Flash and NAND Subsystem for NetBSD

Adam Hoka

February 13, 2011

1 Abstract

Mobile devices are getting more widespread as the technology gets cheaper and cheaper. If NetBSD wants to stay in line with this trend, it needs to support the most common storage technology in these devices, and that is currently flash.

Existing drivers and file systems cannot be used with Flash memory. Thus, if NetBSD wants to compete with Linux, WinCE and other operating systems used in embedded devices, specific support must be implemented.

An effort of developing a Flash specific file system for NetBSD has been started at the University of Szeged, Hungary in 2010. This project urged the design and implementation of a Flash layer for NetBSD, which could be used as a foundation for the file system.

2 Introduction to Flash

2.1 Overview

Embedded systems almost solely use some kind of Flash chip for non-volatile data storage. Flash memory is a type of EEPROM, which stands for Electronically Erasable Programmable ROM. Unlike magnetic or optical disks, Flash uses semi-conductor technology for holding information. The chip is organized as an array of cells, constructed from either NAND or NOR logical gates. These gates are implemented with transistor flip-flop circuits. NOR and NAND Flash memory have subtle differences in behavior, such as addressability, performance, and manufacturing cost. NOR ROMs have separate address and data lines on the chip package, which enables RAM like addressing and usage. This makes NOR Flash suitable for utilizing as a boot device for small systems, as it needs little support from the boot code. Booting from the NAND Flash would require more logic in the boot code to read from the chip and shadow the boot image in the operative memory for execution. It is possible to execute program code directly from a NOR Flash device, which is not only useful at boot time, but also can help memory constrained systems by accessing the Flash directly instead of using space in the RAM.

2.2 NAND memory

NAND memory needs special handling, because of the lack of separate address and data lines. Most chips have three registers for communication and data transfer: address register, command register and data register. These chips also have a CE (Chip enable) line for reusing a single data bus for multiple devices, but that is out of our scope. The registers are usually used in the following way:

1. write a command to the command register
2. write and address to the address register (if the command requires one)
3. read or write the data register for reading metadata and accessing stored data

NAND Flash cells are grouped into pages, which have typical sizes of 512 (small page NAND), 2048 (large page NAND) or 4096 (huge page NAND) bytes. A certain number of pages form a so-called “block”. Block size is typically 64 pages, which is 16kB on a large page NAND device. Every page has a few additional reserved bytes, called the OOB (out of band) area. The size of this reserved area is also called as the “spare size” of the chip. The OOB can hold meta-data, such as bad block information and ECC code for the page.

Addressing the chip is based on column and row addresses. The row address selects which page to access, and the column address can select the offset into that page. The address is entered through the address register one byte at a time.

NAND Flash has several issues we have to deal with. Writing requires a block erase first, so writing is much slower than reading. Sequential writes can be serialized to a single block erase and program operation, but random reads will be very slow. NAND Flash is not 100% reliable, so the data needs to be protected with ECC error correction code. That is usually some version of the Hamming code algorithm. Accidentally flipping bits can be corrected on read with this error correction code stored in the OOB area. The major drawback of the NAND Flash is the aging of the semi-conductor cells. A block can go bad over time if it is erased or written more times than its guaranteed cycles.

3 Existing implementations

3.1 Linux MTD

The Linux MTD subsystem is widely used and feature rich, but the code itself is very chaotic and hard to understand. NAND devices are not accessed through standard ONFI commands, but through different legacy interfaces. Linux MTD supports a tool called mtdblock for raw device access, but it’s usage is not recommended by the official documentation. Available in stock Linux kernels.

3.2 Andrew Turner’s NAND Driver for FreeBSD

Very limited implementation supporting NAND only. It has only partial ECC implementation and doesn’t support 16bit devices. The code is easy to understand and well designed. This implementation heavily affected the NetBSD NAND subsystem’s design. Available from a GitHub.com repository.

3.3 FreeBSD NFC

NAND only implementation using geom. Feature rich and comes with a complex emulator. Supports bot legacy and standard ONFI access. Available from FreeBSD’s Perforce repository.

3.4 Conclusion

The NetBSD NAND subsystem provides a similar abstraction like the Linux MTD, but the design itself is much more modern, similar to FreeBSD NFC. Advantage over Linux MTD is simpler code and ONFI standard implementation.

4 The Flash subsystem

4.1 Overview

The Flash subsystem abstracts different Flash devices with a common interface. NAND and NOR memory needs different drivers, but because the fundamental concepts are the same, we can abstract implementation differences. This level of indirection makes easier to write device agnostic file systems and to use a common block device driver.

4.2 Block device

Each Flash device instance can be accessed from userland through it's device node, for example `/dev/flash0`. The supported operations are `open`, `read`, `write` and `ioctl`. The supported ioctls are the following:

- `FLASH_ERASE_BLOCK`
- `FLASH_DUMP`
- `FLASH_GET_INFO`
- `FLASH_BLOCK_ISBAD`
- `FLASH_BLOCK_MARKBAD`

4.3 Flash API

The `flash(4)` driver requires the implementation of the following functions:

- `erase`
- `read`
- `write`
- `block_markbad`
- `block_isbad`
- `sync`
- `submit`

The `erase` function erases a contiguous area in the Flash memory.

The `read` function fills a pre-allocated buffer with data from the Flash memory.

The `write` function programs data from the given buffer to the Flash memory. Programming requires an erase operation beforehand.

The `block_markbad` function marks a block (erase unit) of Flash memory as bad, which means it will never be used again for storing data.

The `block_isbad` function checks a given block (erase unit) of the Flash memory if it is marked as bad, and thus not usable.

The `sync` function requests to write any pending data to the Flash memory, if the flash driver uses some kind of caching.

The `submit` function submits an I/O request to the flash driver for handling. This can be either a read or a write operation coming from the block device interface.

These functions are registered in the `flash_interface` structure as function pointers with additional data, like size of Flash memory, page size and other parameters.

The function pointers should not be used directly, as addressing may need translations with respect to partition handling. There are access functions for each of the API functions prefixed with flash to avoid namespace pollution:

- flash_erase
- flash_read
- flash_write
- flash_block_markbad
- flash_block_isbad
- flash_sync
- flash_submit

5 The NAND subsystem

5.1 Overview

The NAND subsystem implements the flash(4) API, and tries to handle all NAND devices conforming to the ONFI 2.3 specification. Support for devices not conforming to ONFI is possible, but have not been done yet.

5.2 NAND I/O Thread

Because of NAND devices require a block erase operation before programing pages, we need to ensure ordering and caching of write operations. This is achieved through the NAND I/O thread, which runs as an independent kernel process. Any I/O request coming from the flash(4) layer will be immediately executed, but writes will not be flushed to the Flash until we have a full block written or a specific timeout has been exceeded. On an addition, any read request will cause the cache to be written to the Flash. The timeout is implemented with two timestamps for the cache: one for cache creation, and another for last write to the cache. If any of these timestamps get older than the given values (currently 3 seconds and 1 millisecond respectively), the cache will be committed.

5.3 NAND API

A driver for a NAND controller using the NAND API must implement the following functions:

- select
- command
- address
- read_byte
- read_word
- read_buf_byte
- read_buf_word
- write_byte
- write_word
- write_buf_byte
- write_buf_word
- busy

The select function may be implemented for device that need some kind of CS (chip select) mechanism. Otherwise this can be a no-op.

The command function send an 8bit command to the NAND chip's command register. These commands are listed in the onfi.h header file.

The address command sends an 8 bit address to the NAND chip's address register. We need to send multiple bytes of address in most cases. Two types of addressing is possible: column and row addressing, the former addresses a subpage byte. Each command need different addressing.

The read_byte function reads one byte from the NAND chip's data register.

The read_word function reads one word (16 bit) from the NAND chip's data register.

The read_buf_byte function fills a given buffer with byte reads from the NAND chip's data register.

The read_buf_word function fills a given buffer with word (16 bit) reads from the NAND chip's data register.

The write_byte function writes one byte to the NAND chip's data register.

The write_word function writes one word (16 bit) to the NAND chip's data register.

The write_buf_byte function sends a given buffer with word writes to the NAND chip's data register.

The write_buf_word function sends a given buffer with word (16 bit) writes to the NAND chip's data register.

The busy function delays program execution until the last issued command has finished.

The driver may implement the following functions or use the default implementations provided by the nand(4) subsystem:

- ecc_prepare
- ecc_compute
- ecc_correct

The ecc_prepare function prepares the ECC engine in case a hardware implementation is used. Hardware ECC engines usually calculate the error correction code as the data is written to the data register. For software-ECC, this is functions is a no-op.

The ecc_compute function does the actual computation if the software ECC method is selected. Typical hardware ECC engines already have the computed error checking code in a register, so the driver reads and optionally converts the ECC code in this step.

The ecc_correct function does the error correction part. As various hardware engines may use different ECC schemes, the driver needs to implement the appropriate algorithm for the error correction.

These functions are registered in the nand_interface structure.

6 Legal section

I have the obligation to display the following note: *The Project named „TÁMOP-4.2.1/B-09/1/KONV-2010-0005 – Creating the Center of Excellence at the University of Szeged" is supported by the European Union and co-financed by the European Regional Fund. – http://www.nfu.hu/*