field (before hamming), it will be visible in memory dumps of the NAND device. We use the ASCII character V.

Accordingly, the 26 bit field looks like this

```
01010110eeeeeeeeeeeeeeeeee
```

It is

1. hammed according to section 5,
2. split into most significant word and least significant word

The most significant word is stored big endian in bytes 6 and 7 of the auxiliary area. The least significant word is stored big endian in bytes 11 and 12 of the auxiliary area.

Note that discrimination of Inferno Flash File System blocks from others depends on these 4 bytes passing the hamming code, and the magic field being V. Given that hamming codes can accidentally correct an entirely different value into a V, a corrected V is not good enough as a means to detect the file system. See the section later on file system auto detection.

2.7 Summary of Auxiliary Area

The auxiliary area looks like this:

<table>
<tr><td>Byte/Bit</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td colspan="8">Bits 25– 18 of path</td></tr>
<tr><td>1</td><td colspan="8">Bits 17 – 10 of path</td></tr>
<tr><td>2</td><td colspan="8">Bits 9 – 2 of path</td></tr>
<tr><td>3</td><td colspan="2">Bits 1 – 0 of path</td><td>Unused</td><td colspan="5">Bits 4 – 0 of hamming protection for path</td></tr>
<tr><td>4</td><td colspan="8">Tag</td></tr>
<tr><td>5</td><td colspan="8">Block status</td></tr>
<tr><td>6</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>7</td><td colspan="8">Bits 17 – 10 of erase count</td></tr>
<tr><td>8</td><td colspan="8" rowspan="3">ECC for second half page of data</td></tr>
<tr><td>9</td></tr>
<tr><td>10</td></tr>
<tr><td>11</td><td colspan="8">Bits 9 – 2 of erase count</td></tr>
<tr><td>12</td><td colspan="2">Bits 1 – 0 of erase count</td><td>Unused</td><td colspan="5">Bits 4 – 0 of hamming protection for magic and erase count</td></tr>
<tr><td>13</td><td colspan="8" rowspan="3">ECC for second half page of data</td></tr>
<tr><td>14</td></tr>
<tr><td>15</td></tr>
</table>

3 Format Of Blocks By Tag

Unused bits are set to 1.

3.1 Bad

A bad block is indicated by the *block status* field of page 0 having 2 or more bits set to. If this is the case, the setting of all other bytes in the whole block is undefined, and not to be trusted.

3.2 Free

Free blocks have all the pages set to the same value; within each page all the data and auxiliary fields are set to 1s, except the *erase count*, *magic*, and associated hamming protection. Strictly speaking the *tag* field is set as well, but that is chosen as `0xff` anyway.

3.3 Log/Data

Log/Data blocks have all the fields set. The usage of the *path* is defined elsewhere.

3.4 Boot

Boot blocks have all the fields set. The *path* field of at least the first and last pages have the form

```
llllllllllllllllllllllgg
```

where `gg` is a generation number, and `llll` is the 24 bit boot block number. The generation number increments modulo 4 every time the given boot block is copied as part of an update operation. Having the path the same in first and last blocks, and using a generation number allow a variety of interrupted operation to be recovered from. Boot blocks are numbered incrementally from 0.

The *path* field of pages 1 through last but 1 can be used to store other information. Currently pages 1 through 3 are used; pages 3 through last but 1 also carry the boot block number, but this should not be relied upon by implementations.

Pages 1, 2, and 3 hold the following values:

Page	Value
1	Physical block number on the device where this file system starts
2	Number of blocks used for file system
3	Number of blocks reserved as boot blocks

Thus, so long as a file system includes some boot blocks, it is always possible to automatically detect it, even if it starts at some block other than 0. The redundancy here (this information appears in every boot block) ensures that this configuration information is highly unlikely to be lost, or misinterpreted.

4 Operations

4.1 Formatting

Formatting a file system is straightforward. A single scan of the proposed set of blocks is made. If the block is marked bad in the first page, it is skipped. If the fields in the auxiliary data of the first page pass error correction with at most soft (i.e. one) bit errors, and the magic is ASCII V, then it is assumed that the block was previously formatted to this scheme.

Next, the block is erased, and then formatted. For formatting, the tag is either *free* or *boot*, and the erase count is either that recovered from a previously formatted block plus 1, or 1. The number of *boot* blocks is a parameter to the formatting routine. Typically, the boot blocks are randomly distributed throughout the file system. The generation number for each *boot* block can be set to any value; current implementations set it to 0.

4.2 Initialisation

For initialisation, there are four phases:

Auto detection

The proposed range of blocks is scanned, searching for a block where, for the first 4 pages of the block, the auxiliary error checks all pass without any errors, the magic is V, and the tag is *boot*. This allows the real file system parameters to be extracted from the *path*s of pages 1, 2, and 3

Sanity check

Just in case, the auto detected range of blocks is scanned, counting the number of valid blocks (i.e. pass all error checks without errors), and invalid blocks (i.e. don't pass all the checks, and not marked bad). If the ratio of invalid to valid is less than [10]%, the file system is deemed to be present. As a short cut, if [10] valid blocks have been found, without any invalid blocks, the scan stops

Basic Initialisation

Now the proper initialisation can be completed. The first and last page auxiliaries are read for every block. If either is marked bad, the block is ignored. If the tags don't agree, then the block is considered to have been *partially written*, i.e. the process of writing the block was interrupted by a power or other failure. The treatment of such blocks is tag specific.

If it is a *free* block or a *boot* block, then it is erased and formatted *free* again. The handling of such cases with *log/data* blocks is outside the scope of this document. An additional check is made for blocks that are erased (but not formatted); they too are formatted *free*.

Boot Initialisation

Additional work is required for *boot* blocks. A reverse map is constructed to convert from boot block numbers to file system block numbers. In the process, duplicates may be discovered (left as a result of an unanticipated power failure). These are distinguished using the *generation* number, the newer one being erased

and formatted *free*. The logic here is that unless the old block has been erased, the operation is not complete. If, after these operations, there is a consistent set of incrementally numbered *boot* blocks, the file system is declared open for business

4.3 Reading Boot Blocks

Boot blocks are read a whole block at a time, which typically means 8 or 16K at a time.

The operation is almost as simple as mapping from the boot block number to the file system block using an in-memory table, and then reading the appropriate block.

However, should the *data* ECCs fail, then we must respond as per the SSFDC specifications and mark the block bad. This sub operation is called *block transfer*.

4.4 Transferring Boot Blocks

This refers to copying the data and tags from one block to a carefully chosen *free* block, then erasing and formatting the old block *free* (or sometimes marking the old block bad). After this operation the file system is consistent again, holds the same information, but one block has had an additional write, and one block has had an additional erase (or is now marked bad).

The *free* block is chosen on the basis of having the fewest erases. This does cause some unused blocks to maintain low erase counts which is not so good for the long term life of the device. However, as the boot block interface is intended to be as simple as possible so that it can be implemented in boot loaders, further complication of the block selection algorithm was deemed unnecessary. Instead it is the responsibility of the log structured file system (which is executing in the more full kernel environment) to periodically transfer blocks with relatively low erase counts to prevent this happening.

There are two failure conditions:

1) If the write to the new block fails, then it is marked bad, and a new victim is searched for
2) If the erase of the old block fails, then it is marked bad

4.5 Writing Boot Blocks

Boot blocks are written a whole block at a time, which typically means 16 or 32K at a time. The operation is essentially a block transfer where the data is replaced with new data, rather than the old data.

4.6 Interruption Recovery

This section describes how the file system recovers from interruptions in operation. These are mostly expected to be power failures, but might include software failures.

The fundamental assumption is that page writes and block erases of the NAND device complete before the power fails. A little thought will reveal that this has to be true as nothing about the data stored can be trusted if the power is lost midway through programming.

The design described here can recover from an interruption between any two writes or erases. It all revolves around the block transfer operation previously described.

1) When writing the new block, if the operation is interrupted before all the pages are written, then the partial write detection will catch this
2) When erasing and formatting the old block, if the operation is interrupted between the erase and the first write, then the erased block detection will handle this.
3) When formatting the block free, if the operation is interrupted before all the pages are written, the partial write detection with catch this
4) If a block is determined bad, but there is an interruption before it can be marked bad, then the block will of course be treated as good (but partially written) on resumption. Thus some marginal blocks might earn a reprieve, but the problem will most likely be detected again very soon. Note that if two writes can be guaranteed (which is very likely as they are substantially quicker than a single erase), this will not occur

It is most likely that a platform will include a signal that indicates that the power might fail quite soon i.e. the platform is not sure but it is not safe to risk another write/erase. Thus all writes or erases must be gated with a check of this condition.

If this occurs between pages of boot block write or block format, then the write must be aborted. This leaves the block in an unknown state. It must not be used again without erasing it first. Should the condition pass (i.e. it was a false alarm), then these blocks are now potentially out of commission until a reboot. This can be handled in an implementation by tracking the existence of these blocks and, when the power fail condition has passed, erasing and formatting them *free*.

4.7 Partial Write Limitations

NAND flash devices are very fussy about how many times a page is written to. The limits vary from manufacturer to manufacturer, device to device, package to package. Smaller capacity devices allow 10 writes; larger capacity devices, manufactured by using a die shrink are limited to 3 by one manufacturer, and 2 by another (well 2 to the data area, and 3 to the auxiliary). Thus, writing the data twice, and the auxiliary 3 times looks like a safe bet.

The impact of exceeding this is apparently to induce ECC errors (bit flips) that are not permanent damage (i.e. can be removed by erasing). Unfortunately, these are of course indistinguishable from genuine write/erase wear that is permanent, and requires the block to be marked bad. Thus exceeding the limits is to run the risk of prematurely retiring blocks and reducing the usability of the device. This is not acceptable.

This file system implementation's worst case write is where a block is erased, formatted, written to, the write fails, and the block is marked bad. This results in one write to the data area, and three writes to the auxiliary. We are OK, so long as we do not want to write half pages of data, which is theoretically possible given the individual ECCs. Given that the 'damage' which results from a 4^{th} write is a bit flip, and the *block status* byte has protection against this, a 4^{th} write is probably acceptable, so long as it is to mark the block bad. Toshiba have indicated they consider this acceptable for their devices.

5 Hamming Algorithm for 32 bit fields

The function `_nandfshamming31_26calc()` takes the input 26 bits left justified in a 32 bit unsigned long, and adds the 5 parity bits at the bottom. The unused bit is bit 5. The function `_nandfshamming31_26correct()` corrects the referenced unsigned long, and returns 1 if a correction was made, 0 if the data passed the test.

```
static unsigned long row4 = 0x001fffc0;
static unsigned long row3 = 0x0fe03fc0;
static unsigned long row2 = 0x71e3c3c0;
static unsigned long row1 = 0xb66cccc0;
static unsigned long row0 = 0xdab55540;

static char map[] = {
        -5, -4, 0, -3, 1, 2, 3, -2,
        4, 5, 6, 7, 8, 9, 10, -1, 11,
        12, 13, 14, 15, 16, 17, 18,
        19, 20, 21, 22, 23, 24, 25,
};

#define mashbits(rown) \
        c = (in) & (rown); \
        c ^= c >> 16; \
        c ^= c >> 8; \
        c ^= c >> 4; \
        c ^= c >> 2; \
        c = (c ^ (c >> 1)) & 1; \

static unsigned char
_nandfshamming31_26calcparity(unsigned long in)
{
        unsigned long c;
        unsigned char out;
        mashbits(row4); out = c;
        mashbits(row3); out = (out << 1) | c;
        mashbits(row2); out = (out << 1) | c;
        mashbits(row1); out = (out << 1) | c;
        mashbits(row0); out = (out << 1) | c;
        return out;
}

unsigned long
_nandfshamming31_26calc(unsigned long in)
{
        in &= 0xffffffc0;
        return in | _nandfshamming31_26calcparity(in);
}

int
_nandfshamming31_26correct(unsigned long *in)
{
        unsigned char eparity, parity;
        unsigned long e;
        eparity = _nandfshamming31_26calcparity(*in);
        parity = (*in) & 0x1f;
        e = eparity ^ parity;
        if (e == 0)
                return 0;
        e--;
        if (map[e] < 0)
                return 1;               // error in parity bits
        e = map[e];
        *in ^= 1 << (31 - e);
        return 1;
}
```

6 References

[1] C H Forsyth, "*A Flash File System For Inferno*", Vita Nuova, 27th September 2002

[2] "*SmartMedia™ Physical Format Specifications Web-Online Version 1.00*", SSFDC Forum Technical Committee, 19th May 1999

[3] "*SmartMedia™ ECC reference Manual Ver 2.1 (Software & Hardware)"*, Non-Volatile Memories Engineering Section, Toshiba Corporation, 15th September 1999