Understanding UFFS

Ricky Zheng
< ricky_gz_zheng@yahoo.co.nz >

Created: March 2007
Last modified: Nov 2010

Content

- Why UFFS ?
- Design goal
- Flash: NOR vs NAND ?
- What's wrong with FAT ?
- UFFS basic idea
  - Serial number
  - Tree in memory
  - Journalizing
- UFFS architecture
  - UFFS device
  - Mount point
  - UFFS nodes tree
  - Mounting UFFS
  - Page spare/UFFS tags
  - Block info cache
  - UFFS page buffer
  - Block recover
  - Bad block management
  - How ECC works ?
  - Flash interface
- What's next ?
  - UFFS2

Why UFFS ?

- JFFS/JFFS2
  - Can't go out of Linux/MTD
  - Memory monster
- YAFFS/YAFFS2 still consumes too much RAM
  - 64M FLASH, 500 files ==> 410K RAM
- No open source lightweight flash file system exists yet ...

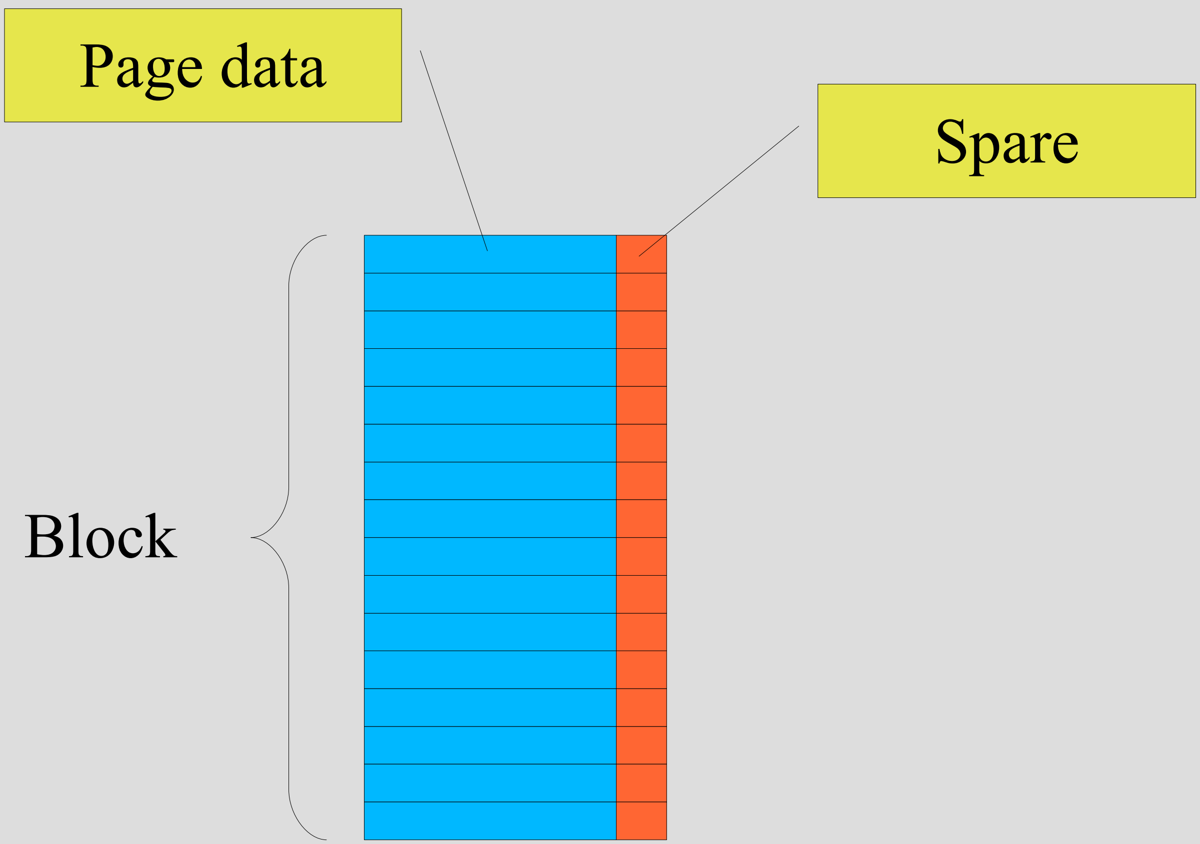
UFFS design goal

- Ultra low cost
  - Low memory cost
  - Fast booting
- Superb Stability
  - Constant RAM consumption, support static memory allocation
  - Guaranteed integrity across unexpected power losses
  - Bad block tolerant, ECC and ware leveling
- NAND flash friendly
  - Support small (512B) or large page (up to 4KB) size
  - Support SLC and MLC
  - Support software ECC or hardware ECC / RS-ECC
  - Direct flash interface
- Well Emulated on PC platform, easy for debugging

Flash: NOR vs NAND

- NOR:
  - Random access for read
  - Big block (minimal erase unit)
  - Byte programing
  - Slow erasing/programing
- NAND:
  - Page/spare access for read
  - Small block
  - Page/spare programing (with limited splits/Restricted rewrite)
  - Fast erasing/programing
  - Delivered with bad blocks

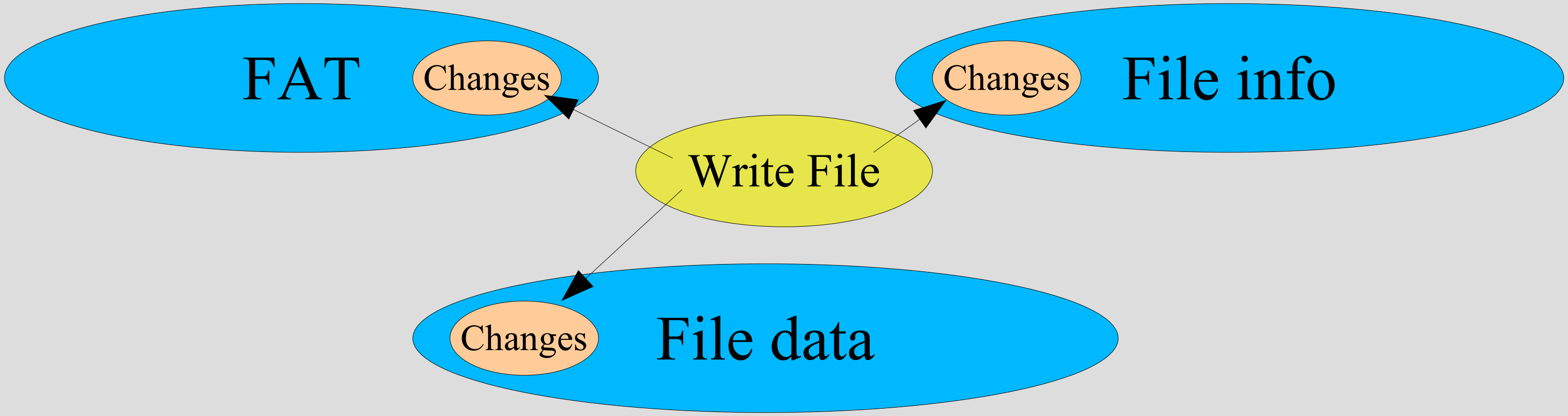
NAND Flash Basic



Erase: '0'->'1',   Write/Program: '1'->'0'

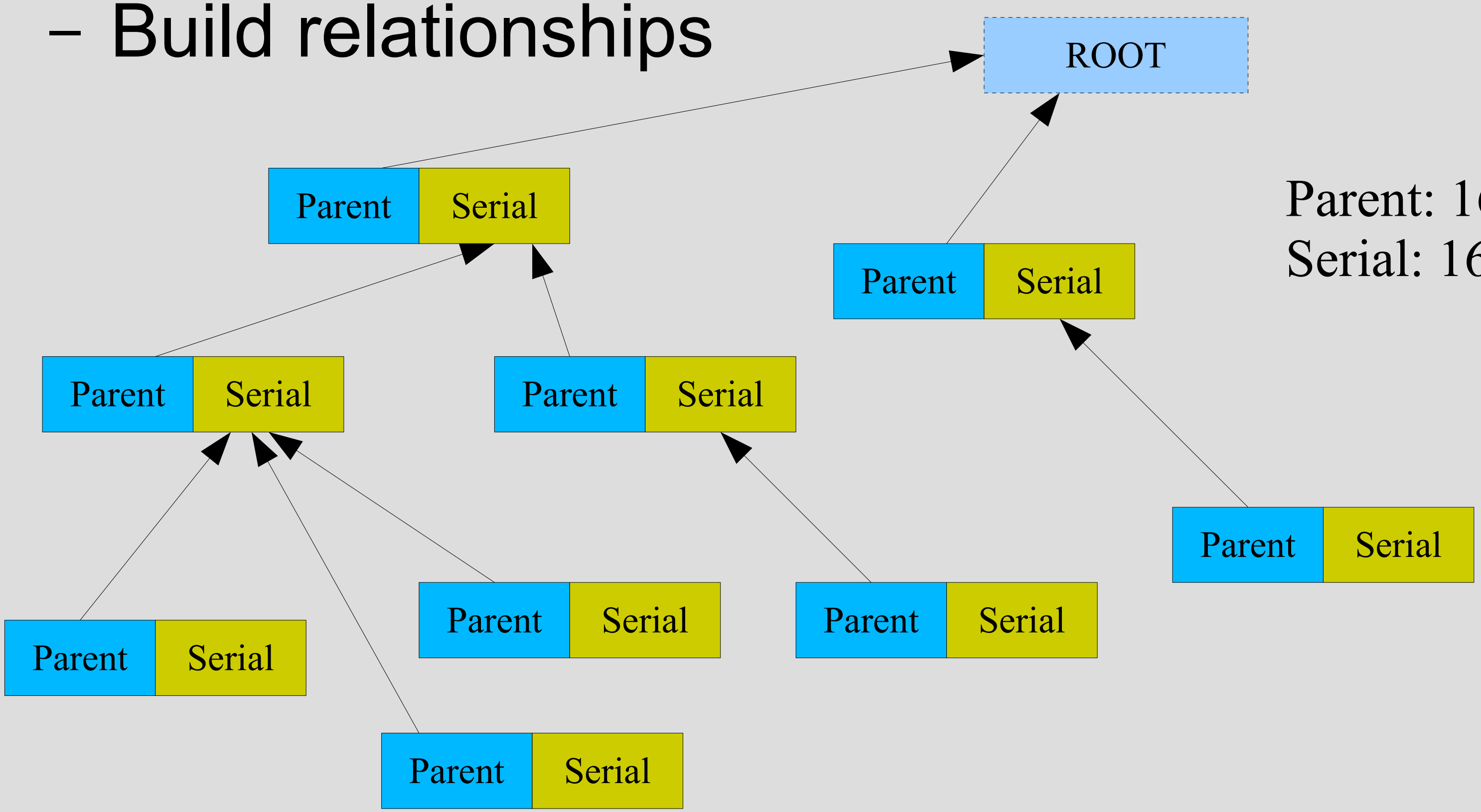
What's wrong with FAT

- Need FTL (which may cost many RAM)
- Big FAT table,slow down the whole system
- Vulnerable when unexpectedly interrupted while updating FAT or File info



UFFS basic idea(1)

- Use unique parent/serial number pair to:
  - Identify blocks
  - Build relationships



Parent: 16 bit
Serial: 16 bit

UFFS basic idea(2)

- Build the relationship tree in memory when mounting UFFS:
  - Erased blocks
  - Bad blocks
  - Hash tables (serial number as key)
    - Dir table
    - File table
    - File data table
- Tree node size: 16 bytes
  - Memory cost: 16 * total_blocks

UFFS basic idea(3)

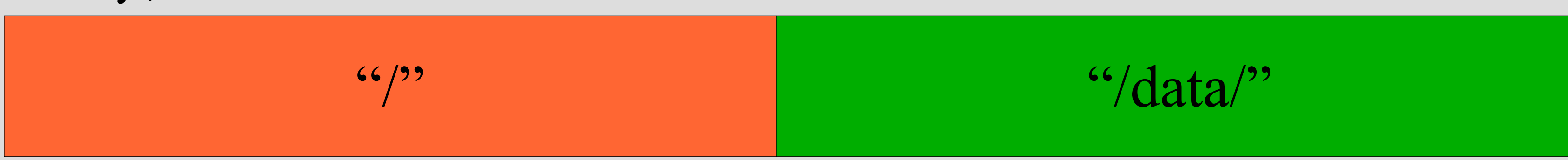
- Journalizing
  - Write to a new block/page instead of modify the old one.
  - Use circular time stamp: 00->01->11->00>...
  - Check and correct conflicts while mounting UFFS
  - Using “mini-header” eliminates partial page programing requirement

UFFS Device

- UFFS Device & Mount Point

```
extern uffs_Device uffs_rootDev;
extern uffs_Device uffs_dataDev;

static struct uffs_mountTableSt
femu_MountTbl[] = {
    {&uffs_rootDev, 0, 200, "/"},
    {&uffs_dataDev, 201, -1, "/data/"},
    {NULL, 0, 0, NULL},
};
```

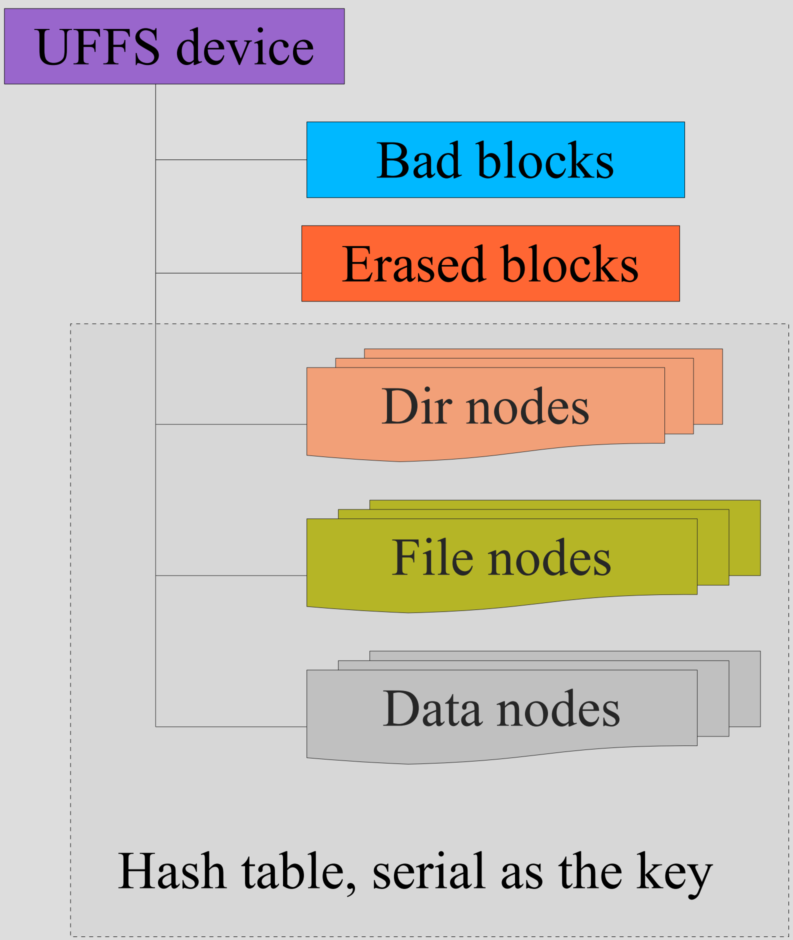


UFFS Device ===> Partition

UFFS Device: individual flash ops, cache/buffer, tree nodes ...

UFFS node tree

- UFFS nodes tree



block,next

block,next

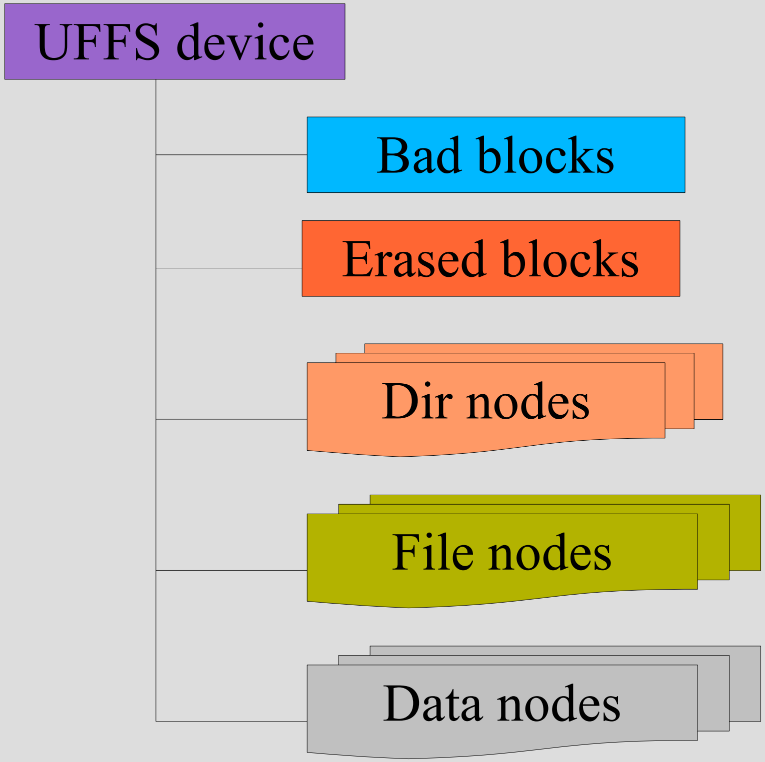
block,parent,serial,sum,next

block,parent,serial,sum,length(4),next

block,parent,serial,length(2),next

sizeof(TreeNode) = 16B

UFFS Mounting



- Mounting UFFS

Step 1:
  - Scan page spares*, classify DIR/FILE/DATA nodes
  - Check bad block
  - Check uncompleted recovering

Step 2:
  - Randomize erased blocks

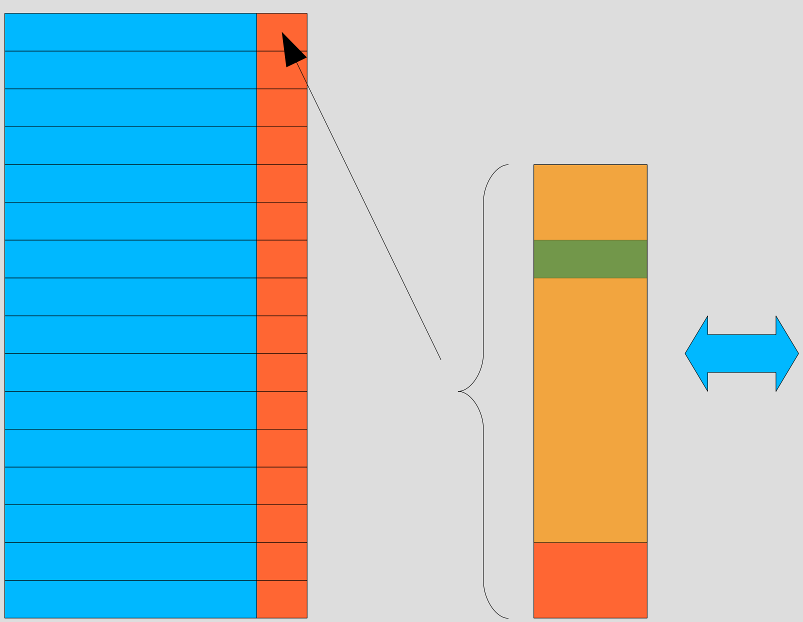
Step3:
  - Check DATA nodes,take care orphan nodes

Super fast !

** UFFS only read a few spares (normally one or two) from each block rather then all spares !!*

UFFS tags

- Page spare/UFFS tags



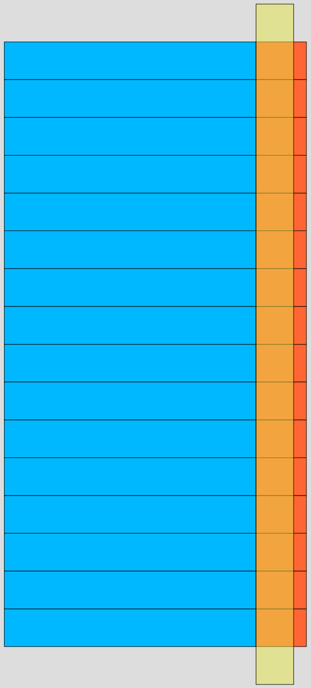
```
struct uffs_TagStoreSt {
        u32 dirty:1;
        u32 valid:1;
        u32 type:2;
        u32 block_ts:2;
        u32 data_len:12;
        u32 serial:14;
        u32 parent:10;
        u32 page_id:6;
        u32 reserved:4;
        u32 tag_ecc:12;
};
```

sizeof(struct uffs_TagStoreSt) = 8, small enough to store on spare area

** Note: if using RS-ECC on small page MLC, then tag may store on page data area*

UFFS block info cache

- UFFS block info cache



```
struct uffs_pageSpareSt {
	u8 expired:1;
	uffs_Tags tag;
};
struct uffs_blockInfoSt {
	struct uffs_blockInfoSt *next;
	struct uffs_blockInfoSt *prev;
	u16 blockNum;
	struct uffs_pageSpareSt *spares;
	int expiredCount;
	int refCount;
};
```

uffs_config.h:
MAX_CACHED_BLOCK_INFO(5 ~10)

Memory: 40 bytes for each cached info

UFFS page buffer

- UFFS page buffer

```
struct uffs_BufSt{
	struct uffs_BufSt *next;
	struct uffs_BufSt *prev;
	struct uffs_BufSt *nextDirty;
	struct uffs_BufSt *prevDirty;
	u8 type;
	u8 ext_mark;
	u16 parent;
	u16 serial;
	u16 pageID;
	u16 mark;
	u16 refCount;
	u16 dataLen;
	u8 * data;
	u8 * header;
};
```

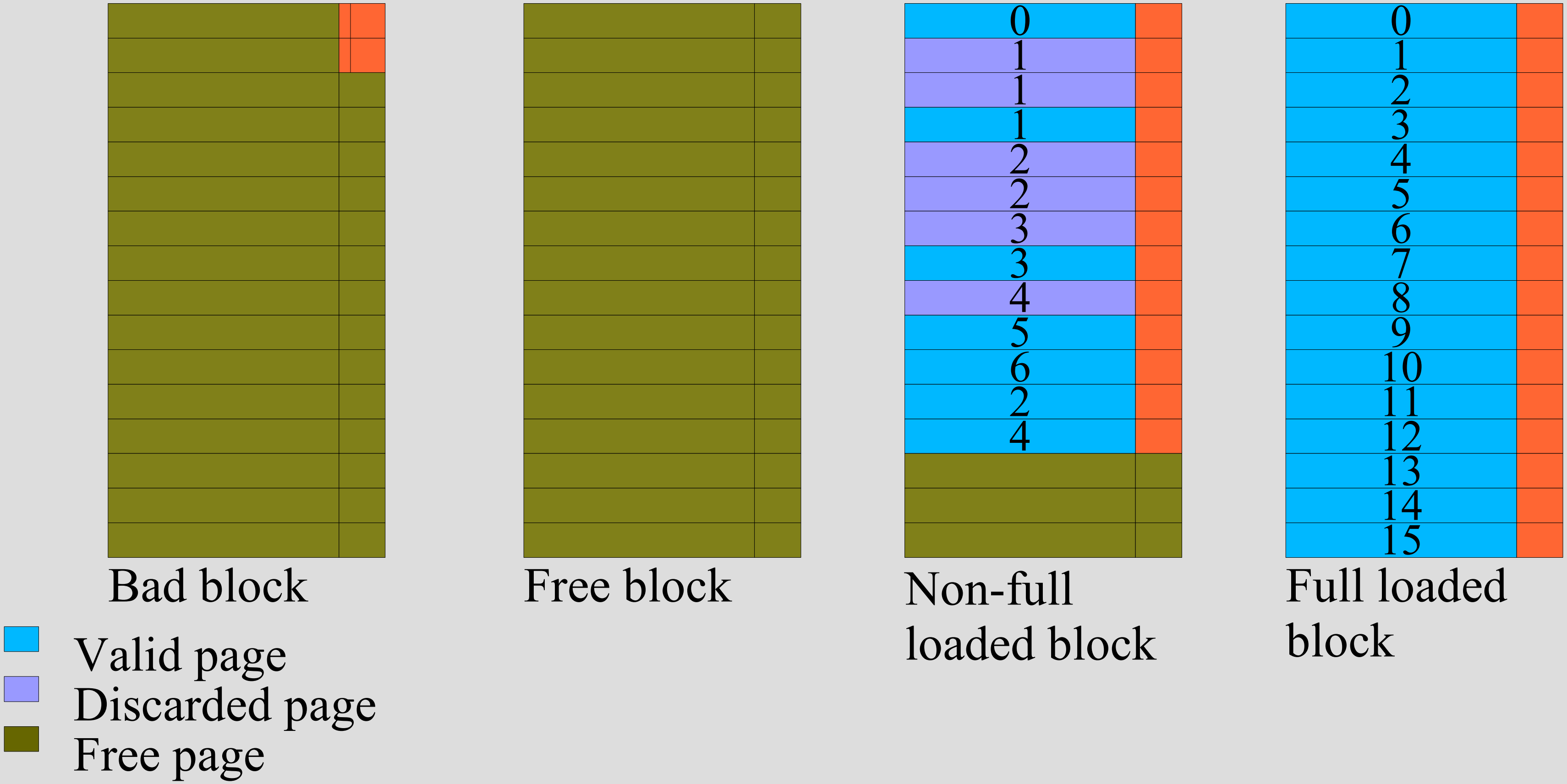
uffs_config.h:
MAX_PAGE_BUFFERS (10 ~ 40)
Memory: (40 + page_size) each buffer

UFFS page status

- Free page: no page id assigned yet. Free pages are always on the bottom.
- Valid page: the page with a id and have max page offset
- Discarded page: the page with page id, there are one or more pages have the same id and bigger page offset.
- Unknown status: interrupted while writing a page.

UFFS block status

- Bad block
- Free/Erased block
- Non-full loaded block (have one or more free pages)
- Full loaded block (no free page, page id = physical page offset)

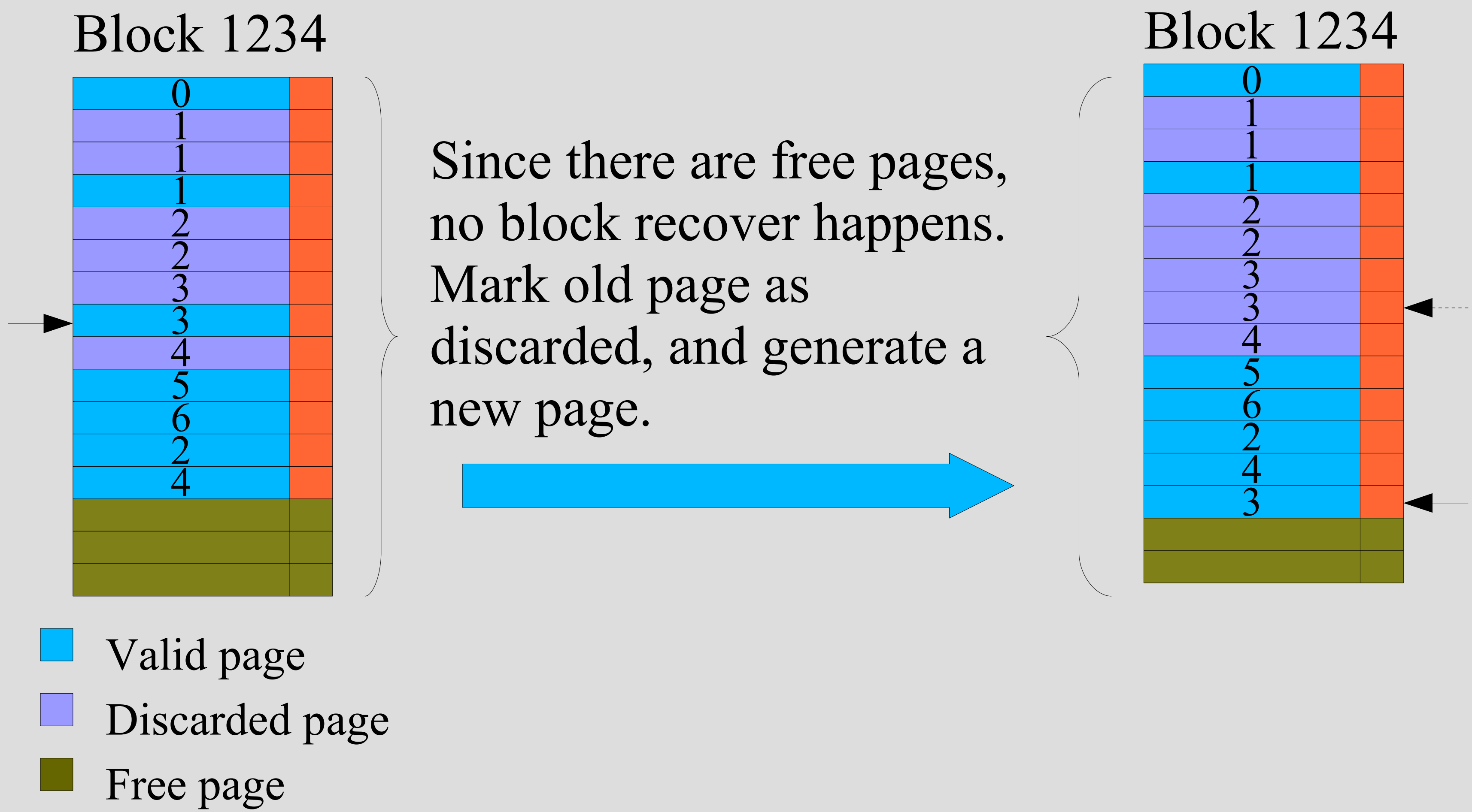


UFFS block recover(1)

- Block recover happens when:
  - No more free pages available inside the block and
  - Data were modified and/or
  - Flush the buffer
- Block recover steps:
  - (1)Get a free/erased block from erased block list
  - (2)Copy pages from old block, write to new block with newer timestamps
  - (3)Erase the old block
  - (4)Put the old block to erased block list
  - Note: (1) and (4) are operating in memory. (2) and (3) identified by timestamps, all steps allow to be interrupted at any time ! (Guaranteed integrity across unexpected power losses)

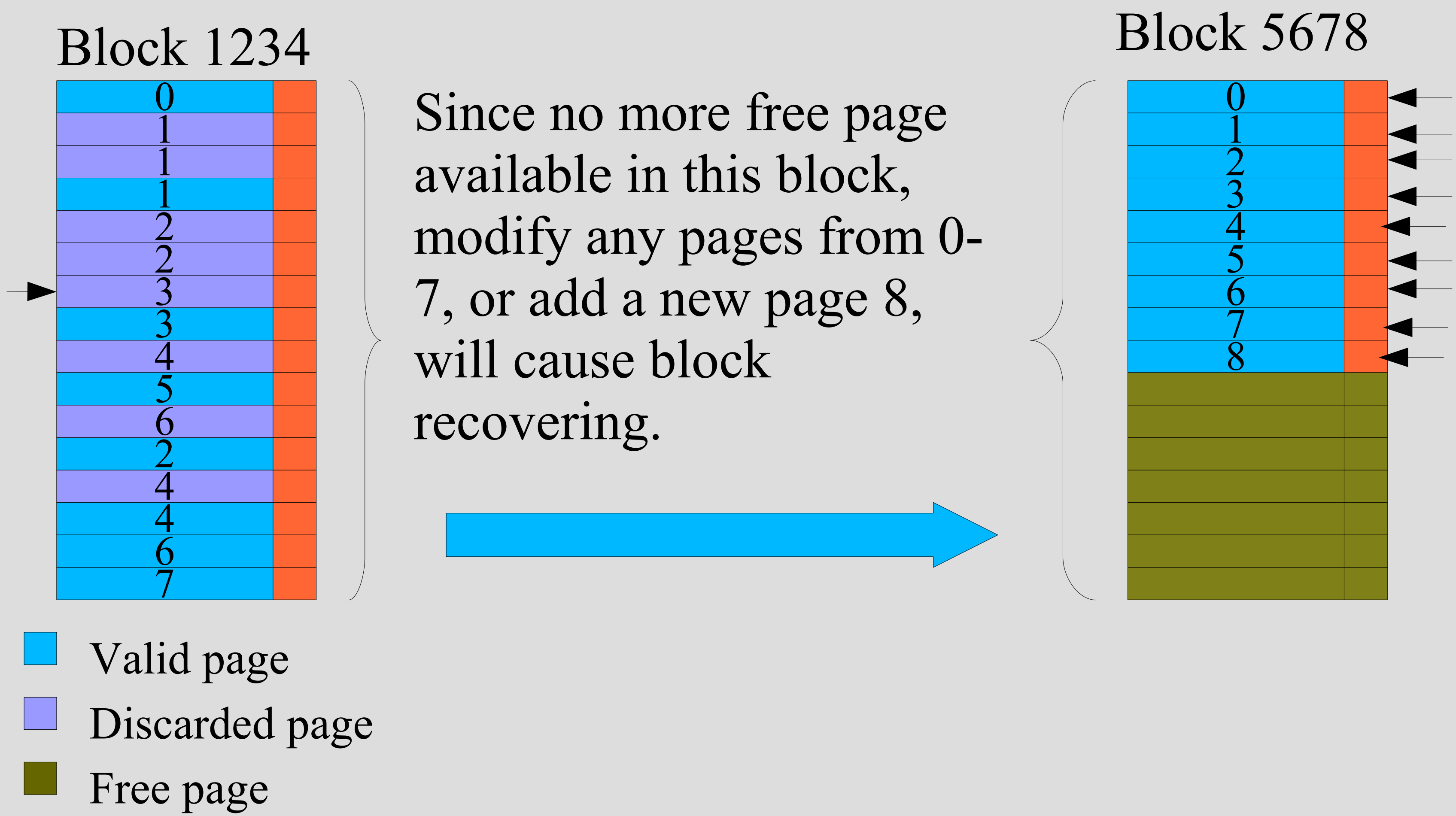
UFFS block recover(2)

- No block recover if there have enough free pages



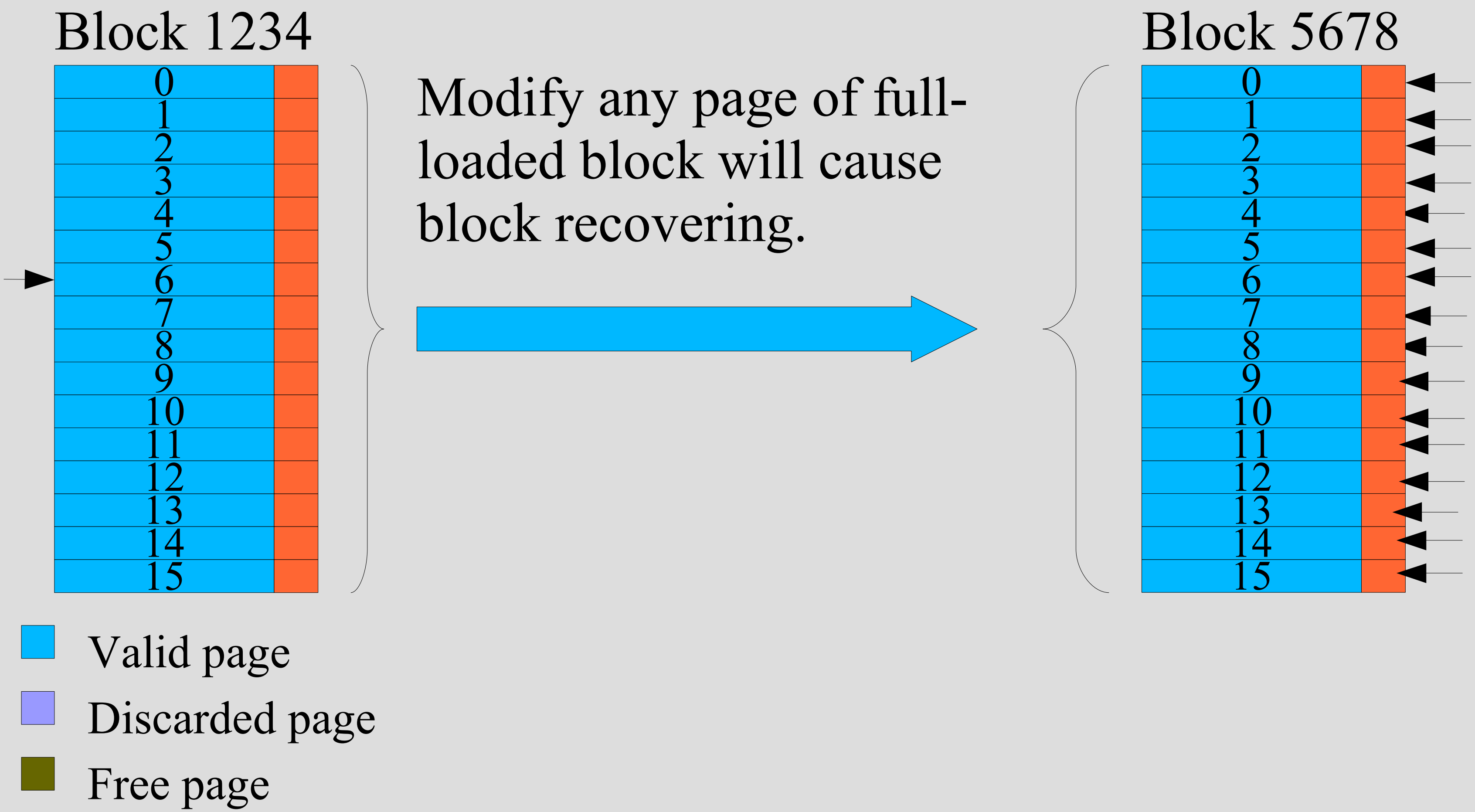
UFFS block recover(3)

- Recover a non-full loaded block

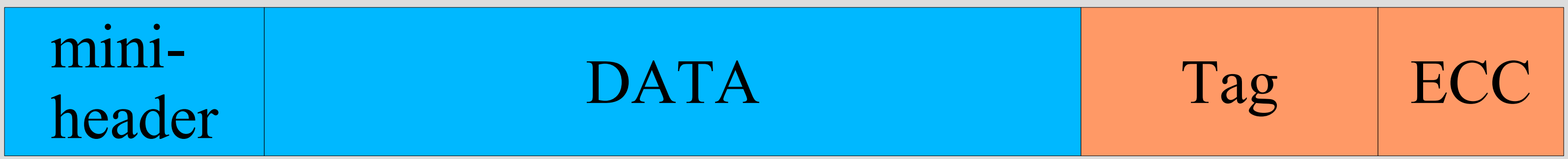


UFFS block recover(4)

- Recover a full-loaded block



UFFS Page data layout



- Mini-header (4B) and DATA on page data area, Tag and ECC on page spare area
- In some case (for example, when using hardware RS-ECC on small page MLC), you can treat the whole NAND page as one body, blur the page data and spare area boundary.

UFFS bad block management

- Bad block discover when mounting UFFS
- Bad block discover when read/write/erase
  - Try ECC error correct
  - If ECC fail, there is no way get valid data
  - Do not process bad block immediately, leave it at the end of Read/Write operation.
  - Only handle one bad block during the one read/write operation.
- Check bad block when formating UFFS

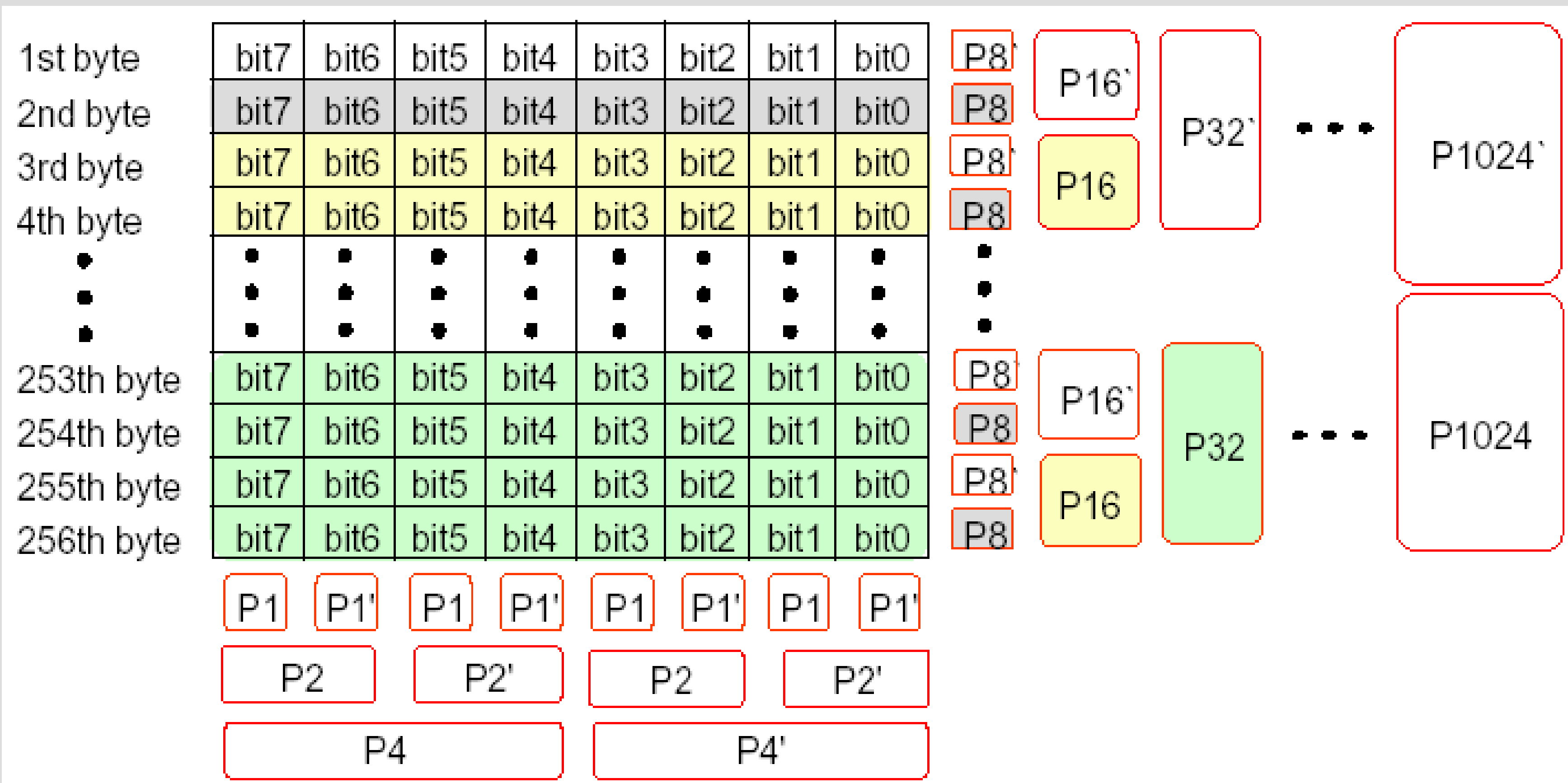
How ECC works ? (1)

- XOR: A ^ B = C
  - 0 ^ 0 = 0
  - 1 ^ 0 = 1
  - 0 ^ 1 = 1
  - 1 ^ 1 = 0
- Knowing any two of A, B and C, will know the rest one.
- UFFS ECC: 3 bytes ECC for 256 bytes data
  - 256 Bytes ==> 2048 Bits ===> 256(row) X 8(col)

How ECC works ? (2)

I/O 7	I/O 6	I/O 5	I/O 4	I/O 3	I/O 2	I/O 1	I/O 0
P64	P64`	**P32**	P32`	**P16**	P16`	**P8**	P8`
P1024	P1024`	**P512**	P512`	**P256**	P256`	**P128**	P128`
P4	P4`	**P2**	P2`	**P1**	P1`	1	1

P8 ~ P1024 : Line parity
P1 ~ P4 : Column parity



UFFS Flash Interface

- struct uffs_FlashOpsSt:
  - Use hardware ECC, or leave it to UFFS
  - Allow driver do the spare layout, or leave it to UFFS
  - Return flash operation status
  - Sequential page programing. No partial page programing.

UFFS Limitations

- Only one file/dir on one block
- Dynamic wear-leveling, Static wear-leveling is not implemented.

The next: UFFS2 ?

- Smaller Tree Node (12 bytes), save 25% RAM
- Use NAND block as buffers
- Multiple files/dirs on one block
- Support 8K, 16K page size
- Static wear-leveling
- Symbol link, FIFO file ?
- NOR flash support ? Maybe ...

The End