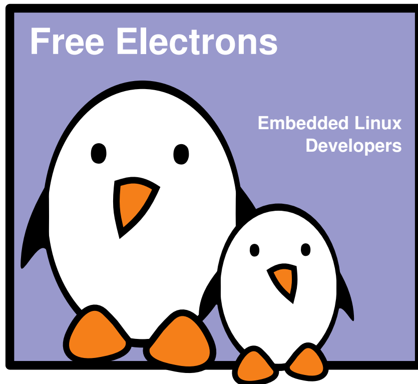
Modernizing the NAND framework: The big picture

Boris Brezillon
Free Electrons
boris@free-electrons.com



© Copyright 2004-2016, Free Electrons.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Boris Brezillon

- Embedded Linux engineer and trainer at Free Electrons
  - Embedded Linux and Android **development**: kernel and driver development, system integration, boot time and power consumption optimization, consulting, etc.
  - Embedded Linux, Linux driver development, Android system and Yocto/OpenEmbedded **training courses**, with materials freely available under a Creative Commons license.
  - `http://free-electrons.com`
- Contributions
  - **Kernel support for the AT91 SoCs** ARM SoCs from Atmel
  - **Kernel support for the sunXi SoCs** ARM SoCs from Allwinner
  - **Regular contributor to the MTD subsystem**
  - **Recently promoted maintainer of the NAND subsystem**
- Living in **Toulouse**, south west of France

What is this talk about?

- Detailing the current state of the NAND framework, its inconsistencies and limitations
- Proposing some approaches to rework it
- Sharing ideas and getting feedback
- Interaction during the talk highly encouraged

Introduction to the NAND framework

NAND framework position in the flash stack

Layer			
File System Layer	...	JFFS2	UBIFS
Wear Leveling Layer			UBI
Flash Memory Abstraction Layer	MTD		
Flash Type Abstraction Layer	...	...	NAND
Flash Drivers	... Drivers	Drivers	NAND controller drivers

NAND subsystem architecture

- Designed to interface with 'raw' NANDs
- (Thin ?) layer between MTD subsystem and NAND controller drivers
- All the abstraction is handled through `struct nand_chip`

Summarizing NAND framework limitations

An ancient framework

- Introduced in linux 2.4.6
- Was meant to support first NAND chips and simple NAND controllers
- Has evolved from time to time to support features exposed by newer NAND chips
- Disruptive growth
- Lot of code duplicated in NAND controller drivers
- Too open to enforce consistency across NAND controller drivers on some aspects...
- ... and too restrictive on other aspects to be really performant
- Now lacks many functionalities to make raw NAND chips really attractive compared to eMMC

Too open to be easily maintainable

- Mainly comes from the way the framework has grown
  - Creation of a new hook in `struct nand_chip` for each new feature
  - Allows for high customization in each NAND controller driver...
  - ... but also becomes a burden to maintain
  - ... and prevents any transversal refactoring of the framework
- Some core methods (which should be controller agnostic) can be overloaded by NAND controller drivers
- Results in hardly maintainable code base, where some NAND controller drivers are only compatible with a limited number of NAND chips.
- This liberty at the controller driver level also penalizes MTD users: inconsistent behavior from one controller to another.

Too restrictive to allow controller side optimizations

- Most modern NAND controllers are able to pipeline NAND commands and data transfers...
- ... but the framework splits commands and data transfer requests
- Some controllers are even capable of queuing several NAND commands thus allowing for maximum NAND throughput with minimum CPU usage...
- ... but the framework is serializing the page read/write requests

Does not take advantage of advanced NAND features

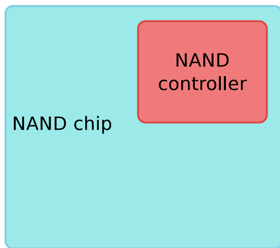
- Modern NANDs provide different solutions to optimize read/write accesses
  - Cached accesses
  - Multi-plane accesses
  - Multi-die accesses
  - ... and probably other funky stuff invented by NAND vendors :)
- An attempt was made to support cached program operations when doing sequential write accesses...
- ... the code is still there but the feature is always disabled because it was not providing a significant improvement at the time it was added
- Tested it recently on a DDR NAND, and the results are really encouraging (20% improvement)
- Combined with NAND controller optimizations we could even get better results

Clarifying NAND framework concepts/interfaces

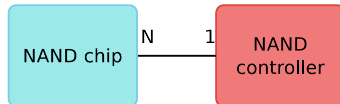
Needs to be redesigned to clarify the different concepts

- NAND framework is not providing a clear separation of the NAND controller and NAND chip concepts
- This leads to bad design in a lot NAND controller drivers
- Reworking the API to expose NAND controllers and automate NAND chip registration attached to those controllers should clarify the separation
- Some `struct nand_chip` hooks should be moved to the new `struct nand_controller` (not necessarily in their current form)
- However, we should not rush on moving methods from `struct nand_chip` to `struct nand_controller`, otherwise we will end up with the same hardly maintainable mess but in a different place

NAND components association



How it's usually done



How it should be done

Fuzzy design leads to erroneous driver implementation

- NAND chip interface is not clear
- Some methods are abused (either intentionally or unintentionally)
- Not clear when the methods should be implemented/overloaded
  - Some are meant to be implemented by simple controllers
  - Some are meant to be implemented by advanced controllers, thus leaving more control to the NAND controller, but when they are not implemented the core provides generic wrappers around the simple controller methods
  - Some should be implemented by NAND vendor specific code, but are overloaded by NAND controller driver
- Everything is mixed up in `struct nand_chip`
- It's easy to get it wrong

Proof by example: `->cmdfunc()` misunderstanding

- One key hook in `struct nand_chip` is `->cmdfunc()`
- Used to tell the NAND chip to launch a NAND operation
- Let's first detail the NAND protocol to understand what `->cmdfunc()` is supposed to do

NAND protocol (1)

CMD : command cycle

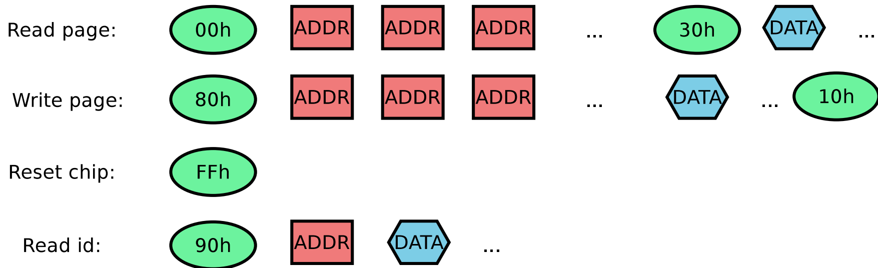
ADDR : address cycle

DATA : data cycle

CMD ADDR ADDR ADDR ... CMD DATA ...

NAND operation

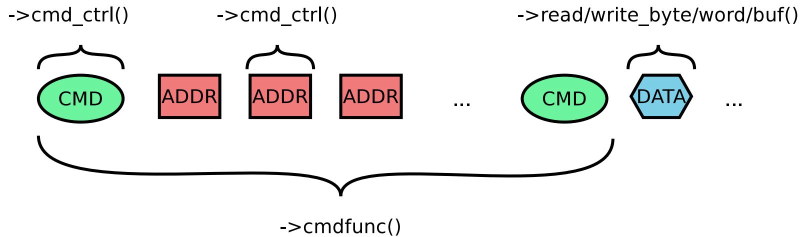
NAND protocol (2)



NAND protocol implementation

- `->cmdfunc()` is only partially handling NAND operations
- Only takes care of sending `CMD` and `ADDR` cycles
- Data are transfered with `->read/write_buf/byte/word()`
- The core provides a default implementation of `->cmdfunc()`
- Default implementation relies on `->cmd_ctrl()` to actually send the `CMD` and `ADDR` cycles on the bus

NAND protocol implementation



NAND protocol implementation limitation

- This implementation worked well for simple controllers implementing `->cmd_ctrl()`
- But some controllers are now able to handle full NAND operations (including the data transfer)
- Hence the ability for NAND controller drivers to overload `->cmdfunc()`
- Introduces a few problems:
  - `->cmdfunc()` does not provide enough information to predict how many bytes the core will actually transfer
  - Not all controllers are able to dissociate data transfers from the `CMD` and `ADDR` cycles which forces some drivers to guess the number of bytes the core will ask directly in `->cmdfunc()`
  - Implies non-uniform support across NAND controllers (some controllers will only support a subset of commands, thus preventing the use of advanced features)
  - Encourages people to support only a minimal set of operations

Addressing this limitation

- `struct nand_controller` should have a method to send full NAND operations (`->exec_op()` ?)
- We can still provide helpers to make this method rely on mechanisms similar to `->cmd_ctrl()` and `->read/write_byte/buf/word()`
- NAND controller drivers should not do any guessing depending on the operation type
- Should improve a bit the performance on controllers supporting a single interrupt event for the whole operation
- Will allow NAND core to implement different NAND operations in a generic manner
- Only limitation: some NAND controllers implement dedicated logic for some commands (READID, RESET, ...)
- They should not use these dedicated logics (unless they have no other choices, which is unlikely the case)

Other problems

Smartness is not necessarily good

- NAND core code tries to fill the gaps between what is supposed to be implemented and what the NAND controller driver really implements
- Set its own default implementations to fill the missing `struct nand_chip` hooks...
- ... but it's not necessarily a good idea
  - Sometimes the implementation is missing because the developer did not care implementing it
  - Blindly assigning a default implementation in this case can be worst than reporting `-ENOTSUPP`
- Solution:
  - Export default implementations, and let NAND controller drivers fill `struct nand_chip` appropriately
  - Fail and complain when mandatory methods are missing
  - Implement a dummy method returning `-ENOTSUPP` when the hook is optional

Adapting the NAND subsystem to developer needs

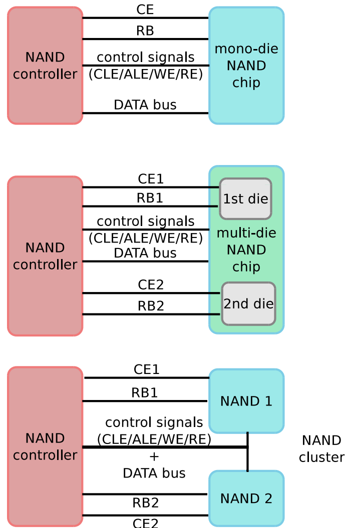
- Nothing new here, but it is worth mentioning that this applies to the NAND subsystem too
- Already happening for some aspects
  - Helpers to fix bitflips in erased pages (added in 4.4)
  - DT parsing automation (added in 4.2)
- Need to be pushed even further
- The introduction of the NAND controller concept should help removing more boilerplate code
- As showed earlier with the `->cmdfunc()` example, the `nand_controller` interface has to take modern NAND controller capabilities and constraints into account

Isolating the NAND layer from the MTD layer

- NAND and MTD concepts are currently mixed all over the NAND subsystem
- NAND controller drivers have to deal with the MTD struct
- Provide a better separation to hide MTD
  - Ease maintainance (future NAND refactoring should not impact the MTD layer and the other way around)
  - Control driver accesses (allowing drivers to mess with `mtd_info` is not a good idea)
  - Clarify the code (only one object passed to NAND controller methods instead of 2)
- Rework has already started:
  - Now `struct nand_chip` directly embeds a `struct mtd_info` object
  - New NAND controller methods should only be passed a `nand_chip` object

Improving NAND performance

Raw NAND flash bus



Controller side optimization

- Modern NAND controllers support advanced features like:
  - Pipelining NAND operation
  - Advanced CS control, allowing interleaved operations (do not have to wait for an operation on a NAND die to finish before launching an operation on a different die)
  - And probably other hacks to improve NAND traffic
- The problem is, the NAND framework design does not allow the controller to do this kind of optimization.
  - NAND chip accesses are serialized at the core level (prevents pipelining NAND operations)
  - Accesses to a given NAND controller are also serialized, thus preventing multi-die or multi-chip accesses

Controller side optimization: ->read() example

Controller side optimization: `->read()` example

MTD layer

NAND layer

NAND controller driver

->read()

->queue_nio_req()

->wait_nio_req_completion()

```
What's done here should be
controller dependent,
but we can provide a default
implementation relying on
->queue_op()/->wait_op_completion().

The idea is to let the NAND
controller prepare the different
NAND operations needed for this
->read() request, and benefit from
advanced queueing features (usually
relying on DMA accesses).
```

Chip side optimizations

- We have different ways to optimize NAND performance
- But they pretty much all depend on NAND controller implementation to be relevant
- Some examples:
  - Cached accesses
  - Multi-plane accesses
  - Multi-die accesses
  - DDR or higher SDR mode support
- Not all chips support those features
  - We need a way to expose which features the chip supports
  - ONFI and JEDEC already have a standard solution to expose that (should be extended for non-ONFI/non-JEDEC chips)
  - Optimizations features are exposed by the mean of new NAND operations (or extension of existing NAND operations)
  - NAND specific hooks should be added to let the chip build a NAND operation description
  - In the end, the decision to use those optimizations should be left to the controller

Chip side optimizations: cached accesses

- Cached accesses: the chip has 2 levels of cache, one is exposed to the user, and the other one is used to retrieve or push data to/from another NAND page
- Cached accesses are relevant when the time spent in ECC calculation and I/O transfer operations is not negligible compared to the READPAGE, PROGPAGE time.
- Particularly true for modern NAND chips with huge pages (greater than 4k)

Optimizing even more: NAND IO scheduling

- NAND I/Os can be dispatched on different plane/dies
- Most controllers support accessing several dies in parallel
  - While one operation is taking place on a specific die the controller can interact with other dies
  - The only contention is on the NAND bus: only a single die can be addressed at a time, but once the command is sent and the chip is busy, the controller can use the NAND bus to interact with another die
- Getting an efficient usage of the NAND bus requires scheduling I/O requests
  - Queue NAND requests per-die, and not per-chip
  - If we want multi-plane support, we should even have a per-plane queue
  - The scheduling algorithm should be generic
  - The NAND controller should only dequeue requests from the different queues
- Still just an idea, not sure about the implementation details yet

Supporting modern/non-ONFI/non-JEDEC chips

Supporting modern/non-ONFI/non-JEDEC NANDs

- MLC/TLC NAND chips need some specific features that are not accessible through standard commands (read-retry, SLC mode, ...)
- We need vendor specific code
- Doing that should also simplify the 'not so generic' chip detection code in `drivers/mtd/nand/nand_base.c`
- Better isolate vendor specific handling in vendor specific `.c` files
- Some `struct nand_chip` hooks should be filled in there
- Vendor specific tweaking is also required for ONFI or JEDEC compliant chips
- This approach should limit the number of full-id NAND entries in `nand_ids` table (and we may even be able to completely get rid of it)

Sharing code between different NAND based devices

Sharing code between different NAND based devices

- Linux kernel currently supports different NAND based devices using different interfaces
  - Raw NANDs (those we are talking about in this talk)
  - OneNAND NANDs
  - SPI NANDs
- Even if they don't use the same physical and logical interfaces, they may share enough to allow some factorization
  - Memory array organisation
  - NAND type (SLC/MLC/TLC), and associated constraints
  - Bad block table handling
  - ECC handling?
  - ...
- Isolating those similarities and factorizing common code sounds like a good idea

Sharing code between different NAND based devices

- Work already started by Brian Norris and Peter Pan (making bad block table code reusable for OneNAND and SPI NAND)
- We should go further:
  - Rename `struct nand_chip` into `struct rawnand_chip`
  - Make `struct rawnand_chip`, `struct onenand_chip` and `struct spinand_chip` inherit from `struct nand_chip`
  - Move all the code under `drivers/mtd/nand` and create one subdirectory per interface type
  - See what else could be factorized

A few words about MLC support

A few words about MLC support

- On the NAND subsystem side, we've made good progress
  - NAND chips can be flagged as needing SCRAMBLING
  - Hardware Randomizer support has been added to the sunxi NAND driver
  - Still need to expose NAND page pairing info to MTD users, but we have a solution and will post it soon
  - The main remaining issue is read-retry support, which is chip dependent, and thus relies on the per-vendor implementation stuff
- On the UBI/UBIFS side, we've made good progress too
  - A solution to address the data retention problem has been posted a few months ago (Richard Weinberger)
  - We are working with Richard to address the paired pages corruption problem
  - We evaluated several solutions on different NAND chips
  - Heading to a LEB consolidation approach at the UBI level
  - RFC should be posted soon: stay tuned ;-)
- Most of this work was funded by NextThing Co.

Questions? Suggestions? Comments?

Boris Brezillon

boris.brezillon@free-electrons.com

Slides under CC-BY-SA 3.0

http://free-electrons.com/pub/conferences/2016/elc/brezillon-nand-framework/