NAND FLASH

http://www.linux-mtd.infradead.org/tech/nand.html

NAND vs. NOR

Beside the different silicon cell design, the most important difference between NAND and NOR Flash is the bus interface. NOR Flash is connected to a address / data bus direct like other memory devices as SRAM etc. NAND Flash uses a multiplexed I/O Interface with some additional control pins. NAND flash is a sequential access device appropriate for mass storage applications, while NOR flash is a random access device appropriate for code storage application. NOR Flash can be used for code storage and code execution. Code stored on NAND Flash can't be executed from there. It must be loaded into RAM memory and executed from there.

	NOR	**NAND**
Interface	Bus	I/O
Cell Size	Large	Small
Cell Cost	High	Low
Read Time	Fast	Slow
Program Time	Fast	Slow
Erase Time	Slow	Fast
Power consumption	High	Low, but requires additional RAM
Can execute code	Yes	No, but newer chips can execute a small loader out of the first page
Bit twiddling	nearly unrestricted	1-3 times, also known as "partial page program restriction"
Bad blocks at ship time	No	Allowed

As NAND Flash is cheaper than NOR Flash and has a very slim interface it was selected as the optimum solution for large nonvolatile storage applications such as solid state file storage, digital audio/voice recorder, digital still camera and portable applications requiring non-volatility.

NAND Types

There are various types of NAND Flash available. Bare NAND chips, SmartMediaCards, DiskOnChip.

SmartMediaCards are bare NAND chips covered by thin plastic. They are very common in digital cameras and MP3 players. The card itself contains nothing smart at all. It gets smart by software.

DiskOnChip is NAND Flash with additional glue logic as a drop in replacement for NOR Flash chips. The glue logic provides direct memory access to a small address window, which contains a boot loader stub, which loads the real boot code from the NAND device. The logic also contains control registers for the static NAND chip control lines and a hardware ECC generator.

NAND technical view

The memory is arranged as an array of pages. A page consists of 256 / 512 Byte data and a 8 / 16 Byte spare (out of band) area. The spare area is used to store ECC (error correction code), bad block information and file system dependent data. n pages build one block. The read / write access to data is on a per page basis. Erase is done on a per block basis. The commands to read / write / erase the chip are given by writing to the chip with the Cmnd Latch Enable pin high. Address is given by writing with the Address Latch Enable pin high.

There are only a few lines neccecary to access up to 256 MB of Flashmemory.

Pin(s)	Function
I/O 0-7	Data Inputs/Outputs
/CE	Chip Enable
CLE	Command Latch Enable
ALE	Address Latch Enable
/RE	Read Enable
/WE	Write Enable
/WP	Write Protect
/SE	Spare area Enable
R/B	Ready / Busy Output

As it is neccecary to use the spare area, the /SE (Spare area Enable) pin should be tied to GND. /CE, CLE and ALE should be GPIO pins or latched signals. It's possible to use address lines for ALE and CLE, but you have to take care about the timing restrictions of the chip !

/RE and /WE can be tied to the corresponding lines of the CPU. Make sure, that they are logicaly combined with the corresponding chip select. You can also use two different chip selects for /RE and /WE, but be aware of data hold time constraints of your NAND chip. Data hold time after rising edge of /WE is different to data hold time after the rising edge of chip select lines!

I/O 0-7 are connected to the databus D0-D7. The /WP pin can be used for write protection or connected to VCC to enable writes unconditionally. As NAND flash uses command driven programming and erasing, an accidential write or erase is not likely to happen. The Ready / Busy output is not neccesary for operation, but it can be tied to a GPIO or an interrupt line.

File systems supporting NAND

One major problem for using NAND Flash is that you cannot write as often as you want to a page. The consecutive writes to a page, before erasing it again, are restricted to 1-3 writes, depending on the manufacturers specifications. This applies simillarly to the spare area. This makes it neccesary for the filesystem to handle a writebuffer, which contains data that is less than a page.

At the moment there are only a few filesystems which support NAND

- JFFS2 for bare NAND Flash and SmartMediaCards
- NTFL for DiskOnChip devices
- TRUEFFS from M-Systems for DiskOnChip devices
- SmartMedia DOS-FAT as defined by the SSFDC Forum

JFFS2 and NTFL are Open Source, while TRUEFFS is a proprietary solution. SmartMedia DOS-Fat is a specification from SSFDC forum. It is somewhat open under a non disclosure agreement with Toshiba, who owns all rights on this specifications. NTFL is designed for the usage of DiskOnChip devices. JFFS2 supports raw NAND chips and SmartMediaCards at the moment. A JFFS2 support for DiskOnChip devices, based on the NAND code, is planned. There are some other Open Source projects for NAND filesystem support, but there's no other working solution than JFFS2 at the moment of this writing. One of them is YAFFS, which is available in a beta version.

There is currently no support for the wide spread SmartMedia DOS-FAT filesystem, mainly because it's not a

reliable filesystem for industrial usage. It's ok for multimedia applications. The hardware support layer is designed to support an implementation of SmartMedia DOS-FAT, but nobody has made an attempt to implement it really. There are a couple of SmartMedia Card adaptors for USB, PCMCIA, FireWire ... with Linux drivers available, which support the SmartMedia DOS-FAT.

JFFS2 includes bad block management, wear leveling, error correction and provides a reliable filesystem on top of NAND Flash.

JFFS2 Out of Band usage

Nand chips with 256 byte pagesize and 8 byte OOB size

Offset	Content	Comment
0x00	ECC byte 0	Error correction code byte 0
0x01	ECC byte 1	Error correction code byte 1
0x02	ECC byte 2	Error correction code byte 2
0x03	reserved	reserved
0x04	ECC valid marker	If one of the lower 4 bits in this byte is zero, it indicates that this page was written with ECC
0x05	Bad block marker	If any bit in this byte is zero, then this block is bad. This applies only to the first page in a block. In the remaining pages this byte is reserved
0x06	Clean marker byte 0	This byte indicates that a block was erased under JFFS2 control. If the page was succesfully erased this byte in the first page of a block is programmed to 0x85. In the remaining pages this byte is reserved
0x07	Clean marker byte 1	This byte indicates that a block was erased under JFFS2 control. If the page was succesfully erased this byte in the first page of a block is programmed to 0x19. In the remaining pages this byte is reserved

Nand chips with 512 byte pagesize and 16 byte OOB size

Offset	Content	Comment
0x00	ECC byte 0	Error correction code byte 0 of the lower 256 Byte data in this page
0x01	ECC byte 1	Error correction code byte 1 of the lower 256 Bytes of data in this page
0x02	ECC byte 2	Error correction code byte 2 of the lower 256 Bytes of data in this page
0x03	ECC byte 3	Error correction code byte 0 of the upper 256 Bytes of data in this page
0x04	ECC valid marker	If one of the lower 4 bits in this byte is zero, it indicates that the lower 256 byte data in this page were written with ECC. If one of the upper 4 bits in this byte is zero, it indicates that the upper 256 byte data in this page were written with ECC.
0x05	Bad block marker	If any bit in this byte is zero, then this block is bad. This applies only to the first page in a block. In the remaining pages this byte is reserved
0x06	ECC byte 4	Error correction code byte 1 of the upper 256 Bytes of data in this page
0x07	ECC byte 5	Error correction code byte 2 of the upper 256 Bytes of data in this page
0x08	Clean marker byte 0	This byte indicates that a block was erased under JFFS2 control. If the page was succesfully erased this byte in the first page of a block is programmed to 0x85. In the remaining pages this byte is reserved
0x08	Clean marker byte 1	This byte indicates that a block was erased under JFFS2 control. If the page was succesfully erased this byte in the first page of a block is programmed to 0x19. In the remaining pages this byte is reserved
0x08	Clean marker byte 2	This byte indicates that a block was erased under JFFS2 control. If the page was succesfully erased this byte in the first page of a block is programmed to 0x03. In the remaining pages this byte is reserved
0x08	Clean marker byte 3	This byte indicates that a block was erased under JFFS2 control. If the page was succesfully erased this byte in the first page of a block is programmed to 0x20. In the remaining pages this byte is reserved
0x08	Clean marker byte 4	This byte indicates that a block was erased under JFFS2 control. If the page was succesfully erased this byte in the first page of a block is programmed to 0x08. In the remaining pages this byte is reserved
0x08	Clean marker byte 5	This byte indicates that a block was erased under JFFS2 control. If the page was succesfully erased this byte in the first page of a block is programmed to 0x00. In the remaining pages this byte is reserved
0x08	Clean marker byte 6	This byte indicates that a block was erased under JFFS2 control. If the page was succesfully erased this byte in the first page of a block is programmed to 0x00. In the remaining pages this byte is reserved
0x08	Clean marker byte 7	This byte indicates that a block was erased under JFFS2 control. If the page was succesfully erased this byte in the first page of a block is programmed to 0x00. In the remaining pages this byte is reserved

HOW TO implement NAND support

Where can you get the code ?

The latest changes to JFFS2 and the underlying NAND code are not in the kernel code at the moment. The latest code is available from CVS and daily snapshots

There are four layers of software

1. JFFS2: filesystem driver
2. MTD: Memory Technology Devices driver
3. NAND: generic NAND driver
4. Hardware specific driver

the MTD driver just provides a mount point for JFFS2. The generic NAND driver providesall functions, which are neccecary to identify, read, write and erase NAND Flash. The hardware dependent functions are provided by the hardware driver. They provide mainly the hardware access information and functions for the genering NAND driver.

Hardware driver

To implement it on your hardware, you will have to write a new hardware driver. Copy one of the existing hardware drivers and modify it to fit your hardware. These basic functions have to be modified:

1. Hardware specific ccontrol function
2. Device ready function
3. Init function

To provide enough flexibilty for all kind of devices, there can be supplied additional functions optionally.

1. Hardware specific command function
2. Hardware specific wait function
3. Hardware ECC functions

Hardware specific control function

This function sets and resets the control pins CLE, ALE and CE of the NAND device depending on the function argument. The argument constants are defined in nand.h and explained in the example function below.

```
void yourboard_hwcontrol(int cmd)
{
switch(cmd){
case NAND_CTL_SETCLE: Hardware specific code to set CLE line to 1; break;
case NAND_CTL_CLRCLE: Hardware specific code to set CLE line to 0; break;
case NAND_CTL_SETALE: Hardware specific code to set ALE line to 1; break;
case NAND_CTL_CLRALE: Hardware specific code to set ALE line to 0; break;
case NAND_CTL_SETNCE: Hardware specific code to set CE line to 0; break;
case NAND_CTL_CLRNCE: Hardware specific code to set CE line to 1; break;
}
}
```

Device ready function

If your hardware interface has the ready busy pin of the NAND chip connected to a GPIO or other accesible I/O pin, this function is used to read back the state of the pin. The function has no arguments and should return 0, if the device is busy (R/B pin is low) and 1, if the device is ready (R/B pin is high). If your hardware interface does not give access to the ready busy pin, you can delete this function and set the function pointer this->dev_ready during init to NULL.

```
int yourboard_device_ready(void)
{
return The state of the ready/busy pin of the chip;
}
```

Init function

This function is called when the driver is initialized either on kernel boot or when you load your driver as a module. The function remaps the I/O area through which your NAND chip can be accessed and allocates all neccecary memory chunks for the device structure and data cache. The structure of the device has to be **zeroed out first** and then filled with the neccecary information about your device. See example code for the most important members of the device structure you have to set.

```
int __init yourboard_init (void)
{
SNIP
/* Allocate memory for MTD device structure and private data */
yourboard_mtd = kmalloc (sizeof(struct mtd_info) + sizeof (struct nand_chip),GFP_KERNEL);
SNIP
/* map physical adress */
yourboard_fio_base=(unsigned long)ioremap(yourboard_fio_pbase,SZ_1K);
SNIP
/* Set address of NAND IO lines */
this->IO_ADDR_R = yourboard_fio_base; The address to read from the chip
this->IO_ADDR_W = yourboard_fio_base; The address to write to the chip
this->hwcontrol = yourboard_hwcontrol;Your function for accessing the controllines
this->dev_ready = yourboard_device_ready; Your function for accessing the ready/busy line or NULL, if you don't have one
this->chip_delay = 20;The delay time (us), after writing a command to the chip, according to datasheet (tR)
SNIP
}
```

Hardware specific command function

Some new CPU's, like Samsungs ARM based S3SC2410 provide a hardware based NAND interface. For them and ugly designed hardware interfaces it may be neccecary to have a customized command write function. You can supply such a function by setting
this->cmdfunc = yourboard_cmdfunc;
This must be done in yourboard_init before calling nand_scan.

The function is defined as
void yourboard_cmdfunc (struct mtd_info *mtd, unsigned command, int column, int page_addr);

Hardware specific wait function

If you have connected the ready/busy pin to an interrupt line, you could supply a own interrupt driven waitfunction for erase and program wait for ready. In this case set

this->waitfunc = yourboard_waitfunction;
This must be done in yourboard_init before calling nand_scan.

The function is defined as
int yourboard_waitfunction (struct mtd_info *mtd, struct nand_chip *this, int state);
Please take care of the functionality, which is in standard nand driver wait function (nand_wait).

Hardware based ECC

If your hardware supplies a hardware based ECC generator, then fill out the following. If not, skip this topic.

this->eccmode = NAND_ECC_HW3_256;
or
this->eccmode = NAND_ECC_HW3_512;
or
this->eccmode = NAND_ECC_HW6_512;

NAND_ECC_HW3_256 is a hardware based ECC generator, which generates 3 byte ECC code for 256 byte of data. Such a generator can be found on DOC devices and on passive SmartMedia adaptors.
NAND_ECC_HW3_512 is a hardware based ECC generator, which generates 3 byte ECC code for 512 byte of data. Such a generator can be found e.g. on Samsungs ARM based S3SC2410 CPU.

NAND_ECC_HW6_512 is a hardware based ECC generator, which generates 6 byte ECC code for 512 byte of data.

You have also to provide following functions:

Enable hardware ECC generator

void yourboard_enable_hwecc (int mode);

Your function to enable (reset) the hardware ECC generator. This function is called from the generic nand driver before reading or writing data from/to the chip. The function is called with NAND_ECC_READ or NAND_ECC_WRITE as argument.
Set the function pointer in init to
this->enable_hwecc = yourboard_enable_hwecc;

Enable hardware ECC generator

void yourboard_readecc(const u_char *dat, u_char *ecc_code);

Your function to read back the ECC bytes from your hardware ECC generator. Fill the data into the ecc_code array, which is given as argument. Ignore the first argument of this function, which is neccecary for software ecc only. Set the function pointer in init to
this->calculate_ecc = yourboard_readecc;

Error detection and data correction function

int xxxx_correct_data(u_char *dat, u_char *read_ecc, u_char *calc_ecc);

A function which implements error detection and data correction corresponding to your hardware ECC algorithm. This function should be incorporated into nand_ecc.c with an appropriate function name, to make it public available for similar hardware drivers. Maybe the function you need is already there. If you implement a new one, please contact NAND driver maintainer to include it in the public source tree. Please consult current implementations in nand_ecc.c to provide a compatible solution. Set the function pointer in init to
this->correct_data = xxxx_correct_data;

All the function pointers must be set in yourboard_init before calling nand_scan.

Supported chips

Most NAND chips actually available from 2 to 128MB should be supported by the current code. If you have a chip, which is not supported, you can easily add it by extending the chiplist in include/linux/mtd/nand_ids.h. Add an entry, which contains following information:
{ name, manufacturer_id, model_idx, chipshift, page256, pageadrlen, erasesize }

ref	comment
name	string, which identifies the chip, usually the manufacturers chip name
manufacturer_id	manufacurer id code. This code is read during nand_scan. Use the existing defines for Toshiba and Samsung
model_idx	chip model code. This code is read during nand_scan. Check datasheet for the code of your chip
chipshift	chip size indicator. Chip size = (1 << chipshift)
page256	set this to 1, if your chip has pagelength 256 byte, else set it to 0.
pageadrlen	indicates the address size of your chip. 2 for devices < 64MB. 3 for devices >= 64MB
erasesize	the erasesize of your chip in bytes. Consult datasheet for proper value.

Please contact NAND driver maintainer to include it in the public source tree.

Config settings

The following config switches have to be set. JFFS2 on NAND **does not** work, if one of these settings is missing.

CONFIG_MTD=y
CONFIG_MTD_PARTITIONS=y
CONFIG_MTD_CHAR=y
CONFIG_MTD_BLOCK=y
CONFIG_MTD_NAND=y
CONFIG_MTD_NAND_ECC=y
CONFIG_MTD_NAND_YOURBOARD=y
CONFIG_JFFS2_FS=y
CONFIG_JFFS2_FS_DEBUG=0
CONFIG_JFFS2_FS_NAND=y

Make sure that fs/Config.in contains the following lines:
dep_tristate 'Journalling Flash File System v2 (JFFS2) support' CONFIG_JFFS2_FS $CONFIG_MTD
if ["$CONFIG_JFFS2_FS" = "y" -o "$CONFIG_JFFS2_FS" = "m"] ; then
int 'JFFS2 debugging verbosity (0 = quiet, 2 = noisy)' CONFIG_JFFS2_FS_DEBUG 0
bool 'JFFS2 support for NAND chips' CONFIG_JFFS2_FS_NAND
fi

FAQ

Can I boot from NAND Flash?

Not from a bare NAND chip. You need a glue logic around, which gives you memory access to the chip on bootup, like the DiskOnChip devices do. This will be a quite complex CPLD. An alternative is to use a small e.g. 1MB NOR Flash, which contains the boot code and maybe a compressed kernel image. Then you can use JFFS2 on NAND as your root filesystem

Some newer chips make the first page available for reading after power up. This could be helpful for starting a small 256/512 byte bootcode. At the time of this writing there is no tested implementation of this.

Samsungs S3C2410 ARM based SOC-CPU provides a mechanism to boot from NAND flash.

Is there support for 16/32bit wide NAND Flash ?

No. The generic NAND driver supports 8 bit wide NAND Flash only. 16 or 32 bit NAND Flash can be built by using 2 or 4 chips and connect them to D0-7, D8-D15, D16-D23 and D24-D31 on the data bus. You can tie all corresponding control signals together. But you have to build a new nand16 or nand32 driver, which can be derived from the existing nand.c. Be aware, that the writebuffer size is 2 or 4 times as big as on 8 bit NAND. This mean's, if you flush the buffer to ensure, that your data are on disk, you may waste much more memory space than on 8 bit NAND. Another point is bad block handling. When a block on 1 chip is bad, the corresponding blocks on the other chips are lost too, as you present them as one big block to the filesystem driver. The JFFS2 code, which handles the writebuffer and the out of band (spare) area of NAND doesn't support 16 / 32 bit neither.

How is ensured, that data is written to flash ?

As we have to handle a writebuffer for writing only full pages to the chips, there could be a loss of data, when the writebuffer is not flushed before power down. There are some mechanisms to ensure, that the writebuffer is flushed. You can force the flush of the writebuffer by using fsync() or sync() in your application. JFFS2 has a timed flush of the write buffer, which forces the flush of the buffer to flash, if there are no writes to the filesystem for more than 2 seconds. When you unmount the filesystem the buffer is flushed too.

Can I copy a JFFS2 Image to NAND via /dev/mtdX ?

Yes, as long as your chip does not contain bad blocks. Make sure, that the erasesize you set to mkfs.jffs2 is the same as the erasesize of your chip. If your kernel is set for JFFS2 on NAND and ECC is enabled, the data will be written with ECC.

Can I use mtdutils erase / eraseall

Yes, the latest nand driver code forces the protection of bad block information. It's safe to erase a NAND flash with erase(all) /dev/mtdX.

Must my bootloader be aware of NAND FLASH ?

Yes, if you use your bootloader to erase the FLASH chip and copy a filesystem image to it. For erase make sure, that you don't erase factory-marked bad blocks. They are marked in the 6th byte (offset 0x5) in the out of band area of the first page of a block. The block is bad, if any bit in this byte is zero. If you erase such a block, the bad block information is erased too and lost. Further use of this block will lead to erronous results

After the erase it's recommended to programm the JFFS2 erased marker into the out of band area of the first page in each erased block. Do not program it into the data area of the page !

For 256 byte pagesize devices program the following data into the out of band area:

Offset	0x06	0x07
Data	0x85	0x19

For 512 byte pagesize devices program the following data into the out of band area:

Offset	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f
Data	0x85	0x19	0x03	0x20	0x08	0x00	0x00	0x00

If you copy a filesystem image to the chip, it's recommended to write it with ECC. You can use the ECC code in the nand driver to do this. If you don't use ECC the nand driver will detect this, as long as you leave the eccvalid byte (offset 0x04) in the out of band area in the erased state (0xff). If you have a bad block on your chip, just skip this block and copy the data to the next block. JFFS2 is able to handle this gap.

References:

Open Source

JFFS2 and NTFL are located on this website

Hardware

Toshiba

Samsung

SSFDC Forum

M-Systems

Maintainers

JFFS2 is maintained by David Woodhouse

The generic NAND driver is maintained by Thomas Gleixner

Please don't contact them direct. Ask your questions on the mtd-mailing-list.

Any suggestions, improvements, bug-reports and bug-fixes are welcome

---

Thomas Gleixner