Flash Memory SIG Discussion

Dongjun Shin

Samsung Electronics

Contents

- Samsung's experience with flash memory file system
- Technical Issues related to flash memory
  - Handling various type of flash devices in a consistent way
  - Bad block management
  - ECC scheme
  - Flash translation layer
  - Wear leveling
  - Garbage collection
  - Boot loader
  - And so on
- Conclusion
- Discussion

CE Linux Forum

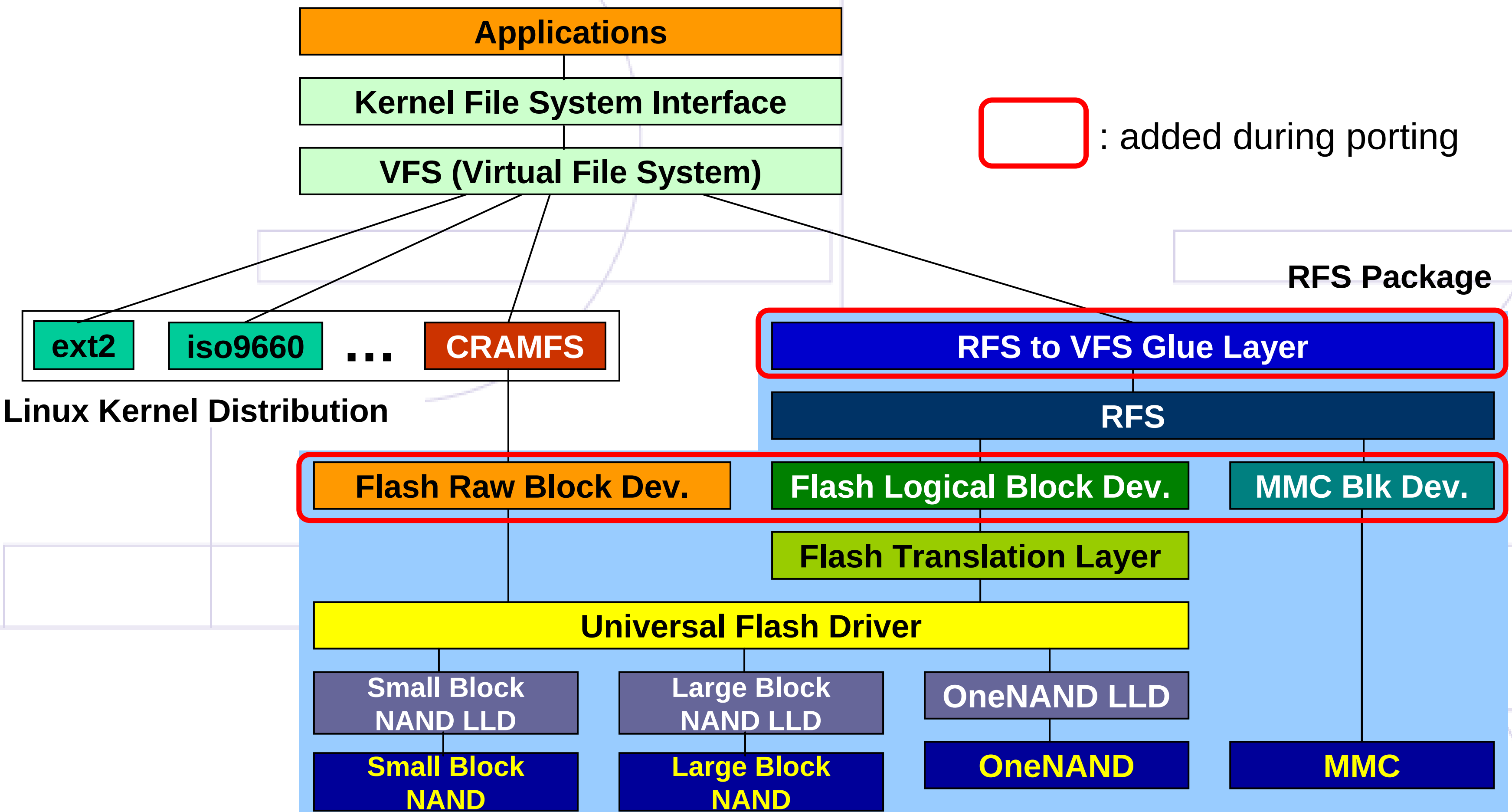
Development of Linux RFS

- Development history
  - No-OS version of RFS was developed, and it was ported to Linux afterwards (named “Linux RFS”) due to the customer’s request
- Requirement of RFS (draft)
  - FAT compatible
  - Robust (against system failure)
  - Optimized for NAND
    - Exception handling for bad-block and ECC
    - Portable across different NAND chip & target platform
    - Should have performance as close as the spec of datasheet
    - Bootable

Issues during Migration

- Design-level decision
  - What about JFFS2 or YAFFS?
  - Having its own S/W stack or use MTD?
  - How does MTD support NAND in its design?
- Integration with Linux file system layer
  - Uncertainty factor: page cache, buffer cache, disk scheduler, faucet
  - Is the full S/W stack controllable? (to meet the performance and robustness requirement)
- Driver optimization & portability (when adopting to MTD)
  - How do we support large-page/multi-planed NAND and OneNAND?
  - How do we support new features of NAND? (ex. Cache read, cache write, copy-back)
  - How to handle bad block using replacement scheme?

Linux RFS Architecture



Issues

- Handling various type of flash devices in a consistent way
- Bad block management
- ECC scheme
- Flash translation layer
- Wear leveling
- Garbage collection
- Partition table
- Boot loader
- Exploiting New flash memory technologies
  - New operations: multi-plane, cache program, cache read, copy back
  - New type devices: OneNAND, MLC
- Other considerations

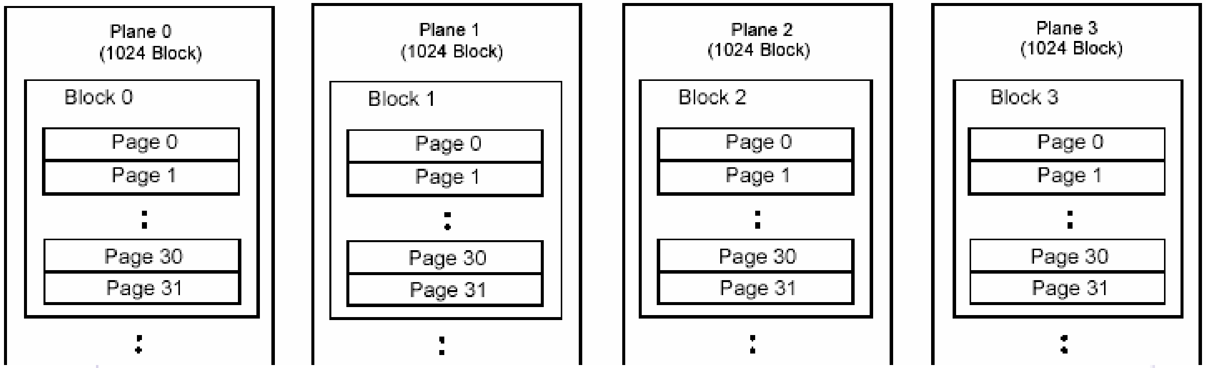
Handling various flash memories in a consistent way

- Supporting different type of flash memory, I/F, and configurations
  - NAND: single/multi plane, small/large page, SLC/MLC, NAND with NOR I/F (OneNAND), mux/demux
  - NOR: SLC/MLC
  - Expanding memory capacity by cascading multiple devices
- Accessing flash memory devices

mtdblock	Sector-oriented (partition, sector)
JFFS2	Byte-oriented (direct access to flash memory operations per partition)
Linux RFS	Page-oriented (dev, block, page-group)

Handling various flash memories in a consistent way (Cont'd)

- Multi-plane structure



- How about MLC?
- Why consistency matters?
  - To make the porting work easy
  - To make the best use of performance provided by underlying H/W

Bad Block Management

- "Bad Block" definition
  - On NAND flash memory, some bad blocks may exist at initial purchase time or at runtime (< 2% of entire blocks)
- Bad block management (BBM) scheme
  - Replace bad block with good one (*preferred!*)
    - Bad block management layer handles bad block both at initialization and at runtime with spare blocks
    - Upper layer doesn't care about the bad block
    - Lots of commercial S/W stack for flash memory use this approach
  - Just skip bad block
    - Example: JFFS2, YAFFS on MTD
    - File system should consider the existence of bad block

Bad Block Management (Cont'd)

- Issues with replacement scheme
  - Need some reserved area for replacement (like spare tire!)
  - Need map table to maintain replacement status
  - Compatibility issue: what if boot loader (or gang programmer) and file system uses different scheme?
  - Possible IP infringement

- Common questions related with BBM
  - Which is the best place to handle bad block?
  - Bad block management unit: partition or chip?
  - How to use file system X on NAND? (including cramfs/romfs)
  - How to boot from NAND?
  - How to gang-program NAND?

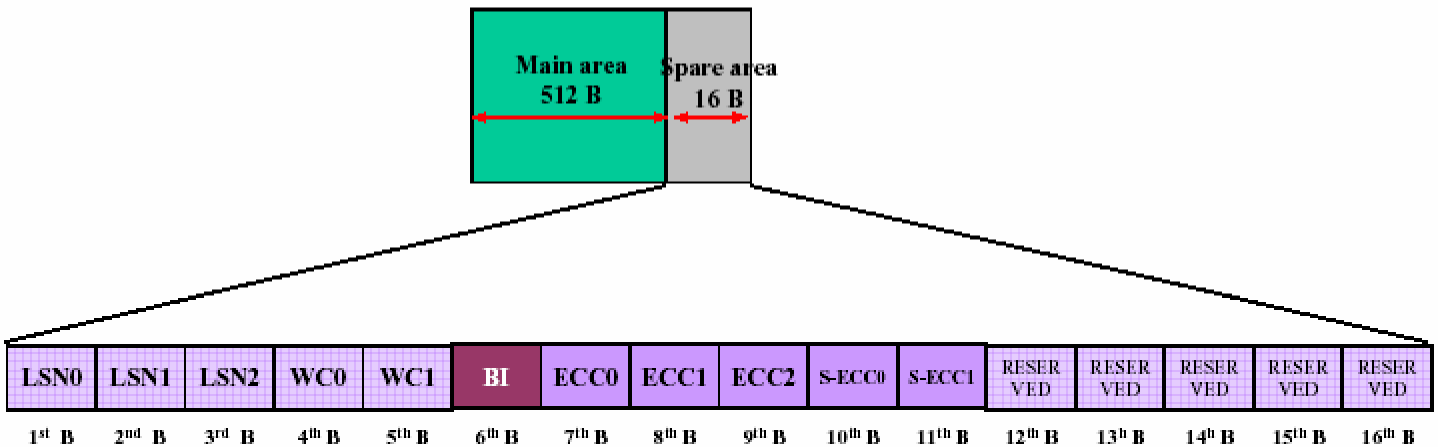
ECC Scheme

- Background
  - On NAND, 1bit error is considered as normal and need to be corrected with ECC
  - ECC code is stored on spare area of NAND
- How ECC is handled?
  - S/W method: generate ECC code by computation
  - H/W method: CPU or NAND chip has a special H/W logic to auto-generate ECC
  - Linux MTD support both methods
- Problem of ECC
  - A flash file system may not work on some H/W platform that support H/W ECC due to different ECC layout on spare area (Ex. YAFFS on OneNAND)*
  - Compatibility issue: which spared bytes are used for ECC bytes?

*: recent version of MTD support ECC layout customization

ECC Scheme (Cont'd)

Spare area assignment standard by Samsung Electronics



> LSN : Logical Sector Number
> WC : Status flag against sudden power failure during write
> ECC0,ECC1,ECC2 : ECC code for Main area data
> S_ECC0,S_ECC1 : ECC code for LSN data
> BI : Bad block Information

(http://www.samsung.com/Products/Semiconductor/Flash/TechnicalInfo/spare_assignment_standard_20030221.pdf)

Wear-leveling

- Background
  - Flash memories have upper limit on the number of erase count for each block
    - 100K for NAND
  - Traditional file systems tend to concentrate updates to specific region (ex. Metadata)
  - Without wear-leveling, flash memory may wear-out in shorter time than expected
- Wear-leveling method
  - Perfect method: keep erase count for each block
  - Heuristic method: reuse blocks in a round-robin way (erase count follows normal distribution)
    - Ex. JFFS2

Flash Translation Layer (FTL)

- Role of FTL
  - Translate sector read/write into flash read/program/erase operations
  - Common to NOR/NAND
- Linux MTD already has this feature
  - mtdblock, mtdchar
  - FTL, NFTL (by M-Sys)*: seems to handle wear-leveling and bad-block replacement as well

*: usage is limited to some devices due to patent

Garbage Collection

- Flash memory does not permit in-place editing
  - A block should be erased before programming
  - In some LFS-like implementations, new blocks are used during update as well as creation of pages
    - Ex. File update, file system metadata update
- Garbage collections of several partially used blocks are required to make clean block => performance issues
- In Linux, JFFS2 and YAFFS do this operation

CE Linux Forum

Boot Loader

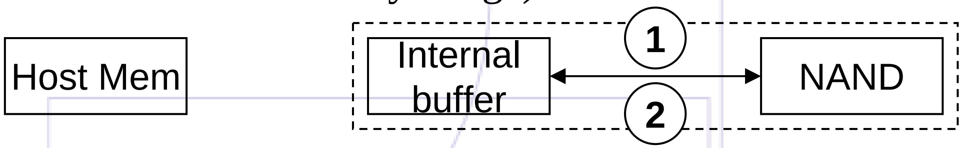
- Some CPU provides facility to boot from NAND using NFC
  - How do we enable it? (n-stage booting)
    - Ex. IPL => u-boot => kernel
  - How to access NAND flash? (NAND driver)
  - How to load/update kernel image? (BBM-related)

- Implementation issue
  - Duplication of BBM and driver sources in boot loader and file system

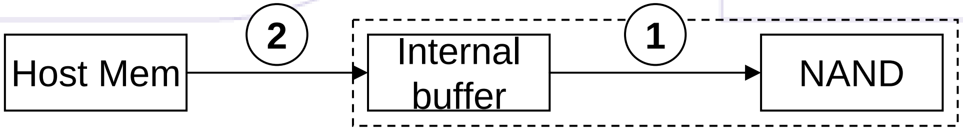
- If you're interested in booting from NAND, please ask me a demo! ☺

Exploiting New Technologies

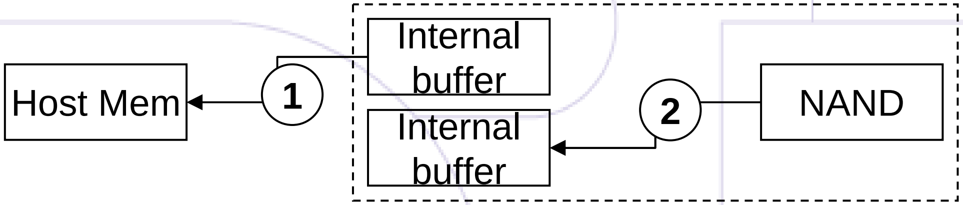
- Copy-back
  - Using internal buffer to load and store a page on NAND cell (bypassing the ‘transfer to host memory’ stage)



- Write while program (or cache program)
  - Overlapping the program & write operation



- Read while load
  - Using dual-buffer for overlapping load & read operation*



* specific to OneNAND

Other Considerations

- Embedded vs. card-type
  - Embedded: directly connected to CPU via memory bus
  - Card-type: via dedicated bus or other bus (ex. USB)
  - Common confusions
    - Block device or USB mass storage? => depends on the type of connectivity
    - What kind of S/W should be stacked? => depends on the existence of controller inside the memory (card) and on the type of file system
- Partition Table
  - In the source code
  - On the flash memory (runtime modifiable)

Conclusion

- Linux MTD is a versatile framework for memory-type devices
  - Lots of support for NOR have already been made and NAND support is catching up
- However, there are still some issues that need to be addressed and improved in MTD
  - Bad block management by replacement
  - Assumption about I/O unit size, ECC usage
  - Optimization for new technologies
- It’s not just a matter of implementation, but also of standardization

Discussion

- Should there be a FM WG?
  - Anyway, the flash memory vendor will provide necessary S/W stack
- Scope of FM WG
  - MTD only?
  - Incorporate all flash memory related issues?