Embedding it better...

µTasker Document

µTasker – NAND Flash

Table of Contents

1. Introduction

This document describes the implementation of a NAND flash driver for a 256 Mbit Samsung NAND flash operating together with the µTasker µtFAT file system and an AT91SAM7SE processor.

The particular NAND device is the Samsung K9F5608U0D-P which is a 256 Mbit NAND chip with 8 bit data bus. That means that it is organised as a 32MByte 8 bit device.

It has a page size of 512 bytes with 16 spare bytes. This in fact means that it has 528 byte pages whereby 512 of the bytes are destined for saving data and the 16 spare bytes for saving page error checking/correction data and other management information.

It is possible to individually program bytes within a page (including the spare space) from values of binary '1' to binary '0'. A page cannot be deleted (all bits in the page converted to binary '1') on a page basis but needs to be performed on a block bases.

32 pages are organised into a block. This means that a block contains 16k bytes of page data plus 512 bytes of page spare bytes. 16'384 + 512 = 16'896 bytes in total. NAND content can be deleted on a block basis and so 32 individual pages – with each pages spare bytes - are deleted when a block delete is performed.

A random read from the NAND flash takes up to 15us to be performed. This is due to the fact that the internal location of the byte to be read must first be addressed. Once a byte has been addressed, the internal location pointer is automatically incremented to the next location within the particular page and the next byte can thus be read much faster, requiring no more than 50ns for sequential reads within a page.

The time that it takes to program page content is typically 200us, on top of any addressing overhead. It takes typically 2ms to erase a block.

The first things that become clear from this introductory information is that it is efficient to read sequential data from a single page but less efficient to randomly address individual bytes in the device. Furthermore, it is not possible to change the content of any byte within a block from '0' to '1' without first deleting the complete block. It is however possible to change (write) any already deleted bytes within a page or block.

NAND Flash is not perfect and can have blocks which do not operate correctly. This is also valid for new NAND chips and the manufacturer guaranties that only 2013 of the 2048 blocks in the device are operational on delivery (at least 1004 in each 128M area). Block 0 is however guaranteed to be operational and have a quality to allow it to be written and deleted at least 1'000 times before it shows first signs of deterioration.

Blocks can typically be written and deleted up to 100'000 times, although individual ones may fail earlier and it is to be expected that some blocks are already bad on delivery and others will become bad during normal use of the device.

This means that the use of NAND flash requires that the failure of individual blocks is considered in the design. This means that the data to be stored must be able to fit into the available memory space; that is the available space that is operational after the anticipated numbers of blocks have failed after a certain length of time. It is thus not possible to plan for the storage of 32Meg Bytes of data in a 32Meg Byte device since redundant space must be foreseen.

In applications involving a large number of write/erase cycles it is an advantage to avoid the write/cycles taking place at concentrated block locations which will otherwise fail faster than those in areas with less such activity. By spreading the write/erase cycles across the total available blocks, the life time of the chip can be maximised.

When single bit errors occur in a page this doesn't immediately mean that it is no longer usable. Single bit errors can be easily corrected by using an error correction code stored in the spare bytes area of its page. Multiple bit errors can usually be detected by the error correction code, allowing the block to then be added to the list of bad blocks. The AT91SAM7SE processor not only contains the NAND memory interface for the physical NAND device but also an error correction controller which monitors NAND reads and writes and automatically calculates error correction values to be store, or to be used for checking and/or correction.

A FAT based file system consists of three memory areas: a configuration area informing of the size and location of the other areas (optionally with partitions); the FAT area (file allocation table); data area (storage clusters). The configuration area is usually static and so doesn't change with time. The data cluster use depends on the data being stored, as does the FAT content. The FAT content tends to need updating whenever anything in the data storage space changes and so represents an area of concentrated write/erase operations.

FAT based file systems work with sectors and clusters, whereby a cluster is a multiple of sectors. The typical sector size is 512 bytes and there is usually just one sector in a cluster in a small file system (under several hundred Mbytes in size). The sectors are addressed by numbers and the numbers are fixed values which do not change with time. In the case of the NAND chip used for this discussion, its page size matched the typical section size and so pages and sections are essentially interchangeable. However, since blocks (containing 32 pages) can be, or become, bad blocks it is not possible to directly map FAT sectors to NAND pages. Instead, the FAT works in sectors but the NAND driver must be able to map the sectors to NAND pages in a dynamic fashion; the mapping needs to be able to be changed as bad blocks are detected. The FAT sector content has no restrictions in terms of reading and writing, meaning that the FAT layer may want individual bits within the sector to be modified from '0' to '1', which is not directly possible in the physical NAND page. In this case it is either necessary to completely delete the block that the page is in before writing back the new content or else write the updated content to a free block in the NAND and then remap the FAT's sector to within that block. All such details must be performed by the NAND driver at a lower level than the FAT layer.

The NAND driver thus needs to perform a certain amount of memory management in addition to just controlling the reading and writing of data buffers. The management of the use of block remapping often includes also consideration of a balanced use of the blocks so that write/erase activity is not concentrated but is instead spread out across the physical medium. This activity is typically known as level-wear management and is either dynamic in nature (when new data is written it is written to areas which have had less activity before) or static. Static management also writes new data to already occupied areas with lower past activity – moving the stored data about – in order to also make use of the available cycles in area which otherwise would never be modified. The static operation is more complicated but achieves longer life time by utilising the write/erase cycles of all possible blocks in the device.

This means that the NAND driver has four main tasks:

- read/write of NAND flash. This is based on buffers since the NAND flash is essentially a serial device and cannot be memory mapped.
- Management of bad blocks. A list of bad blocks must be maintained and the bad blocks avoided.
- Address remapping. This is required in order to avoid bad blocks but is also advantageous to dynamically map FAT sectors to NAND pages. It can be further used by level-wear techniques to keep the write/erase load spread over the entire operating memory space.
- Level-wear management.

Read/write: The exact method of reading, writing and deleting (deleting is an inherent part of writing due to the fact that writing '0' bits to '1' bits involves an intermediate delete operation) is NAND device specific and so NAND drivers are not necessarily compatible.

NAND devices are not memory mapped and so can be considered as serial devices. The fact that the K9F5608U0D-P has an 8 bit bus interface means that the content can be transferred 8 bits at a time (more efficient that a single bit interface) but the addressing of the content still needs to be performed specifically.

The K9F5608U0D-P is addressed on a page basis whereby blocks are automatically addressed based on the page number being addressed. The first 32 pages address also the first block. The next 32 pages address the second block, etc. The K9F5608U0D-P allows an additional 8 bit address to be set to locate individual bytes in the particular page. This means that 256 bytes of a particular page can be specifically addressed, which is not enough to address all of the 528 bytes that exist. The solution used by the K9F5608U0D-P is to divide the 528 byte pages into 3 page areas; the A-area contains the first 256 bytes of the page; the B-area contains the second 256 bytes; the C-area contains 16 bytes from the spare area. There are three different commands available to specify in which area the read or write is to take place, allowing the NAND driver to specify any individual address in the NAND device. The three commands are; 0x00 to specify an address in the A-area, 0x01 to specify an address in the B-are and 0x50 for the C-area. Full details of the commands and the internal pointer operation are included in the data sheet of the NAND flash.

Bad-block management: The NAND manufacturer marks bad blocks by setting the 6th byte in the spare area of the first or second page of the bad block to a value not equal to 0xff – all other bytes in a new NAND chip are deleted and so set to 0xff. By reading the corresponding bytes at initialisation it is this possible to known which are good and which are bad blocks and a table of this information can be kept in RAM to act as a look-up table by the NAND driver. Bad blocks are then avoided.

When the NAND driver detects a problem with a good block (errors when trying to write or erase it) it can then add the block to its list of bad blocks and mark it as bad block by writing the same bytes in the first and second pages of the bad block. One subsequent system starts the bad block table is reconstructed based on the page content so that a block marked as a bad block will never be used again.

In the case of the detection of a new bad block the NAND driver may also need to rescue any already programmed data (in pages that are still valid) and copy these to another block before marking the block as bad and thus excluding its further use.

It is important to avoid deleting any bad blocks since this otherwise results in the loss of the bad block information (either pre-programmed by the manufacturer or set by the NAND driver during the normal operational life-time). For this reason, the NAND driver should check that no writes or deletes are started when the block is already marked as bad.

Address remapping: The NAND driver allows the user to specify that a certain sector is read or written (in a FAT it is typical for all operations to be based on a fixed sector size). The sector number will have a fixed value but the physical location can move around with time. This means that the NAND driver needs to maintain a conversion table between the user's sector (practically the user block that the sector belongs to) and the physical sector (block) so that it can read and write to the correct physical NAND pages. Since this information must also be accurate after system restarts it must be based on non-volatile storage, meaning that it can also be stored within the NAND Flash, which is the technique described here.

A complication of the block remapping table being in NAND Flash is that it also has the same restrictions of bad blocks and limits of write/erase cycles. Since it is however the base of the remapping data, containing additional level-wear information in this implementation, it is not possible to include itself within the area that is being managed and use remapping to coordinate itself. The result is that it needs to use a method of inherently reducing write/erase

cycles and also have enough redundancy to be able to accept loss of blocks in its own area. This essentially means that it resides in an area of the NAND flash not under remapping and level-wear management control but with enough space to allow it to spread its own write/erased around and not be impaired when some of this space is lost due to bad blocks during its useable lifetime.

As in the case of bad block management, the remapping requires also moving existing data content from an original physical block to another.

Level-wear management: The NAND driver needs to include level wear management as an integral part of its address remapping operation since remapping causes wear to take place and the remapping actions are this input to the level-wear manager. It is logical that the level wear data is stored together with the address remapping information since it can also be updated together with the remapping information on each change.

To speed up the algorithms used by the address remapping and level-wear management, copies of the most up to date information can also be held in RAM to speed up accesses and searches.

All NAND Flash operations should be executed in a manner that can withstand an interruption in the process without serious corruption of data. An interruption of a particular operation can of course lead to the loss of data (new data that is being added) but should neither lead to the loss of old data nor corruption of remapping or file system information.

2. NAND driver routines:

```
static int _fnReadNANDdata(unsigned short usBlockNumber, unsigned
char ucPageNumberInBlock, unsigned short usOffsetInPage, unsigned
char *ptrBuffer, unsigned short usLength)
```

This is used when reading data from the NAND flash. `usBlockNumber` is a reference to the physical block in the device (0..2047 in this particular case). `ucPageNumberInBlock` is the page within this block (0..31 in this particular case). `usOffsetInPage` is the offset to start reading from (0..527 in this particular case). `ptrBuffer` is a pointer to a buffer where the read data is to be copied to and `usLength` is the amount of bytes to be read. Generally the read length will be up to the end of the page and not beyond it. The NAND driver routine automatically controls the addressing of the area in the page to be read and will perform ECC checking of the content when 512 bytes are read starting from the first location in the page – in all other cases it will not check the ECC since it is not valid. The ECC occupies the first 4 bytes in the spare area (offsets 512, 513, 514 and 515 in the page).

The routine returns `UTFAT_SUCCESS` or `ECC_READ_ERROR`. `ECC_READ_ERROR` is however only returned when an ECC check was made and found to be incorrect.

```
static int fnReadNANDsector(unsigned long ulUserSector, unsigned
short usOffsetInSector, unsigned char *ptrBuffer, unsigned short
usLength)
```

This is used to read from a user sector. The user sector is located in a block in the user area which is physically mapped to a physical block. The user doesn't necessarily know the location of the physical block being worked with.

`ulUserSector` is the user sector number to be read from. The range is from 0 to the maximum user sectors that the file system has been allocated with. `usOffsetInSector` is the offset within the sector to be read from. The value is usually in the range from 0..511 since user sectors are 512 byte in size in this case. However, it is also possible to use the command to read bytes from the spare area, although usually avoided.

`ptrBuffer` is the buffer where the read data is to be copied to. `usLength` is the length of data to be read. Normally the length is not read past the end of the 512 byte sector data otherwise spare area data will be returned – this is usually avoided.

The return values are the same as for `_fnReadNANDdata` and also the ECC operation is the same.

```
static int _fnWriteNANDdata(unsigned short usBlockToWrite, unsigned
char ucPageNumberInBlock, unsigned short usOffsetInPage, unsigned
char *ptrBuffer, unsigned short usLength)
```

This is used to write data to the NAND Flash, whereby the data area to be written should always be known to be blank before attempting.

`usBlockToWrite` is a reference to the physical block in the device (0..2047 in this particular case) to be written to. `ucPageNumberInBlock` is the page within this block (0..31 in this particular case). `usOffsetInPage` is the offset to start writing to (0..527 in this particular case). `ptrBuffer` is a pointer to a buffer where the written data is to be copied from and `usLength` is the amount of bytes to be written. Generally the write length will be up to the end of the page and not beyond it. The NAND driver routine automatically controls the addressing of the area in the page to be written and will automatically write ECC checking data for the content when 512 bytes are written starting from the first location in the page – in all other cases result in an ECC being set since it is not valid.

The ECC occupies the first 4 bytes in the spare area (offsets 512, 513, 514 and 515 in the page).

An attempt to write to a bad block will result in the return value `BAD_BLOCK_CAN_NOT_BE_MODIFIED`. `WRITE_FAILED` is returned when the write failed due to the block no longer being normally writable (in this case the caller needs to handle moving another block and possibly rescuing already saved data from this block).

`UTFAT_SUCCESS` is returned when the write was successful.

```
static int fnWriteNANDsector(unsigned long ulUserSector, unsigned
short usOffsetInPage, unsigned char *ptrBuffer, unsigned short
usLength)
```

This is used to write to a user sector. The user sector is located in a block in the user area which is physically mapped to a physical block. The user doesn't necessarily know the location of the physical block being worked with.

`ulUserSector` is the user sector number to be written to. The range is from 0 to the maximum user sectors that the file system has been allocated with. `usOffsetInSector` is the offset within the sector to be written to. The value is usually in the range from 0..511 since user sectors are 512 byte in size in this case. However, it is also possible to use the command to write bytes in the spare area, although usually avoided.

`ptrBuffer` is the buffer from where the data is to be copied from. `usLength` is the length of data to be written. Normally the length is not written past the end of the 512 byte sector data otherwise spare area data may be corrupted – this is usually avoided.

The return value are the same as for `_fnWriteNANDdata` and also the ECC operation is the same.

```
static int _fnEraseNAND_block(unsigned short usBlockToErase)
```

This routine erases a complete block (32 pages in this particular case).
`usBlockToErase` is the physical block number to be deleted. This function is not usually used by a file system since file systems only read and write sectors, whereby writing to all 0xff is equivalent to a sector erase.

3. Bad-block table:

Samsung delivers the NAND devices in a deleted state. That means that all data is read at as 0xff. However, any bad blocks that exist in the device on delivery are marked as such by setting the 6th byte of one or both of the first pages spare area in the defective block with a value not equal to 0xff. This means that the NAND driver can establish its own bad-block list on each start-up by reading these two byte locations in each block and then creating a table in RAM for look-up purposes.

The table is an array of bytes (referenced by ptrBadBlocks on HEAP) which is initially zeroed. As each bad block is detected during the initialisation its physical block location in the set with a value 1. This means that it is very efficient for the NAND driver software to verify that a block is bad by checking with if (ptrBadBlocks[physical_block_number] != 0). This acts as an efficient look up table.

Should bad blocks be detected during operation the location in the bad block table is simply set with 1 so that it can then be avoided. At the same time, the same locations in the spare area of the first two pages of the bad block are set to 0x00 so that this will also be detected the next time that the system is started.

When bad blocks are detected and marked during normal operation there may be data that needs to be rescued. Details about rescuing this data are given later in the document.

4. Address mapping and level-wear storage area:

Since it is not possible for a file system to write any data pattern to erase individual sectors (pages) the NAND driver needs to be able to move the data around between different blocks to achieve the equivalent operation. The file system itself sees just sectors which it can read and write as it desires without knowing anything about the underlying details.

This block, or address, remapping requires a table which can be used to keep track of which physical NAND flash blocks are mapped to file system (user) sectors.

At the same time it is advantageous to have a table keeping track of how many times individual blocks within the user area have been deleted so that the erase cycles can be actively spread around the available NAND Flash user block area.

The implementation groups both of these task together in order to be able to keep track of which physical NAND Flash block is being used by a particular user sector and how often this block has already been erased.

This information is contained both in RAM for fast access and fast updating and in NAND Flash so that it is not lost when system power is removed or a reset occurs. It is however not practical to perform similar address mapping and level wear management on the area of NAND Flash being used to store this type of information and so redundancy is used instead. The technique chosen is based on minimising the number of erase operations in the address mapping area so that it can handle as many update cycles as required during the life-time of the device. The lifetime of the device being considered as more than 100'000 erase cycles per block in the user area and a certain percentage of blocks in that area failing.

The remapping area is divided into two sub-areas – the main remapping area and a remapping swap area. The swap area is used only infrequently to make a consolidated backup of the most up to date mapping information while the main remapping area is cleaned (erased due to it being filled up).

Redundancy in the main remapping area is achieved by saving data values more than once so that a certain level of errors can be corrected – no ECC is used since the data is not saved on a page basis but instead written in smaller amounts as required.

The swap block is smaller since it doesn't need to save as much data and also makes use of ECC techniques during the short amount of time that it is used.

Additional space is also reserved in the remapping area to account for bad blocks that may also develop with time and use.

4.1. Main remapping area:

The main remapping area consists of a number of NAND Flash blocks. The greater the number, the more reserve it has in case of bad block development and the less amount of erase cycles need to be withstood during the life-time of the device.

Initially the area is deleted and the first remapping table is constructed as follows:

User block 0 remapping information

```
0x00000000 0x00000000 0x00000000 0x00000000
0x0000 0x0000 0x0000 0x0000 0x0024 0x0024 0x0024 0x0024
```

User block 1 remapping information

```
0x00000000 0x00000000 0x00000000 0x00000000
0x0001 0x0001 0x0001 0x0001 0x0025 0x0025 0x0025 0x0026
```

Etc. until all user blocks have an entry.

Following the initial remapping information there is a free area of NAND Flash which will be used to inform of changes to the original information once this is needed.

The entries shows 4 copies of the delete count of each block (double words) followed by 4 copies of the user block number (16 bit words) then 4 copies of the physical block mapped to the user block (16 bit words). In the case of errors in one or two of the copies the original value is still known and so this replaces any ECC control of a compete page.

Originally the sequence is from user block 0 to the final user block in the user block area but as the block get exchanged the information needs to be replaced.

When the NAND driver starts it creates its own mapping and delete count tables based on the information read form the remapping area, whereby the last entry found is the one that it considers to be valid.

As an example, if the blocks 0 and 1 are to be exchanged these need to be re-written to the end of the remapping data (assuming enough space to do this). This means that the NAND driver will initially find the very first entry and then later, before reaching the end of the remapping information, new mapping information which will take precedence over the first. This avoids any need to delete old remapping information until the complete remapping area is exhausted.

When exchanged information data is written to the NAND Flash it is in fact written in the reverse order - first the second block to be added and then the first to be added. The reason is in case of a power loss or reset before the new data could be saved and so this case can be detected due to an empty entry before the end of the data, meaning that the exchange had not been completed and so is not to be taken as being valid.

Once the remapping area is exhausted the NAND drivers set of data (a consolidated set) is written to the swap area. This allows the main remapping area to be completely deleted without risk of remapping data loss due to power down or reset. The swap area uses ECC on the data and so doesn't need further duplication. If a block in the swap area fails there should be reserve blocks that can be used instead.

This in fact means that the NAND driver initialisation doesn't just involve creating a working remap table in RAM from the main remapping area but also checking whether there is valid data in the swap area. The swap area takes preference if it is valid and the main remapping area is only used when there is a further remapping change, when the initial remapping information is written to the main remapping area from the swap area (with the new changes) and only then is the swap area deleted ready for its next use in the future.

The following show a calculation for the dimensioning of the 2048 block NAND Flash being discussed, whereby it is assumed that the end of life of the device is reached when 10% of the blocks in the NAND Flash are bad:

The user area consists of (2048 – remapping blocks) and each of its blocks should be deleted 100'000 times (assumed spread over all blocks). A block in the user area is deleted every time a block exchange takes place, whereby the information written to the remapping information area is 64 bytes in size (2 entries with redundancy). The remapping information takes up about 2048 x 32 bytes of space in the remapping area (it is in fact a little less since there are less user blocks used), or 4 blocks. An extra block in the remapping area is also written after every 256 user block swaps (equivalent to this many user block erases) [16k block size / 64 bytes per entry]. A remapping area delete takes place once every time that the remapping area becomes full and so is dependent on the number of blocks reserved for this purpose. If it is 1 extra block in size it will be deleted once every 256 user block deletes, whereby each user block can be deleted 100'000 times. If there were 2048 user blocks, each being deleted a maximum number of times, it would need to withstand a total of 204'800'000 / 256 = 800'000 erase cycles, which is too high. With 2 extra blocks this is reduced to less than 400'000. With 4 extra blocks to less that 200'000 and with 8 extra blocks to less that 100'000.

To allow extra reserve for bad blocks developing in the remapping area a value of 16 extra blocks (18 in total) results in the remapping area being subjected to less that 50'000 erase cycles during the life time of the user area, where each block is subjected to 100'000 erase cycles.

The swap area is also subjected to less that 50'000 erase cycles, whereby it has the advantage for ECC, but needs redundant blocks to move on to in case the first block does fail prematurely.

Since blocks in the swap area can also fail prematurely (bad blocks) an extra 50% is added, resulting in 27 blocks – rounded up to 32 for practicality.

8 swap blocks are used for safety, whereby it is advisable to check that a new part doesn't has a high density of bad blocks in the swap block area, making it inherently unsuitable for such use.

Note that the NAND Flash device used for development had no bad blocks as delivered!

4.2. Swap area

The swap area's content is a consolidated version of the present mapping and level-wear tables. If a valid version is present it takes precedence over any data in the remapping area. The remapping area will be deleted in case it is not completely blank.

The content is build up as follows:

0x0123 [deletes of user block 0 – high word]

0x4567 [deletes of user block 0 – low word]

0x0002 [physical block used by user block 0]

0x6543 [deletes of user block 1 – high word]

0x4567 [deletes of user block 1 – low word]

0x0743 [physical block used by user block 1]

Etc.

Note that the user block reference is not required since it is built up from user block 0 to last user block. Therefore a user area of 2048 blocks (it is always smaller that this value) would require 2048 x 6 bytes = 12k, which fits in a single NAND Flash block. The swap block therefore uses only one block (starting with block 0, which is guaranteed to be a good block on delivery and also error free up to at least 1000 erase cycles) and only moves on to using another in its area should the first become defective. In its life-time it shouldn't not need to withstand more than 50'000 delete cycles.

Each page is written as a 512 page area and so uses ECC.

Each time the system starts it first checks to see whether there is a valid swap area. If there is, it copies the content to two tables; a remap table (`unsigned short *ptrRemap`, pointing to a list of user block remaps of length equal to the user space) and a delete count table (`unsigned long * ptrLevelWear`, pointing to a list of user delete counters of length equal to the user space). A user block remap to a physical block is a simple look up of the user block in the table: `physical_block = ptrRemap[user_block];` To check the number of times the physical block remapped to a user block has been deleted – `number_of_physical_deletes = ptrLevelWear[user_block]`. Since the remapping of user blocks to physical blocks can change with time both the physical block number and the delete count can change with time.

In order to simplify swapping blocks during file system operation a third table is maintained; a blank block table (`unsigned char * ptrBlankBlocks`, pointing to a list of blank user blocks of length equal to the user space). This table is initialised based on the content of each user block and the table is also changed each time a user block is deleted or written to.

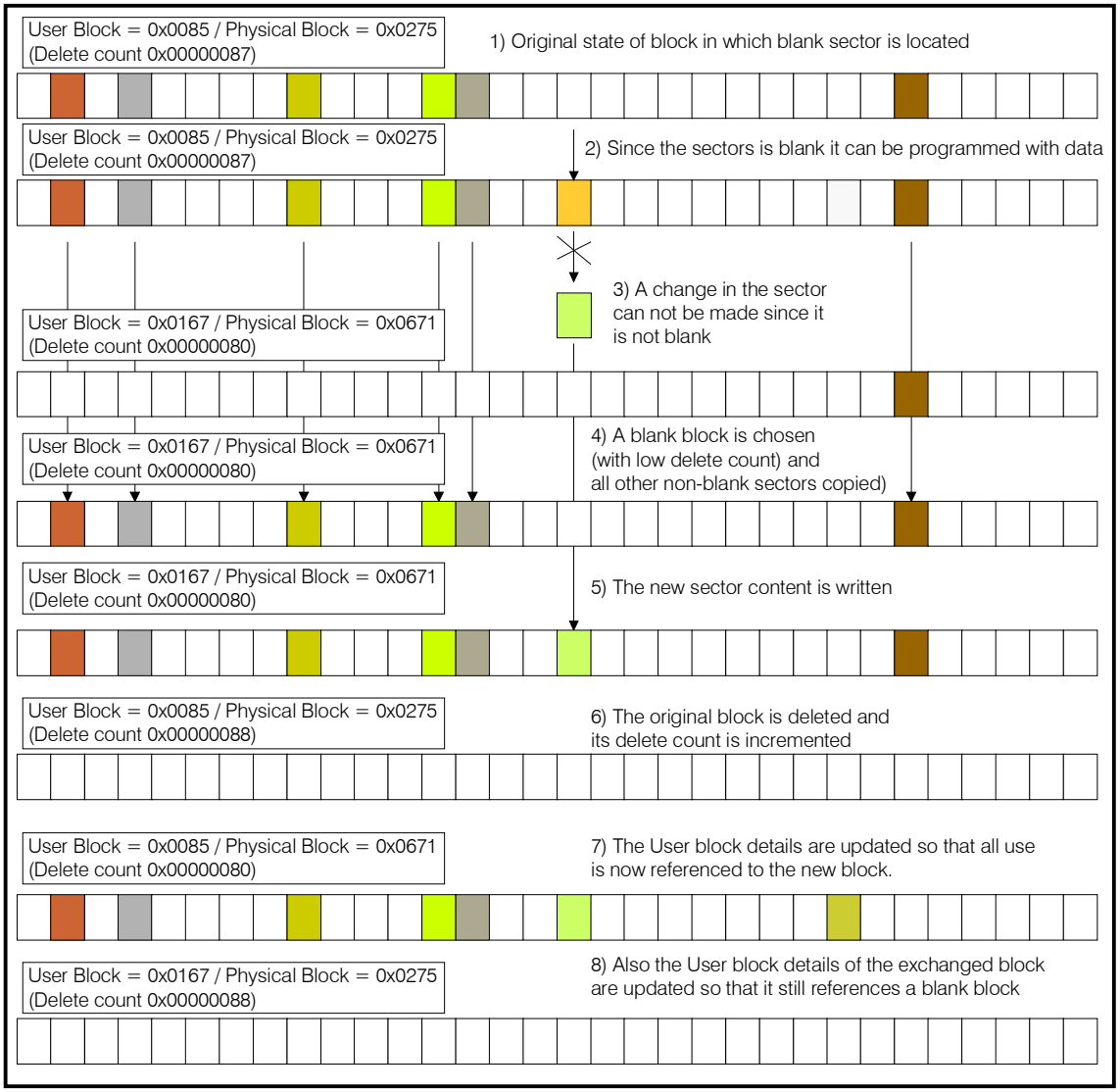
5. User sector writes:

The file system works with sectors (512 bytes), which it can read or write as sectors of this size.

If a sector is read it implies that the data from a page (excluding spare area bytes) is read.

If a sector is written it means that a page write is performed, with an ECC value to the spare area. However a write can only be performed to a page which is blank (all bytes in the page are 0xff). In the blank case a simple page program can be performed which takes about 200us to complete.

Should the page not be blank it is generally not possible to program to it since a program operation cannot change bits from '0' to '1'. A delete operation followed by a program operation is also only limited due to the fact that this will delete all 32 pages (user sectors) in the particular block that they reside in. For this reason a write to a user sector which is not already deleted results in a physical block swap as shown in the following diagram:



Whereas it is possible to program a blank page without any difficulty it is seen that changing the content of an already programmed page results in several steps (4 to 8):

- Up to 32 other sectors in the block need to be copied to a blank block (31 x 200us = 6.2ms)
- The original block is deleted (2ms)
- The updates remapping information is programmed to the remapping area (64 bytes) about 200us.
- If the remapping area becomes full a swap must be made which involves writing the swap block (about 4.8ms) and the remapping area deleted (about 64ms if performed in one go)

During the process no further file system operation is possible.

6. Monitoring the NAND Flash data content through the Command Menu

As well as the usual utFAT command line interface which allows sectors to be displayed with the “sect” command, there are additional commands available when NAND Flash is used.

“secti” supplies information about the sector (the physical block that it is mapped to and the number of deleted that that block has been subjected to).

“page” displays the physical page in NAND Flash plus the 16 bytes of spare data. The first 32 pages (0..31) reside in the first physical block in the chip, the pages 32..63 reside in the second block, etc. Whereas the “sect” can only display the content of the pages mapped to user sectors the “page” command can display all pages in the NAND Flash, including those in the swap and remapping areas.

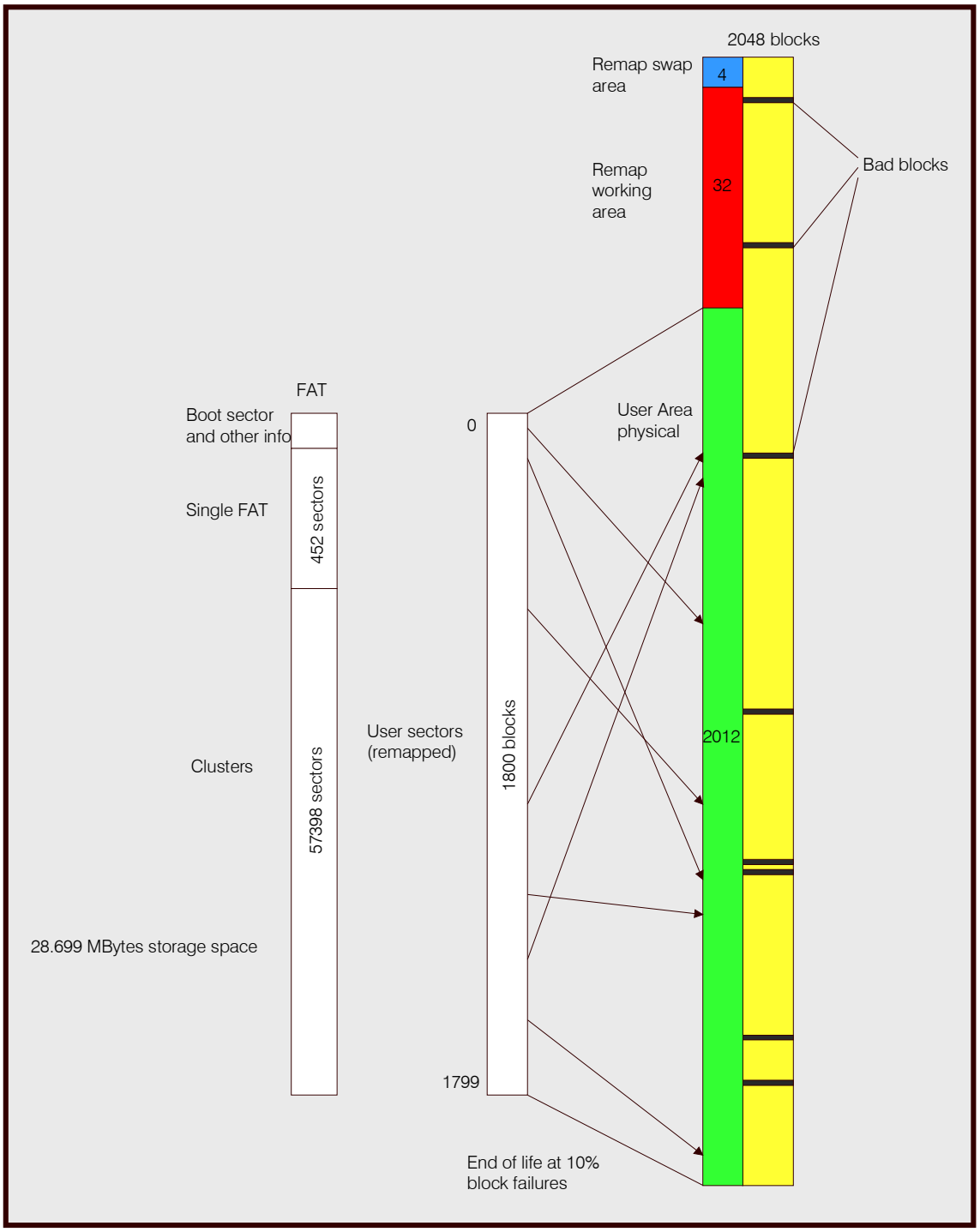
As long as the project define VERIFY_NAND is active low-level NAND tests are available as follows – the test will however corrupt any existing FAT formatting and data saved in the NAND Flash and should therefore only be used for verification of low level NAND driver operation.

“tnand” writes the values 0x0000, 0x0001, 0x0002 etc. to the first two bytes of the first page of each NAND block in the entire user block area.

“vnand” reads all user blocks in the order 0, 1, 2, etc. and verifies that the first bytes of the content also has the same ordering. The test can thus detect any errors in the swap mechanism.

Furthermore, when VERIFY_NAND is enabled the message “NAND cleaned” is sent to the debug output when all non-deleted blocks in the free-block area are completely deleted by the background operation after use.

7. File System Dimensions



This figure shows an example of a possible configuration for 32M NAND Flash where the user block is dimensioned to tolerate a useful life time up to the point when 10% of the NAND Flash blocks fail. This gives a usable data space in the FAT of 28.699 Mbytes.

8. Conclusions

This document has described the implementation of a FAT file system in NAND Flash including swap block and level wear management. It focuses on an example case with 32MByte NAND Flash from Samsung using the AT91SAM7SE with NAND Flash controller, with advantage of ECC calculation in hardware. The principles are however valid for general NAND Flash use. In addition the swap block and level wear management technique could be used with different medium, such as SPI based data Flash (in particular AT45Bxxx devices with 264 byte pages and multiples thereof).

Modifications:

V0.02 20.3.2011 – first published version

Important: Please read the disclaimers at the end of this document. The use of the µtFAT module implies their acceptance in their entirety.

Disclaimers

The information contained in this document is presented only as an overview of NAND Flash and FAT and is provided "AS-IS" without any representations or warranties of any kind. No responsibility is assumed by the µTasker developers for any damages or infringements of patents which may result from its use. No license is granted by the µTasker developers implication, estoppel or otherwise under any patent or other rights of the µTasker developers or any third party. Nothing herein shall be construed as an obligation by the µTasker developers to disclose or distribute any technical information, know-how or other confidential information to any third party.

The µtFAT module is offered "AS-IS" for users of the µTasker project for hobby and/or commercial work. In the case of commercial applications a basic µTasker commercial license for the used processor is implied. The module is supported by the µTasker developers and all attempts will be made to correct any operation which proves to be faulty but the licensee/user accepts that no claims for compensation may be made, for any reasons whatsoever, whether due to µTasker code, its environment and tools or use thereof.

To this effect please ensure that development work is not performed with medium containing important data – potential data loss can be simply avoided by using such medium reserved exclusively for experimental and development work; please adhere to this simple rule and have fun with the µtFAT module!