如何编写 linux 下 nand flash 驱动

Version: V0.1

Time:10/06/2008

Author: green-waste@163.com

【编写驱动之前要了解的知识】

1.硬件方面：

【Flash 的种类】

Flash 主要分 nand flash 和 nor flash

除了网上最流行的这个解释之外：

NAND和NOR的比较

我再多说几句，nor 的成本相对高，比较适合应用于存储少量的代码。

Nand flash 相对成本低，因此可以用来存储大量的数据，其在嵌入式系统中的作用，相当于 PC 上的硬盘，用于存储大量数据。

Nor flash，有类似于 dram 之类的地址总线，因此可以直接和 CPU 相连，CPU 可以直接通过地址总线对 nor flash 进行访问，而 nand flash 没有这类的总线，只有 IO 接口，只能通过 IO 接口发送命令和地址，对 nand flash 内部数据进行访问。相比之下，nor flash 就像是并行访问，nand flash 就是串行访问，所以相对来说，前者的速度更快些。

但是由于物理制程/制造方面的原因，导致 nor 和 nand 在一些具体操作方面的特性不同：

Nand 和 Nor Flash 的区别

	NOR	NAND	（备注）
接口	总线	I/O 接口	这个两者最大的区别
单个 cell 大小	大	小	
单个 Cell 成本	高	低	
读耗时	快	慢	
单字节的编程时间	快	慢	
多字节的编程时间	慢	快	
擦除时间	慢	快	
功耗	高	低，但是需要额外的 RAM	
是否可以执行代码	是	不行，但是一些新的芯片，可以在第一页之外执行一些小的 loader	即是否允许，**芯片内执行(XIP, eXecute In Place)**
位反转 Bit twiddling	几乎无限制	1-3 次，也称作 “部分页编程限制”	也就是数据错误了？
在芯片出厂时候是否允许坏块	不允许	允许	

【Nand Flash 的种类】

具体再分，又可以分为

1)Bare NAND chips：裸片，单独的 nand 芯片

2)SmartMediaCards： =裸片+一层薄塑料，常用于数码相机和 MP3 播放器中。之所以称 smart，是由于其软件 smart，而不是硬件本身有啥 smart 之处。^_^

3)DiskOnChip：裸片+glue logic，glue logic=硬件 ECC 产生器+用于静态的 nand 芯片控制的寄存器+直接访问一小片地址窗口，那块地址中包含了引导代码的 stub 桩，其可以从 nand flash 中拷贝真正的引导代码。

【Nand flash 的特点】

Nand flash 的操作，和其他一些常见的设备，如硬盘等，不同，其有自己特殊的方式。

其特殊就在于：

1.Nand flash 的最小单位是页 page，而不是其他很多设备所说的位 bit。

2.写入数据之前必须先进行擦除 erase 操作

3.写的时候，最小单位是页 page，对也进行写操作，也称作“页编程”，page programming

4.擦除的最小单位是块 block

5.由于物理特性，容易出错，所以无论是读还是写，都要采取检测和校验，即 EDC。

6.nand flash 出厂时候，就有一定坏的块 block，成为换块，并且做了一定标记。

7.nand flash 中有个额外的空间，叫做 spare area/oob

【spare area/oob】

Nand 由于最初硬件设计时候考虑到，额外的错误校验等需要空间，专门对应每个页，额外设计了叫做 spare area 空区域，在其他地方，比如 jffs2 文件系统中，也叫做 oob（out of band）数据。

其具体用途，总结起来有：

1. 标记是否是坏快
2. 存储 ECC 数据
3. 存储一些和文件系统相关的数据，如 jffs2 就会用到这些空间存储一些特定信息

【常见 Nand Flash 的大小及参数】

常见的 nand flash 的大小，由最开始的小于 256M,到现在的常见的 1G，2G，甚至更大。

以前的 nand flash 的

Pagesize 页大小，多为 512B+16B 的 oob，block 大小为 64*（512B+16B）=32KB+1KB

现在目前市场上见到的，绝大多数，都是新的 nand falsh，其 Pagesize 页大小多为 2KB+64B 的 oob，block 大小多为 64pages 页=64*（2K+64B）=128KB+4KB，一个 nand flash 中的芯片，一般含有 4096 个块，比如 samsung 的 K9F4G08U0M，所以这个 nand flash 大小就是

4096 Blocks = 4096 * 64 *（2K+64B）=512MB

即：

1 Page = (2K + 64)Bytes

1 Block = (2K + 64)B x 64 Pages

= (128K + 4K) Bytes

1 Device = (2K+64)B x 64Pages x 4,096 Blocks

= 4,224 Mbits =512MB

【Nand flash 工作原理】

所谓工作原理，其实也就是对应对其如何操作的。

还是以上面提到的 samsung 的 K9F4G08U0M 的 nand flash 为例，简单描述如下：

1.nand flash 定义了一些引脚，使得你可以发送命令过去，实现对 nand flash 的操控：

Pin Name	Pin Function
$I/O_0 \sim I/O_7$	**DATA INPUTS/OUTPUTS** The I/O pins are used to input command, address and data, and to output data during read operations. The I/O pins float to high-z when the chip is deselected or when the outputs are disabled.
CLE	**COMMAND LATCH ENABLE** The CLE input controls the activating path for commands sent to the command register. When active high, commands are latched into the command register through the I/O ports on the rising edge of the $\overline{WE}$ signal.
ALE	**ADDRESS LATCH ENABLE** The ALE input controls the activating path for address to the internal address registers. Addresses are latched on the rising edge of $\overline{WE}$ with ALE high.
$\overline{CE}$	**CHIP ENABLE** The $\overline{CE}$ input is the device selection control. When the device is in the Busy state, $\overline{CE}$ high is ignored, and the device does not return to standby mode in program or erase operation.
$\overline{RE}$	**READ ENABLE** The $\overline{RE}$ input is the serial data-out control, and when active drives the data onto the I/O bus. Data is valid tREA after the falling edge of $\overline{RE}$ which also increments the internal column address counter by one.
$\overline{WE}$	**WRITE ENABLE** The $\overline{WE}$ input controls writes to the I/O port. Commands, address and data are latched on the rising edge of the $\overline{WE}$ pulse.
$\overline{WP}$	**WRITE PROTECT** The $\overline{WP}$ pin provides inadvertent program/erase protection during power transitions. The internal high voltage generator is reset when the $\overline{WP}$ pin is active low.
$R/\overline{B}$	**READY/BUSY OUTPUT** The $R/\overline{B}$ output indicates the status of the device operation. When low, it indicates that a program, erase or random read operation is in process and returns to high state upon completion. It is an open drain output and does not float to high-z condition when the chip is deselected or when outputs are disabled.
Vcc	**POWER** Vcc is the power supply for device.
Vss	**GROUND**
N.C	**NO CONNECTION** Lead is not internally connected.

上面这些引脚，需要解释的主要是

CLE，使能，只有使能有效，你才能进行后续的操作。只能选中了这个 nand 芯片，才能进行后续的读写。

ALE，在发送地址时，要锁住地址总线，才可以操作。

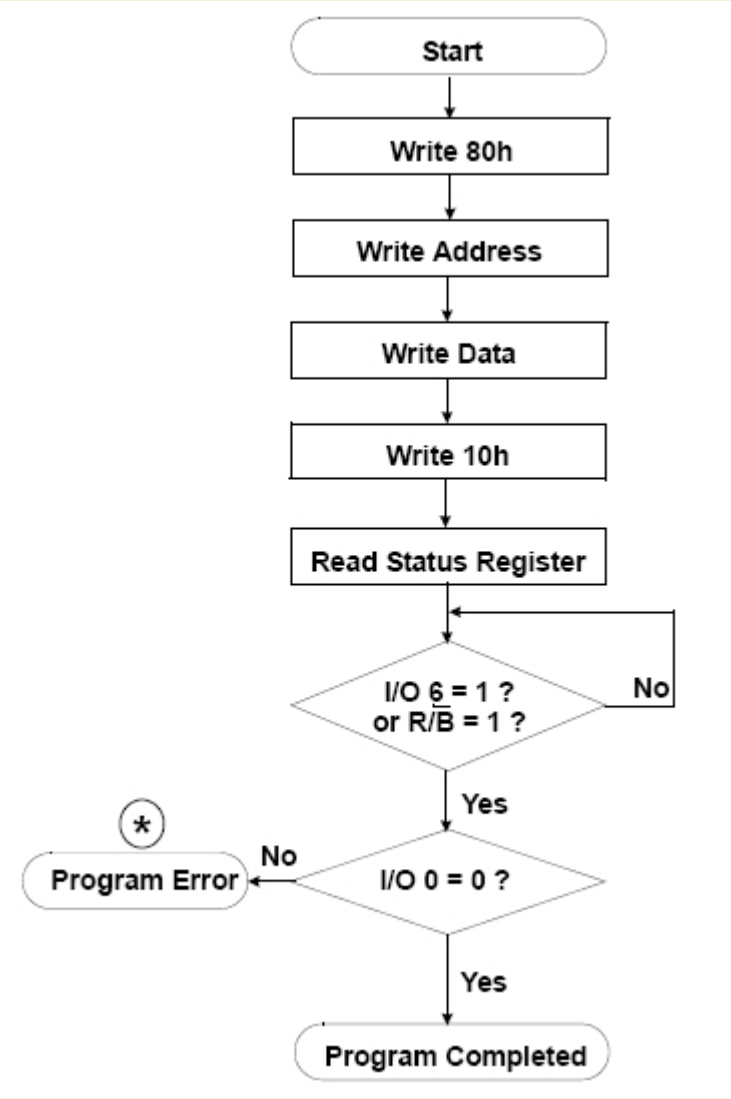
RE，WE，在读写之前，要对应的引脚有效，才可以进行读写的。

其他见解释皆可。

所有如上的引脚，都是驱动中，通过发送命令，具体内部控制逻辑去实现的。

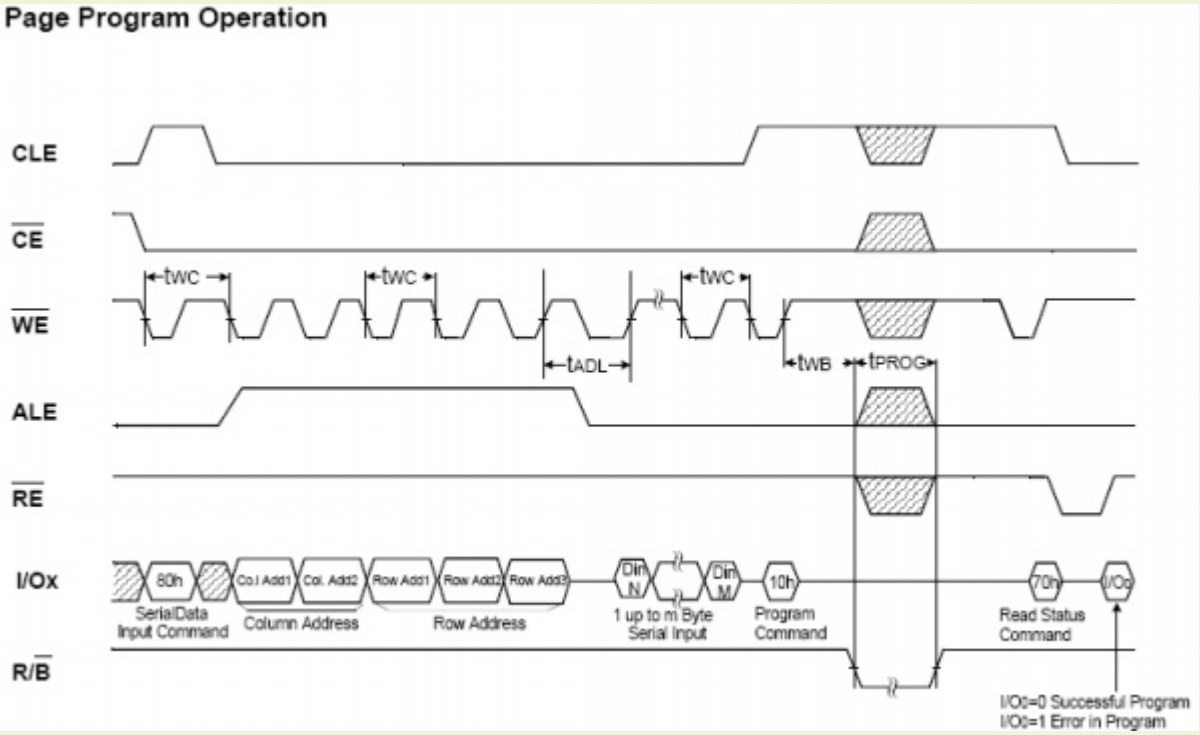
而驱动开发者要关心的，是何时去发送对应的命令。具体想要实现一操作，要何时发送什么命令才能实现，要继续看下面的 datasheet 中的解释：

如果想要对 nand flash 操作，就要根据 datasheet 中的规范进行，比如对页的写操作/写编程要根据这个顺序去操作：



其具体的细节，包含了哪些操作，可以看这个时序图：

Page Program Operation



从上图可以看出来，要先发 80h 的命令，再继续后面的操作。

这里的 80h 命令的含义，参考这个图表：

Table 1. Command Sets

Function	1st. Cycle	2nd. Cycle	Acceptable Command during Busy
Read	00h	30h	
Read for Copy Back	00h	35h	
Read ID	90h	-	
Reset	FFh	-	O
Page Program	80h	10h	
Two-Plane Page Program	80h---11h	81h---10h	
Copy-Back Program	85h	10h	
Two-Plane Copy-Back Program	85h---11h	81h---10h	
Block Erase	60h	D0h	
Two-Plane Block Erase	60h---60h	D0h	
Random Data Input(1)	85h	-	
Random Data Output(1)	05h	E0h	
Read Status	70h		O
Read EDC Status(2)	7Bh		O

从这个图表，可以看到，80h 就是 page program 的第一个命令。

而时序图中的先两个 col 列地址，后三个 row 行地址，是根据下面这个图得来的：

	I/O 0	I/O 1	I/O 2	I/O 3	I/O 4	I/O 5	I/O 6	I/O 7	
1st Cycle	A0	A1	A2	A3	A4	A5	A6	A7	Column Address
2nd Cycle	A8	A9	A10	A11	*L	*L	*L	*L	Column Address
3rd Cycle	A12	A13	A14	A15	A16	A17	A18	A19	Row Address
4th Cycle	A20	A21	A22	A23	A24	A25	A26	A27	Row Address
5th Cycle	A28	A29	*L	*L	*L	*L	*L	*L	Row Address

NOTE : Column Address : Starting Address of the Register.

* L must be set to "Low".

* The device ignores any additional input of address cycles than reguired.

此时才能定位到你所要操作的页，才能进行写操作。

再多说几句，上图表示了，如果想要对这个 nand flash 操作，读或写，需要进行 3 次列地址和三次的行地址，才能定位到你所要操作的地方，才能读写。同时显示了，每次每个位，对应代表的地址位。

此处的 2 个列地址和 3 个行地址，是由硬件设计决定的。

老的 nand flash 由于每个页只有 512B，所以，最后的操作，只需要 1 个列地址和 3 个行地址就可以定位了。感兴趣的可以去网上下载 samsung 一些老的 nand flash，可以对比着看，更明白些。

而每个操作，比如上面的写操作，都是需要一定时间的，其具体硬件操作所需的时间，见这个图：

Program / Erase Characteristics

Parameter		Symbol	Min	Typ	Max	Unit
Program Time		tPROG(2)	-	200	700	µs
Dummy Busy Time for Two-Plane Page Program		tDBSY	-	0.5	1	µs
Number of Partial Program Cycles in the Same Page	Main Array	Nop	-	-	4	cycles
	Spare Array		-	-	4	cycles
Block Erase Time		tBERS	-	1.5	2	ms

此处可以看到，写一个页，即页编程，一般需要 200us，最大需要 700us，所以驱动开发者在开发的时候，要知道这个细节，要等待对应的时间，再去判断，是否写操作顺利完成了，才能接下来继续后面其他的操作的。

上述的具体代码实现，可以去看

drivers\mtd\nand\nand_base.c 中的 nand_command_lp()函数中的代码：

```
/* Command latch cycle */
//对命令锁存，然后才能发送命令
        chip->cmd_ctrl(mtd, command & 0xff,
                       NAND_NCE | NAND_CLE | NAND_CTRL_CHANGE);

        if (column != -1 || page_addr != -1) {
                int ctrl = NAND_CTRL_CHANGE | NAND_NCE | NAND_ALE;

//先发送 2 次 col 地址，构成整个列地址
                /* Serially input address */
                if (column != -1) {
                        /* Adjust columns for 16 bit buswidth */
                        if (chip->options & NAND_BUSWIDTH_16)
```

```
                column >>= 1;
            chip->cmd_ctrl(mtd, column, ctrl);
            ctrl &= ~NAND_CTRL_CHANGE;
            chip->cmd_ctrl(mtd, column >> 8, ctrl);
        }
//再发送 3 次 row 地址，构成整个行地址
        if (page_addr != -1) {
            chip->cmd_ctrl(mtd, page_addr, ctrl);
            chip->cmd_ctrl(mtd, page_addr >> 8,
                        NAND_NCE | NAND_ALE);
            /* One more address cycle for devices > 128MiB */
            if (chip->chipsize > (128 << 20))
                chip->cmd_ctrl(mtd, page_addr >> 16,
                            NAND_NCE | NAND_ALE);
        }
    }
```

这样就可以定位到我们要操作的位置，进行操作了。

2.软件方面：

Linux 驱动原理

具体内部很多实现，已经包含在 drivers\mtd\nand\nand_base.c 中了

【nand flash 驱动加载识别 nand 类型过程】

在驱动加载的时候，会去调用：

nand_get_flash_type（）

其中，就会对 nand 的类型和其他相关参数进行检查。

1） 选中对应设备，如果此时只有一个 nand 芯片，则此步可以省略

```
    /* Select the device */
    chip->select_chip(mtd, 0);
```

x

2） 发读命令，去读取设备类型代码

```
    /* Send the command for reading device ID */
    chip->cmdfunc(mtd, NAND_CMD_READID, 0x00, -1);
```

3） 判断是哪个厂商的，哪个类型的 flash

```
    /* Read manufacturer and device IDs */
    *maf_id = chip->read_byte(mtd);
    dev_id = chip->read_byte(mtd);
```

4） 在事先已经定义好的 nand flash 类型中查找属于何种厂商和型号

```
    /* Lookup the flash id */
    for (i = 0; nand_flash_ids[i].name != NULL; i++) {
        if (dev_id == nand_flash_ids[i].id) {
            type =  &nand_flash_ids[i];
            break;
        }
```

}

5） 继续判断具体 nand flash 的各个参数，包括
芯片信息，Pagesize 页大小，oobsize 即 oob 的大小，blocksize 块大小，buswidth 总线宽度是 8 位还是 16 位。

如果页大小不是之前老的 nand 的 512B，而是新的 nand 的 2K 或更大，则后面对应的发送给 nand flash 命令的的时候，调用的函数就由 nand_command()变成 nand_command_lp()了。
后者主要比前者多一发个命令：
chip->cmd_ctrl(mtd, column >> 8, ctrl);
即，多发一个列地址命令。
因为大页面（>2KB）的寻址需要 2 次 column，而小页面（512B）只需要 1 次的列地址。
具体可以参考 nand flash 的 datasheet。

6） 接着会做一些其他初始化操作，包括最后调用 nand_set_defaults()去实现的默认函数的挂载，如果你的 nand flash 驱动没有实现的话，就是挂载默认的了。

【模块加载原理】
这个内容太大，此处只是简单说说。
驱动加载的功能主要是 probe 函数实现的，主要去识别设备的类型和各个参数，并且为设备的使用进行正常的初始化。
对应卸载时候执行的 remove 函数，施放对应的，之前申请的一些资源。

【MTD 设备】
在 Linux 下，将 nand Flash 等设备归属到 MTD 设备下进行统一管理。
Mtd，即 memory technology deveice，即将 nand 看出是存储设备来管理。
之所以会这么说和这么做，是因为前面提高的 nand flash 和普通硬盘等设备的特殊性：
IO 接口，最小单位是页，写前需擦除等，导致了，不能像平常对待硬盘等操作一样去操作 nand flash，只能采取一些特殊方法，这就诞生了 mtd 设备的统一抽象层，将 nand flash，nor flash 和其他类型的 flash 等设备，统一抽象成 mtd 设备来管理，根据这些设备的特点，上层实现了常见的操作函数封装，底层具体的内部实现，就需要驱动设计者自己来实现了。

MTD 设备和硬盘设备之前的区别

HARD drives	**MTD device**
连续的扇区	连续的可擦除块
扇区都很小(512B,1024B)	可擦除块比较大 (32KB,128KB)
主要通过两个操作对其维护操作：读扇区，写扇区	主要通过三个操作对其维护操作：从擦除块中读，写入擦除块，**擦写可擦除块**
坏快被重新映射，并且被硬件隐藏起来了（至少是在如今常见的 LBA 硬盘设备中是如此）	坏的可擦除块没有被隐藏，软件中要处理对应的坏块问题。
HDD 扇区没有擦写寿命超出的问题。	可擦除块是有擦除次数限制的，大概是 10^4-10^5次.

【Linux 下 nand flash 驱动编写步骤简介】

1. 了解硬件的 nand flash 的各个参数和工作原理

具体参考 nand flash 的 datasheet，主要包括，自己 nand flash 的厂商，型号等。

Nand flash 的页大小，oob 大小，块大小，位宽 8bit 还是 16bit。

工作原理，上面已经做了一定描述，不清楚的，可以参考 datasheet，多看看，就会明白很多。

2. 按照 linux 下驱动编写规范编写 nand flash 驱动，

可以参考其他已经有的驱动，比如内核源码中已经有的

drivers\mtd\nand\s3c2410.c

就是个很好的例子。

自己以其为模板，实现自己板子的 nand flash 驱动。

其实主要工作就是，实现

```
static struct platform_driver s3c2410_nand_driver = {
        .probe          = s3c2410_nand_probe,
        .remove         = s3c2410_nand_remove,
        .suspend    = s3c24xx_nand_suspend,
        .resume         = s3c24xx_nand_resume,
        .driver         = {
                .name       = "s3c2410-nand",
                .owner      = THIS_MODULE,
        },
};
```

中的

XXX_nand_probe 函数

XXX_nand_remove 函数

XXX_nand_enable_hwecc，如果支持硬件 ecc 的话。

对 nand flash 的读写，这两个函数，实现了对 nand 的具体操作。

【Linux 下 Nand Flash 驱动编写简单步骤】

软件和硬件知识，都已经了解的话，由于上层的 linux 的 mtd 框架中，已经完全封装好了，对 nand flash 的 write page，write oob 等相关函数的实现，那么剩下的只是相对来说已经是很少量的，关于 nand 驱动具体内部操作方面的工作：

1.初始化

先是在 nand 芯片初始化的时候，对其

XXX_nand_init_chip（）

给对应的芯片 chip 赋给对应的

XXX_nand_read_buf 和 XXX_nand_write_buf 等函数：

```
        chip->cmd_ctrl    = XXX_nand_hwcontrol;
        chip->dev_ready = XXX_nand_devready;
        chip->read_buf    = XXX_nand_read_buf;
        chip->write_buf    = XXX_nand_write_buf;
```

以实现后续的对 nand 芯片的操作。

然后根据 ecc 类型，赋给对应的 ecc 的校验与纠错函数：

```
chip->ecc.hwctl         = XXX_nand_enable_hwecc;
chip->ecc.calculate = XXX _nand_calculate_ecc;
```

3. 实现上面提到的对应的各个函数，关于如何实现，参考一下其他 nand 驱动，就会理解很多了。
4. 驱动测试，参考具体的 ldd3（Linux Device Driver version 3）的测试相关部分内容。

说得很乱，希望对大家有些帮助。

【MTD 设备驱动推荐文章】

关于 mtd 设备驱动，感兴趣的可以去参考

MTD原始设备与FLASH硬件驱动的对话

MTD原始设备与FLASH硬件驱动的对话-续

那里，算是比较详细地介绍了整个流程，方便大家理解整个 mtd 框架和 nand flash 驱动。

【引用文章】

Brief Intro of Nand Flash

Samsung K9F4G08U0M.pdf （Samsung Nand Flash 的 Datasheet ， nand 型号是 K9F4G08U0M）

nand falsh read operation