**NandFlash详述**

**1. 硬件特性：**

**【Flash的硬件实现机制】**

Flash全名叫做Flash Memory，属于非易失性存储设备(Non-volatile Memory Device)，与此相对应的是易失性存储设备(Volatile Memory Device)。这类设备，除了Flash，还有其他比较常见的如硬盘，ROM等，

与此相对的，易失性就是断电了，数据就丢失了，比如大家常用的内存，不论是以前的SDRAM，DDR SDRAM，还是现在的DDR2，DDR3等，都是断电后，数据就没了。

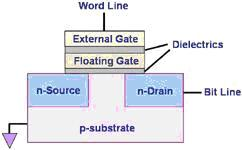
Flash的内部存储是**MOSFET**，里面有个悬浮门(Floating Gate)，是真正存储数据的单元。

-------------------------------------------------------------------------------------------------------------------------

金属-氧化层-半导体-场效[晶体管](http://baike.baidu.com/view/30363.htm)，简称**金氧半场效晶体管**（Metal-Oxide-Semiconductor Field-Effect Transistor, MOSFET）是一种可以广泛使用在模拟[电路](http://baike.baidu.com/view/134362.htm)与数字电路的场效晶体管（field-effect transistor）。MOSFET依照其“通道”的极性不同，可分为n-type与p-type的MOSFET，通常又称为NMOSFET与PMOSFET，其他简称尚包括NMOS FET、PMOS FET、nMOSFET、pMOSFET等。

-------------------------------------------------------------------------------------------------------------------------

在Flash之前，紫外线可擦除(uv-erasable)的EPROM，就已经采用用Floating Gate存储数据这一技术了。



**图1.典型的Flash内存单元的物理结构**

数据在Flash内存单元中是的。

存储电荷的多少，取决于图中的外部门（external gate）所被施加的电压，其控制了是向存储单元中冲入电荷还是使其释放电荷。

数据的表示，以所存储的电荷的电压是否超过一个特定的阈值Vth来表示。

**【SLC和MLC的实现机制】**

Nand Flash按照内部存储数据单元的电压的不同层次，也就是单个内存单元中，是存储1位数据，还是多位数据，可以分为**SLC**和**MLC**：

**1. SLC，Single Level Cell:**

单个存储单元，只存储一位数据，表示成1或0.

就是上面介绍的，对于数据的表示，单个存储单元中内部所存储电荷的电压，和某个特定的阈值电压Vth，相比，如果大于此Vth值，就是表示1，反之，小于Vth，就表示0.

对于nand Flash的数据的写入1，就是控制External Gate去充电，使得存储的电荷够多，超过阈值Vth，就表示1了。而对于写入0，就是将其放电，电荷减少到小于Vth，就表示0了。

关于为何Nand Flash不能从0变成1，我的理解是，物理上来说，是可以实现每一位的，从0变成1的，但是实际上，对于实际的物理实现，出于效率的考虑，如果对于，每一个存储单元都能单独控制，即，0变成1就是，对每一个存储单元单独去充电，所需要的硬件实现就很复杂和昂贵，同时，所进行对块擦除的操作，也就无法实现之前的，一闪而过的速度了，也就失去了Flash的众多特性了。

// 也就是放电的思路还是容易些。1->0

**2. MLC，Multi Level Cell：**

与SLC相对应，就是单个存储单元，可以存储多个位，比如2位，4位等。其实现机制，说起来比较简单，就是，通过控制内部电荷的多少，分成多个阈值，通过控制里面的电荷多少，而达到我们所需要的存储成不同的数据。比如，假设输入电压是Vin＝4V（实际没有这样的电压，此处只是为了举例方便），那么，可以设计出2的2次方＝4个阈值，1/4的Vin＝1V，2/4的Vin＝2V，3/4的Vin＝3V，Vin＝4V，分别表示2位数据00，01，10，11，对于写入数据，就是充电，通过控制内部的电荷的多少，对应表示不同的数据。

对于读取，则是通过对应的内部的电流（与Vth成反比），然后通过一系列解码电路完成读取，解析出所存储的数据。这些具体的物理实现，都是有足够精确的设备和技术，才能实现精确的数据写入和读出的。

单个存储单元可以存储2位数据的，称作2的2次方＝4 Level Cell，而不是2 Level Cell；

同理，对于新出的单个存储单元可以存储4位数据的，称作2的4次方＝16 Level Cell。

**【关于如何识别SLC还是MLC】**

Nand Flash设计中，有个命令叫做Read ID，读取ID，意思是读取芯片的ID，就像大家的身份证一样，这里读取的ID中，是：

读取好几个字节，一般最少是4个，新的芯片，支持5个甚至更多，从这些字节中，可以解析出很多相关的信息，比如：

此Nand Flash内部是几个芯片（chip）所组成的，

每个chip包含了几片（Plane），

每一片中的页大小，块大小，等等。

在这些信息中，其中有一个，就是识别此flash是SLC还是MLC。下面这个就是最常见的Nand Flash的datasheet中所规定的，第3个字节，3rd byte，所表示的信息，其中就有SLC/MLC的识别信息：

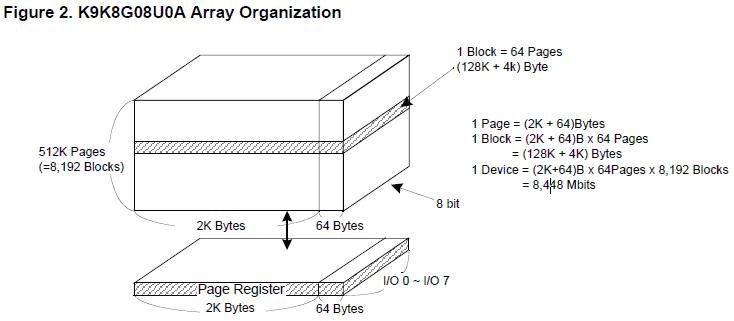
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Description** | **I/O7** | **I/O6** | **I/O5 I/O4** | **I/O3 I/O2** | **I/O1 I/O0** |
| **Internal**  **Chip Number** | **1**  **2**  **4**  **8** |  |  |  |  | **0 0**  **0 1**  **1 0**  **1 1** |
| **Cell Type** | **2 Level Cell**  **4 Level Cell**  **8 Level Cell**  **16 Level Cell** |  |  |  | **0 0**  **0 1**  **1 0**  **1 1** |  |
| **Number of**  **Simultaneously**  **Programmed Pages** | **1**  **2**  **4**  **8** |  |  | **0 0**  **0 1**  **1 0**  **1 1** |  |  |
| **Interleave Program**  **Between multiple chips** | **Not Support**  **Support** |  | **0**  **1** |  |  |  |
| **Cache Program** | **Not Support**  **Support** | **0**  **1** |  |  |  |  |

**表1.Nand Flash第3个ID的含义**

**【Nand Flash的物理存储单元的阵列组织结构】**

Nand flash的内部组织结构，此处还是用图来解释，比较容易理解：

**图2.Nand Flash物理存储单元的阵列组织结构**

****

上图是K9K8G08U0A的datasheet中的描述。

简单解释就是:

**1.**一个nand flash由很多个块（Block）组成，

块的大小一般是

-> 128KB，

-> 256KB，

-> 512KB

此处是128KB。

**2.**每个块里面又包含了很多页（page）。每个页的大小，

老的nand flash，页大小是256B，512B，

这类的nand flash被称作small block，。地址周期只有4个。

对于现在常见的nand flash多数是2KB，

被称作big block，对应的发读写命令地址，一共5个周期(cycle)，

更新的nand flash是4KB，

**块**，也是Nand Flash的擦除操作的基本/最小单位。

**3.**每一个页，对应还有一块区域，叫做空闲区域（spare area）/冗余区域（redundant area），而Linux系统中，一般叫做**OOB**（Out Of Band），这个区域，是最初基于Nand Flash的硬件特性：数据在读写时候相对容易错误，所以为了保证数据的正确性，必须要有对应的检测和纠错机制，此机制被叫做EDC(Error Detection Code)/ECC（Error Code Correction,或者Error Checking and Correcting），所以设计了多余的区域，用于放置数据的校验值。

**页**, 是Nand Flash的写入操作的基本/最小的单位。

【Nand Flash数据存储单元的整体架构】

简单说就是，常见的nand flash，内部只有一个chip，每个chip只有一个plane。

而有些复杂的，容量更大的nand flash，内部有多个chip，每个chip有多个plane。这类的nand flash，往往也有更加高级的功能，比如下面要介绍的Multi Plane Program和Interleave Page Program等。

**比如**，型号为**K9K8G08U0A**这个芯片（chip），

内部有:

K9F4G08U0A (256MB) **:** Plane (1Gb), Plane (1Gb)

K9F4G08U0A (256MB) **:** Plane (1Gb), Plane (1Gb)

K9WAG08U1A ，内部包含了2个K9K8G08U0A

K9NBG08U5A ，内部包含了4个K9K8G08U0A

【Flash名称的由来】

Flash的擦除操作是以block块为单位的，与此相对应的是其他很多存储设备，是以bit位为最小读取/写入的单位，Flash是一次性地擦除整个块：在发送一个擦除命令后，一次性地将一个block，常见的块的大小是128KB/256KB。。，**全部擦除为1，也就是里面的内容全部都是0xFF了**，由于是一下子就擦除了，相对来说，擦除用的时间很短，可以用一闪而过来形容，所以，叫做Flash Memory。中文有的翻译为（快速）闪存。

【Flash相对于普通设备的特殊性】

1. 上面提到过的，Flash最小操作单位，有些特殊。

一般设备，比如硬盘/内存，读取和写入都是以bit位为单位，读取一个bit的值，将某个值写入对应的地址的位，都是可以按位操作的。

但是Flash由于物理特性，使得内部存储的数据，只能从1变成0，这点，可以从前面的内部实现机制了解到，只是方便统一充电，不方便单独的存储单元去放电，所以才说，只能从1变成0，也就是释放电荷。

所以，总结一下Flash的特殊性如下：

|  |  |  |
| --- | --- | --- |
|  | 普通设备(硬盘/内存等) | Flash |
| 读取/写入的叫法 | 读取/写入 | 读取/编程(Program)① |
| 读取/写入的最小单位 | Bit/位 | Page/页 |
| 擦除(Erase)操作的最小单位 | Bit/位 | Block/块② |
| 擦除操作的含义 | 将数据删除/全部写入0 | 将整个块都擦除成全是1，也就是里面的数据都是0xFF③ |
| 对于写操作 | 直接写即可 | 在写数据之前，要先擦除，然后再写 |

表2.Flash和普通设备相比所具有的特殊性

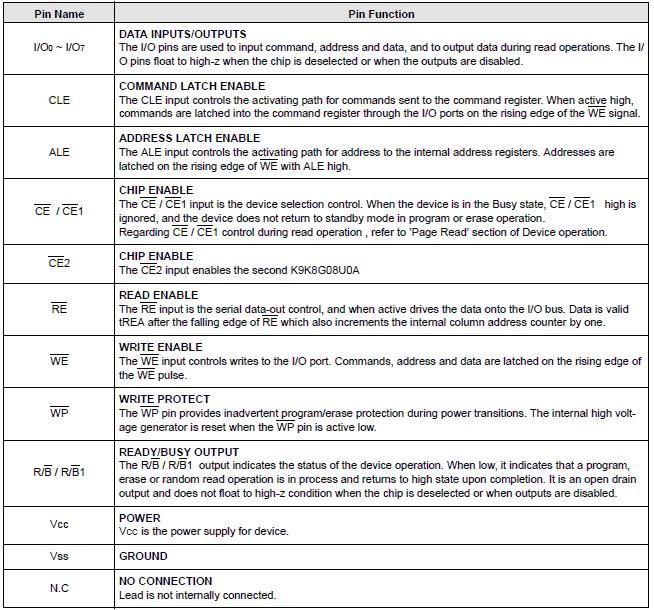
注：

①之所以将写操作叫做编程，是因为，flash和之前的EPROM，EEPROM继承发展而来，而之前的EEPROM(Electrically Erasable Programmable Read-Only Memory)，往里面写入数据，就叫做编程Program，之所以这么称呼，是因为其对数据的写入，是需要用电去擦除/写入的，就叫做编程。

②对于目前常见的页大小是2K/4K的Nand Flash，其块的大小有128KB/256KB/512KB等。而对于Nor Flash，常见的块大小有64K/32K等。

③在写数据之前，要先擦除，内部就都变成0xFF了，然后才能写入数据，也就是将对应位由1变成0。

**【Nand Flash引脚(Pin)的说明】**

****

**图3.Nand Flash引脚功能说明**

上图是常见的Nand Flash所拥有的引脚（Pin）所对应的功能，简单翻译如下：

1. I/O0 ~ I/O7：用于输入地址/数据/命令，输出数据

2. CLE：Command Latch Enable，命令锁存使能，在输入命令之前，要先在模式寄存器中，设置CLE使能

3. ALE：Address Latch Enable，地址锁存使能，在输入地址之前，要先在模式寄存器中，设置ALE使能

4. CE#：Chip Enable，芯片使能，在操作Nand Flash之前，要先选中此芯片，才能操作

5. RE#：Read Enable，读使能，在读取数据之前，要先使CE＃有效。

6. WE#：Write Enable，写使能,在写取数据之前，要先使WE＃有效。

7. WP#：Write Protect，写保护

8. R/B#:Ready/Busy Output,就绪/忙,主要用于在发送完编程/擦除命令后,检测这些操作是否完成,忙,表示编程/擦除操作仍在进行中,就绪表示操作完成.

9. Vcc：Power，电源

10. Vss：Ground，接地

11. N.C：Non-Connection,未定义，未连接。

**[小常识]**

在数据手册中，你常会看到，对于一个引脚定义，有些字母上面带一横杠的，那是说明此引脚/信号是低电平有效，比如你上面看到的RE头上有个横线，就是说明，此RE是低电平有效，此外，为了书写方便，在字母后面加“＃”，也是表示低电平有效，比如我上面写的CE＃；如果字母头上啥都没有，就是默认的高电平有效，比如上面的CLE，就是高电平有效。

**【为何需要ALE和CLE】**

突然想明白了，Nand Flash中,为何设计这么多的命令,把整个系统搞这么复杂的原因了:

比如命令锁存使能(Command Latch Enable,CLE)和地址锁存使能(Address Latch Enable，ALE)，那是因为，Nand Flash就8个I/O，而且是复用的，也就是，可以传数据，也可以传地址，也可以传命令，为了区分你当前传入的到底是啥，所以，先要用发一个CLE（或ALE）命令，告诉nand Flash的控制器一声，我下面要传的是命令（或地址），这样，里面才能根据传入的内容，进行对应的动作。否则,nand flash内部,怎么知道你传入的是数据,还是地址,还是命令啊,也就无法实现正确的操作了.

**【Nand Flash只有8个I/O引脚的好处】**

1. 减少外围引脚：相对于并口(Parellel)的Nor Flash的48或52个引脚来说，的确是大大减小了引脚数目，这样封装后的芯片体积，就小很多。现在芯片在向体积更小，功能更强，功耗更低发展，减小芯片体积，就是很大的优势。同时，减少芯片接口，也意味着使用此芯片的相关的外围电路会更简化，避免了繁琐的硬件连线。

2. 提高系统的可扩展性，因为没有像其他设备一样用物理大小对应的完全数目的addr引脚，在芯片内部换了芯片的大小等的改动，对于用全部的地址addr的引脚，那么就会引起这些引脚数目的增加，比如容量扩大一倍，地址空间/寻址空间扩大一倍，所以，地址线数目/addr引脚数目，就要多加一个，而对于统一用8个I/O的引脚的Nand Flash，由于对外提供的都是统一的8个引脚，内部的芯片大小的变化或者其他的变化，对于外部使用者(比如编写nand flash驱动的人)来说，不需要关心，只是保证新的芯片，还是遵循同样的接口，同样的时序，同样的命令，就可以了。这样就提高了系统的扩展性。

**【Nand flash的一些典型(typical)特性】**

1.页擦除时间是200us，有些慢的有800us。

2.块擦除时间是1.5ms.

3.页数据读取到数据寄存器的时间一般是20us。

4.串行访问（Serial access）读取一个数据的时间是25ns，而一些旧的nand flash是30ns，甚至是50ns。

5.输入输出端口是地址和数据以及命令一起multiplex复用的。

以前老的Nand Flash，编程/擦除时间比较短，比如K9G8G08U0M，才5K次，而后来很多6.nand flash的编程/擦除的寿命，最多允许的次数，以前的nand flash多数是10K次，也就是1万次，而现在很多新的nand flash，技术提高了，比如，Micron的MT29F1GxxABB，Numonyx的NAND04G-B2D/NAND08G-BxC，都可以达到100K，也就是10万次的编程/擦除。和之前常见的Nor Flash达到同样的使用寿命了。

7.48引脚的TSOP1封装或 52引脚的ULGA封装

**【Nand Flash中的特殊硬件结构】**

由于nand flash相对其他常见设备来说，比较特殊，所以，特殊的设备，也有特殊的设计，所以，有些特殊的硬件特性，就有比较解释一下：

1. 页寄存器（Page Register）：由于Nand Flash读取和编程操作来说，一般最小单位是页，所以，nand flash在硬件设计时候，就考虑到这一特性，对于每一片，都有一个对应的区域，专门用于存放，将要写入到物理存储单元中去的或者刚从存储单元中读取出来的，一页的数据，这个数据缓存区，本质上就是一个buffer，但是只是名字叫法不同，datasheet里面叫做Page Register，此处翻译为页寄存器，实际理解为页缓存，更为恰当些。

注意：只有写到了这个页缓存中，只有等你发了对应的编程第二阶段的确认命令0x10之后，实际的编程动作才开始，才开始把页缓存中的数据，一点点写到物理存储单元中去。

所以，简单总结一下就是，对于数据的流向，实际是经过了如下步骤：

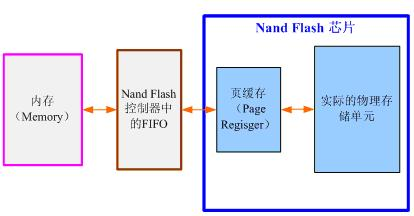


图4 Nand Flash读写时的数据流向

**【Nand Flash中的坏块(Bad Block)】**

Nand Flash中，一个块中含有1个或多个位是坏的，就成为其为坏块。

坏块的稳定性是无法保证的，也就是说，不能保证你写入的数据是对的，或者写入对了，读出来也不一定对的。而正常的块，肯定是写入读出都是正常的。

坏块有两种：

（1）一种是出厂的时候，也就是，你买到的新的，还没用过的Nand Flash，就可以包含了坏块。此类出厂时就有的坏块，被称作factory (masked)bad block或initial bad/invalid block，在出厂之前，就会做对应的标记，标为坏块。

具体标记的地方是，对于现在常见的页大小为2K的Nand Flash，是块中第一个页的oob起始位置（关于什么是页和oob，下面会有详细解释）的第1个字节（旧的小页面，pagesize是512B甚至256B的nand flash，坏块标记是第6个字节），如果不是0xFF，就说明是坏块。相对应的是，所有正常的块，好的块，里面所有数据都是0xFF的。

（2）第二类叫做在使用过程中产生的，由于使用过程时间长了，在擦块除的时候，出错了，说明此块坏了，也要在程序运行过程中，发现，并且标记成坏块的。具体标记的位置，和上面一样。这类块叫做worn-out bad block。

对于坏块的管理，在Linux系统中，叫做坏块管理（BBM，Bad Block Managment），对应的会有一个表去记录好块，坏块的信息，以及坏块是出厂就有的，还是后来使用产生的，这个表叫做 **坏块表（BBT，Bad Block Table）**。在Linux内核MTD架构下的Nand Flash驱动，和Uboot中Nand Flash驱动中，在加载完驱动之后，如果你没有加入参数主动要求跳过坏块扫描的话，那么都会去主动扫描坏块，建立必要的BBT的，以备后面坏块管理所使用。

而关于好块和坏块，Nand Flash在出厂的时候，会做出保证：

1.关于好的，可以使用的块的数目达到一定的数目，比如三星的K9G8G08U0M，整个flash一共有4096个块，出厂的时候，保证好的块至少大于3996个，也就是意思是，你新买到这个型号的nand flash，最坏的可能，有3096－3996＝100个坏块。不过，事实上，现在出厂时的坏块，比较少，绝大多数，都是使用时间长了，在使用过程中出现的。

2.保证第一个块是好的，并且一般相对来说比较耐用。做此保证的主要原因是，很多Nand Flash坏块管理方法中，就是将第一个块，用来存储上面提到的BBT，否则，都是出错几率一样的块，那么也就不太好管理了，连放BBT的地方，都不好找了，^\_^。

一般来说，不同型号的Nand Flash的数据手册中，也会提到，自己的这个nand flash，最多允许多少个坏块。就比如上面提到的，三星的K9G8G08U0M，最多有100个坏块。

对于坏块的标记，本质上，也只是对应的flash上的某些字节的数据是非0xFF而已，所以，只要是数据，就是可以读取和写入的。也就意味着，可以写入其他值，也就把这个坏块标记信息破坏了。对于出厂时的坏块，一般是不建议将标记好的信息擦除掉的。

uboot中有个命令是“nand scrub”就可以将块中所有的内容都擦除了，包括坏块标记，不论是出厂时的，还是后来使用过程中出现而新标记的。一般来说，不建议用这个。不过，我倒是经常用，其实也没啥大碍，呵呵。

最好用“nand erase”只擦除好的块，对于已经标记坏块的块，不擦除。

【nand Flash中页的访问顺序】

在一个块内，对每一个页进行编程的话，必须是顺序的，而不能是随机的。比如，一个块中有128个页，那么你只能先对page0编程，再对page1编程，。。。。，而不能随机的，比如先对page3，再page1，page2.，page0，page4，.。。。

**【片选无关(CE don’t-care)技术】**

很多Nand flash支持一个叫做CE don’t-care的技术，字面意思就是，不关心是否片选，

那有人会问了，如果不片选，那还能对其操作吗？答案就是，这个技术，主要用在当时是不需要选中芯片却还可以继续操作的这些情况：在某些应用，比如录音，音频播放等应用，中，外部使用的微秒（us）级的时钟周期，此处假设是比较少的2us，在进行读取一页或者对页编程时，是对Nand Flash操作，这样的串行（Serial Access）访问的周期都是20/30/50ns，都是纳秒（ns）级的，此处假设是50ns，当你已经发了对应的读或写的命令之后，接下来只是需要Nand Flash内部去自己操作，将数据读取除了或写入进去到内部的数据寄存器中而已，此处，如果可以把片选取消，CE#是低电平有效，取消片选就是拉高电平，这样会在下一个外部命令发送过来之前，即微秒量级的时间里面，即2us－50ns≈2us，这段时间的取消片选，可以降低很少的系统功耗，但是多次的操作，就可以在很大程度上降低整体的功耗了。

总结起来简单解释就是：由于某些外部应用的频率比较低，而Nand Flash内部操作速度比较快，所以具体读写操作的大部分时间里面，都是在等待外部命令的输入，同时却选中芯片，产生了多余的功耗，此“不关心片选”技术，就是在Nand Flash的内部的相对快速的操作（读或写）完成之后，就取消片选，以节省系统功耗。待下次外部命令/数据/地址输入来的时候，再选中芯片，即可正常继续操作了。这样，整体上，就可以大大降低系统功耗了。

注:Nand Flash的片选与否，功耗差别会有很大。如果数据没有记错的话，我之前遇到我们系统里面的nand flash的片选，大概有5个mA的电流输出呢，要知道，整个系统优化之后的待机功耗，也才10个mA左右的。

**【带EDC的拷回操作以及Sector的定义（Copy-Back Operation with EDC & Sector Definition for EDC）】**

Copy-Back功能，简单的说就是，将一个页的数据，拷贝到另一个页。

如果没有Copy-Back功能，那么正常的做法就是，先要将那个页的数据拷贝出来放到内存的数据buffer中，读出来之后，再用写命令将这页的数据，写到新的页里面。

而**Copy-Back功能**的好处在于，不需要用到外部的存储空间，不需要读出来放到外部的buffer里面，而是可以直接读取数据到内部的页寄存器（page register）然后写到新的页里面去。而且，为了保证数据的正确，要硬件支持EDC（Error Detection Code）的，否则，在数据的拷贝过程中，可能会出现错误，并且拷贝次数多了，可能会累积更多错误。

而对于错误检测来说，硬件一般支持的是512字节数据，对应有16字节用来存放校验产生的ECC数值，而这512字节一般叫做一个扇区。对于2K＋64字节大小的页来说，按照512字节分，分别叫做A，B，C，D区，而后面的64字节的oob区域，按照16字节一个区，分别叫做E，F，G，H区，对应存放A，B，C，D数据区的ECC的值。

总结：

512+16

2K +64 ： A B C D - E F G H区

Copy-Back编程的主要作用在于，去掉了数据串行读取出来，再串行写入进去的时间，所以，这部分操作，是比较耗时的，所以此技术可以提高编程效率，提高系统整体性能。

**【多片同时编程(Simultaneously Program Multi Plane)】**

对于有些新出的Nand Flash，支持同时对多个片进行编程，比如上面提到的三星的K9K8G08U0A，内部包含4片(Plane)，分别叫做Plane0，Plane1，Plane2，Plane3。.由于硬件上，对于每一个Plane，都有对应的大小是2048+64=2112字节的页寄存器（Page Register），使得同时支持多个Plane编程成为可能。K9K8G08U0A支持同时对2个Plane进行编程。不过要注意的是，只能对Plane0和Plane1或者Plane2和Plane3，同时编程，而不支持Plane0和Plane2同时编程。

【交错页编程（Interleave Page Program）】

多片同时编程，是针对一个chip里面的多个Plane来说的，

而此处的交错页编程，是指**对多个chip而言**的。

可以先对一个chip，假设叫chip1，里面的一页进行编程，然后此时，chip1内部就开始将数据一点点写到页里面，就出于忙的状态了，而此时可以利用这个时间，对出于就绪状态的chip2，也进行页编程，发送对应的命令后，chip2内部也就开始慢慢的写数据到存储单元里面去了，也出于忙的状态了。此时，再去检查chip1，如果编程完成了，就可以开始下一页的编程了，然后发完命令后，就让其内部慢慢的编程吧，再去检查chip2，如果也是编程完了，也就可以进行接下来的其他页的编程了。如此，交互操作chip1和chip2，就可以有效地利用时间，使得整体编程效率提高近2倍，大大提高nand flash的编程/擦写速度了。

【随机输出页内数据（Random Data Output In a Page）】

在介绍此特性之前，先要说说，与Random Data Output In a Page相对应的是，普通的，正常的，sequential data output in a page。

正常情况下，我们读取数据，都是先发读命令，然后等待数据从存储单元到内部的页数据寄存器中后，我们通过不断地将RE#(Read Enale，低电平有效)置低，然后从我们开始传入的列的起始地址，一点点读出我们要的数据，直到页的末尾，当然有可能还没到页地址的末尾，就不再读了。所谓的顺序（sequential）读取也就是，根据你之前发送的列地址的起始地址开始，每读一个字节的数据出来，内部的数据指针就加1，移到下个字节的地址，然后你再读下一个字节数据，就可以读出来你要的数据了，直到读取全部的数据出来为止。

而此处的随机（random）读取，就是在你正常的顺序读取的过程中，

先发一个随机读取的开始命令0x05命令，

再传入你要将内部那个数据指针定位到具体什么地址，也就是2个cycle的列地址，

然后再发随机读取结束命令0xE0，

然后，内部那个数据地址指针，就会移动到你所制定的位置了，

你接下来再读取的数据，就是从那个制定地址开始的数据了。

而nand flash数据手册里面也说了，这样的随机读取，你可以多次操作，没限制的。

请注意，上面你所传入的地址，都是列地址，也就是页内地址，也就是说，对于页大小为2K的nand flash来说，所传入的地址，应该是小于2048+64＝2112的。

不过，实际在nand flash的使用中，好像这种用法很少的。绝大多数，都是顺序读取数据。

【页编程（写操作）】

Nand flash的写操作叫做编程Program，编程，一般情况下，是以页为单位的。

有的Nand Flash，比如K9K8G08U0A，支持部分页编程，但是有一些限制：在同一个页内的，连续的部分页的编程，不能超过4次。一般情况下，很少使用到部分页编程，都是以页为单位进行编程操作的。

一个操作，用两个命令去实现，看起来是多余，效率不高，但是实际上，有其特殊考虑，

至少对于块擦除来说，开始的命令0x60是擦除设置命令(erase setup comman)，然后传入要擦除的块地址，然后再传入擦除确认命令（erase confirm command）0xD0，以开始擦除的操作。

这种，分两步：开始设置，最后确认的命令方式，是为了避免由于外部由于无意的/未预料而产生的噪音，比如，，此时，即使被nand flash误认为是擦除操作，但是没有之后的确认操作0xD0，nand flash就不会去擦除数据，这样使得数据更安全，不会由于噪音而误操作。

分类: Flash驱动

【读（read）操作过程详解】

以最简单的read操作为例，解释如何理解时序图，以及将时序图中的要求，转化为代码。

解释时序图之前，让我们先要搞清楚，我们要做的事情：那就是，要从nand flash的某个页里面，读取我们要的数据。

要实现此功能，会涉及到几部分的知识，至少很容易想到的就是：需要用到哪些命令，怎么发这些命令，怎么计算所需要的地址，怎么读取我们要的数据等等。

下面，就一步步的解释，需要做什么，以及如何去做：

1.需要使用何种命令

首先，是要了解，对于读取数据，要用什么命令。

下面是datasheet中的命令集合：

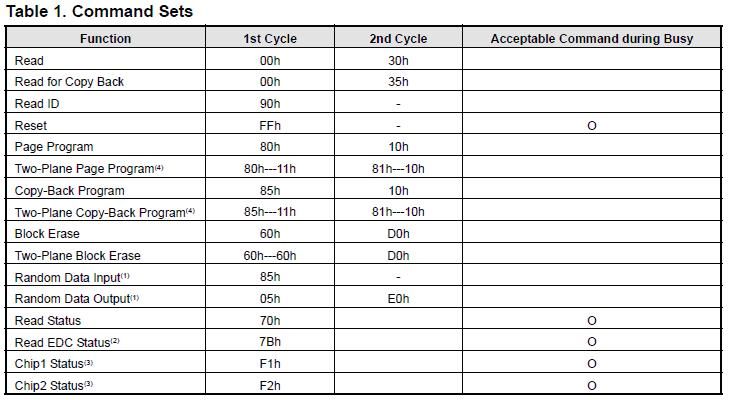


图5.Nand Flash K9K8G08U0A的命令集合

很容易看出，我们要读取数据，要用到Read命令，该命令需要2个周期，第一个周期发0x00，第二个周期发0x30。

2.发送命令前的准备工作以及时序图各个信号的具体含义

知道了用何命令后，再去了解如何发送这些命令。

[小常识]

在开始解释前，多罗嗦一下”使能”这个词，以便有些读者和我以前一样，在听这类虽然对于某些专业人士说是属于最基本的词汇了，但是对于初次接触，或者接触不多的人来说，听多了，容易被搞得一头雾水：使能（Enable），是指使其（某个信号）有效，使其生效的意思，“使其”“能够”怎么怎么样。。。。比如，上面图中的CLE线号，是高电平有效，如果此时将其设为高电平，我们就叫做，将CLE使能，也就是使其生效的意思。

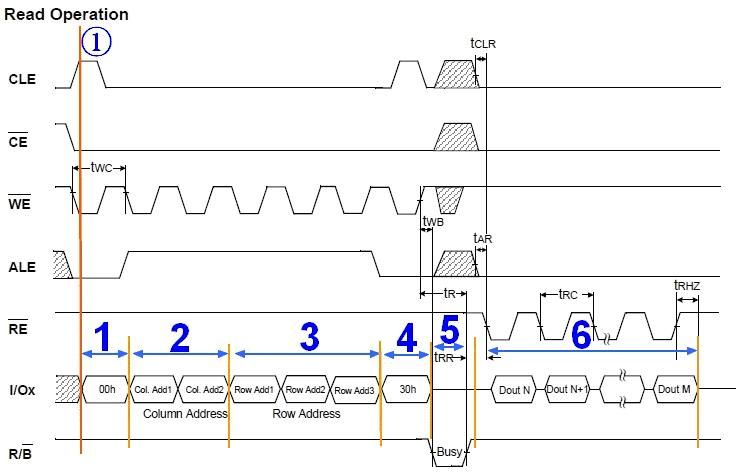


图6.Nand Flash数据读取操作的时序图

注：此图来自三星的型号K9K8G08U0A的nand flash的数据手册(datasheet)。

我们来一起看看，我在图6中的特意标注的①边上的黄色竖线。

黄色竖线所处的时刻，是在发送读操作的第一个周期的命令0x00之前的那一刻。

让我们看看，在那一刻，其所穿过好几行都对应什么值，以及进一步理解，为何要那个值。

**（1）**黄色竖线穿过的第一行，是CLE。还记得前面介绍命令所存使能（CLE）那个引脚吧？CLE，将CLE置1，就说明你将要通过I/O复用端口发送进入Nand Flash的，是命令，而不是地址或者其他类型的数据。只有这样将CLE置1，使其有效，才能去通知了内部硬件逻辑，你接下来将收到的是命令，内部硬件逻辑，才会将受到的命令，放到命令寄存器中，才能实现后面正确的操作，否则，不去将CLE置1使其有效，硬件会无所适从，不知道你传入的到底是数据还是命令了。

**（2）**而第二行，是CE#，那一刻的值是0。这个道理很简单，你既然要向Nand Flash发命令，那么先要选中它，所以，要保证CE#为低电平，使其有效，也就是片选有效。

**（3）**第三行是WE#，意思是写使能。因为接下来是往nand Flash里面写命令，所以，要使得WE#有效，所以设为低电平。

**（4）**第四行，是ALE是低电平，而ALE是高电平有效，此时意思就是使其无效。而对应地，前面介绍的，使CLE有效，因为将要数据的是命令，而不是地址。如果在其他某些场合，比如接下来的要输入地址的时候，就要使其有效，而使CLE无效了。

**（5）**第五行，RE#，此时是高电平，无效。可以看到，知道后面低6阶段，才变成低电平，才有效，因为那时候，要发生读取命令，去读取数据。

**（6）**第六行，就是我们重点要介绍的，复用的输入输出I/O端口了，此刻，还没有输入数据，接下来，在不同的阶段，会输入或输出不同的数据/地址。

**（7）**第七行，R/B#,高电平，表示R（Ready）/就绪，因为到了后面的第5阶段，硬件内部，在第四阶段，接受了外界的读取命令后，把该页的数据一点点送到页寄存器中，这段时间，属于系统在忙着干活，属于忙的阶段，所以，R/B#才变成低，表示Busy忙的状态的。

介绍了时刻①的各个信号的值，以及为何是这个值之后，相信，后面的各个时刻，对应的不同信号的各个值，大家就会自己慢慢分析了，也就容易理解具体的操作顺序和原理了。

3.如何计算出，我们要传入的地址

在介绍具体读取数据的详细流程之前，还要做一件事，那就是，先要搞懂我们要访问的地址，以及这些地址，如何分解后，一点点传入进去，使得硬件能识别才行。

此处还是以K9K8G08U0A为例，此nand flash，一共有8192个块，每个块内有64页，每个页是2K+64 Bytes，假设，我们要访问其中的第7000个块中的第25页中的1208字节处的地址，此时，我们就要先把具体的地址算出来：

物理地址=块大小×块号+页大小×页号+页内地址=7000×128K+64×2K+1208=0x36B204B8,接下来，我们就看看，怎么才能把这个实际的物理地址，转化为nand Flash所要求的格式。

在解释地址组成之前，先要来看看其datasheet中关于地址周期的介绍：

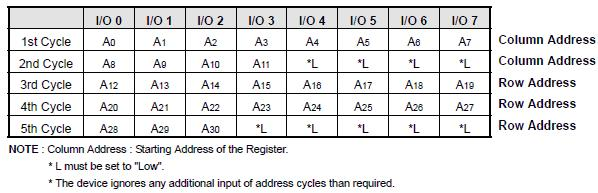


图7 Nand Flash的地址周期组成

结合图7和图5中的2，3阶段，我们可以看出，此nand flash地址周期共有5个，2个列(Column)周期，3个行（Row）周期。

而对于对应地，我们可以看出，实际上，

列地址A0~A10，就是页内地址，地址范围是从0到2047，而对出的A11，理论上可以表示2048～4095，但是实际上，我们最多也只用到了2048～2112，用于表示页内的oob区域，其大小是64字节。

A12～A30，称作页号，页的号码，可以定位到具体是哪一个页。

而其中，A18～A30，表示对应的块号，即属于哪个块。

// 可见：地址的传输顺序是是 页内地址，页号，块号。 从小到大。

简单解释完了地址组成，那么就很容易分析上面例子中的地址了：

0x36B204B8 = 0011 0110 1011 0010 0000 0100 1011 1000，分别分配到5个地址周期就是：

1st 周期，A7～A0 ：1011 1000 = 0x B8

2nd周期，A11～A8 ：0000 0100 = 0x04

3rd周期，A19～A12 ：0010 0000 = 0x20

4th周期，A27～A20 ：0110 1011 = 0x6B

5th周期，A30～A28 ：0000 0011 = 0x03

注意，与图7中对应的，\*L，意思是低电平，由于未用到那些位，datasheet中强制要求设为0，所以，才有上面的2nd周期中的高4位是0000.其他的A30之后的位也是类似原理，都是0。

因此，接下来要介绍的，我们要访问第7000个块中的第25页中的1208字节处的话，所要传入的地址就是分5个周期，分别传入两个列地址的：0xB8，0x04，然后再传3个行地址的：0x20，0x6B，0x03，这样硬件才能识别。

4.读操作过程的解释

准备工作终于完了，下面就可以开始解释说明，对于读操作的，上面图中标出来的，1-6个阶段，具体是什么含义。

（1） 操作准备阶段：此处是读（Read）操作，所以，先发一个图5中读命令的第一个阶段的0x00,表示，让硬件先准备一下，接下来的操作是读。

（2） 发送两个周期的列地址。也就是页内地址，表示，我要从一个页的什么位置开始读取数据。

（3） 接下来再传入三个行地址。对应的也就是页号。

（4） 然后再发一个读操作的第二个周期的命令0x30。接下来，就是硬件内部自己的事情了。

（5） Nand Flash内部硬件逻辑，负责去按照你的要求，根据传入的地址，找到哪个块中的哪个页，然后把整个这一页的数据，都一点点搬运到页缓存中去。而在此期间，你所能做的事，也就只需要去读取状态寄存器，看看对应的位的值，也就是R/B#那一位，是1还是0，0的话，就表示，系统是busy，仍在”忙“（着读取数据），如果是1，就说系统活干完了，忙清了，已经把整个页的数据都搬运到页缓存里去了，你可以接下来读取你要的数据了。

对于这里。估计有人会问了，这一个页一共2048+64字节，如果我传入的页内地址，就像上面给的1028一类的值，只是想读取1028到2011这部分数据，而不是页开始的0地址整个页的数据，那么内部硬件却读取整个页的数据出来，岂不是很浪费吗？答案是，的确很浪费，效率看起来不高，但是实际就是这么做的，而且本身读取整个页的数据，相对时间并不长，而且读出来之后，内部数据指针会定位到你刚才所制定的1208的那个位置。

（6） 接下来，就是你“窃取“系统忙了半天之后的劳动成果的时候了，呵呵。通过先去Nand Flash的控制器中的数据寄存器中写入你要读取多少个字节(byte)/字(word)，然后就可以去Nand Flash的控制器的FIFO中，一点点读取你要的数据了。

至此，整个Nand Flash的读操作就完成了。

对于其他操作，可以根据我上面的分析，一点点自己去看datasheet，根据里面的时序图去分析具体的操作过程，然后对照代码，会更加清楚具体是如何实现的。

**【Flash的类型】**

Flash的类型主要分两种，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

再多说几句：

1.nor的成本相对高，具体读写数据时候，不容易出错。总体上，比较适合应用于存储少量的代码。

2.Nand flash相对成本低。使用中数据读写容易出错，所以一般都需要有对应的软件或者硬件的数据校验算法，统称为ECC。由于相对来说，容量大，价格便宜，因此适合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。

所以，一个常见的应用组合就是，用小容量的Nor Flash存储启动代码，比如uboot，系统启动后,初始化对应的硬件，包括SDRAM等，然后将Nand Flash上的Linux 内核读取到内存中，做好该做的事情后，就跳转到SDRAM中去执行内核了，然后内核解压（如果是压缩内核的话，否则就直接运行了）后，开始运行，在Linux内核启动最后，去Nand Flash上，挂载根文件，比如jffs2，yaffs2等，挂载完成，运行初始化脚本，启动consle交互，才运行你通过console和内核交互。至此完成整个系统启动过程。

而Nor Flash存放的是Uboot，Nand Flash存放的是Linux的内核镜像和根文件系统，以及余下的空间分成一个数据区。

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：



表3 Nand Flash 和 Nor Flash的区别

1. 理论上是可以的，而且也是有人验证过可以的，只不过由于nand flash的物理特性，不能完全保证所读取的数据/代码是正确的，实际上，很少这么用而已。因为，如果真是要用到nand flash做XIP，那么除了读出速度慢之外，还要保证有数据的校验，以保证读出来的，将要执行的代码/数据，是正确的。否则，系统很容易就跑飞了。。。

2. 芯片内执行(XIP, eXecute In Place):

<http://hi.baidu.com/serial_story/blog/item/adb20a2a3f8ffe3c5243c1df.html>

【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。

【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，yaffs2文件系统，会在oob中，存放很多和自己文件系统相关的信息。

**【内存技术设备，MTD（Memory Technology Device）】**

MTD，是Linux的存储设备中的一个子系统。其设计此系统的目的是，对于内存类的设备，提供一个抽象层，一个接口，使得对于硬件驱动设计者来说，可以尽量少的去关心存储格式，比如FTL，FFS2等，而只需要去提供最简单的底层硬件设备的读/写/擦除函数就可以了。而对于数据对于上层使用者来说是如何表示的，硬件驱动设计者可以不关心，而MTD存储设备子系统都帮你做好了。

对于MTD字系统的好处，简单解释就是，他帮助你实现了，很多对于以前或者其他系统来说，本来也是你驱动设计者要去实现的很多功能。换句话说，有了MTD，使得你设计Nand Flash的驱动，所要做的事情，要少很多很多，因为大部分工作，都由MTD帮你做好了。

当然，这个好处的一个“副作用”就是，使得我们不了解的人去理解整个Linux驱动架构，以及MTD，变得更加复杂。但是，总的说，觉得是利远远大于弊，否则，就不仅需要你理解，而且还是做更多的工作，实现更多的功能了。

此外，还有一个重要的原因，那就是，前面提到的nand flash和普通硬盘等设备的特殊性：

有限的通过出复用来实现输入输出命令和地址/数据等的IO接口，最小单位是页而不是常见的bit，写前需擦除等，导致了这类设备，不能像平常对待硬盘等操作一样去操作，只能采取一些特殊方法，这就诞生了MTD设备的统一抽象层。

MTD，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。具体的内部硬件设备的读/写/擦除函数，那就是你必须实现的了。

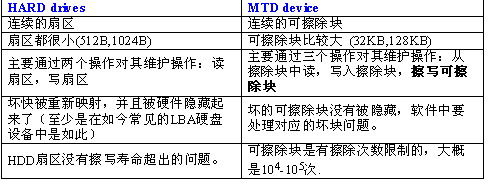


表4.MTD设备和硬盘设备之间的区别

==========  
多说一句，关于MTD更多的内容，感兴趣的，去附录中的MTD的主页去看。

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](http://www.cnitblog.com/luofuchong/archive/2007/08/31/32682.html)

[MTD原始设备与FLASH硬件驱动的对话-续](http://www.cnitblog.com/luofuchong/archive/2007/09/04/32939.html)

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。

**【Nand flash驱动工作原理】**

在介绍具体如何写Nand Flash驱动之前，我们先要了解，大概的，整个系统，和Nand Flash相关的部分的驱动工作流程，这样，对于后面的驱动实现，才能更加清楚机制，才更容易实现，否则就是，即使写完了代码，也还是没搞懂系统是如何工作的了。

让我们以最常见的，Linux内核中已经有的三星的Nand Flash驱动，来解释Nand Flash驱动具体流程和原理。

此处是参考2.6.29版本的Linux源码中的\drivers\mtd\nand\s3c2410.c，以2410为例。

1. 在nand flash驱动加载后，第一步，调用对应的init函数 ---- s3c2410\_nand\_init: 去将nand flash驱动注册到Linux驱动框架中。

2. 驱动本身真正的开始，是从probe函数： s3c2410\_nand\_probe->s3c24xx\_nand\_probe,  
在probe过程中：  
clk\_enable //打开nand flash控制器的clock时钟，  
request\_mem\_region //去申请驱动所需要的一些内存等相关资源。  
s3c2410\_nand\_inithw //去初始化硬件相关的部分，主要是关于时钟频率的计算，以及启用nand flash控制器，使得硬件初始化好了，后面才能正常工作。

3. 需要多解释一下的，是这部分代码：

for (setno = 0; setno < nr\_sets; setno++, nmtd++) {  
pr\_debug("initialising set %d (%p, info %p)\n", setno, nmtd, info);

/\*调用init chip去挂载你的nand驱动的底层函数到"nand flash的结构体"中，以及设置对应的"ecc mode"，挂载ecc相关的函数 \*/  
s3c2410\_nand\_init\_chip(info, nmtd, sets);

/\* scan\_ident，扫描nand 设备，设置nand flash的默认函数，获得物理设备的具体型号以及对应各个特性参数，这部分算出来的一些值，对于nand flash来说，是最主要的参数，比如nand flash的芯片的大小，块大小，页大小等。 \*/  
nmtd->scan\_res = nand\_scan\_ident(&nmtd->mtd, (sets) ? sets->nr\_chips : 1);

if (nmtd->scan\_res == 0) {  
s3c2410\_nand\_update\_chip(info, nmtd);

/\*扫描的后一阶段，经过前面的scan\_ident，我们已经获得对应nand flash的硬件的各个参数，  
\*然后就可以在scan tail中，根据这些参数，去设置其他一些重要参数，尤其是ecc的layout，即ecc是如何在oob中摆放的，  
\*最后，再去进行一些初始化操作，主要是根据你的驱动，如果没有实现一些函数的话，那么就用系统默认的。 \*/  
nand\_scan\_tail(&nmtd->mtd);

/\*add partion，根据你的nand flash的分区设置，去分区 \*/  
s3c2410\_nand\_add\_partition(info, nmtd, sets);  
}  
if (sets != NULL)  
sets++;  
}

4. 等所有的参数都计算好了，函数都挂载完毕，系统就可以正常工作了。

上层访问你的nand falsh中的数据的时候，通过MTD层，一层层调用，最后调用到你所实现的那些底层访问硬件数据/缓存的函数中。

**【Linux下nand flash驱动编写步骤简介】**

关于上面提到的，在nand\_scan\_tail的时候，系统会根据你的驱动，如果没有实现一些函数的话，那么就用系统默认的。如果实现了自己的函数，就用你的。  
"那么到底我要实现哪些函数呢，而又有哪些是可以不实现，用系统默认的就可以了呢。"  
此问题的，就是我们下面要介绍的，也就是，你要实现的，你的驱动最少要做哪些工作，才能使整个nand flash工作起来。

**1.**对于驱动框架部分

其实，要了解，关于驱动框架部分，你所要做的事情的话，只要看看三星的整个nand flash驱动中的这个结构体，就差不多了：

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,  
},  
};

对于上面这个结构体，没多少要解释的。从名字，就能看出来：  
（1）probe就是系统“探测”，就是前面解释的整个过程，这个过程中的多数步骤，都是和你自己的nand flash相关的，尤其是那些硬件初始化部分，是你必须要自己实现的。  
（2）remove，就是和probe对应的，“反初始化”相关的动作。主要是释放系统相关资源和关闭硬件的时钟等常见操作了。  
（3）suspend和resume，对于很多没用到电源管理的情况下，至少对于我们刚开始写基本的驱动的时候，可以不用关心，放个空函数即可。

**2.** 对于nand flash底层操作实现部分

而对于底层硬件操作的有些函数，总体上说，都可以在上面提到的s3c2410\_nand\_init\_chip中找到：

static void **s3c2410\_nand\_init\_chip**(struct s3c2410\_nand\_info \*info,  
struct s3c2410\_nand\_mtd \*nmtd, //主要是完善该结构体  
struct s3c2410\_nand\_set \*set)  
{

struct nand\_chip \*chip = &nmtd->chip;  
void \_\_iomem \*regs = info->regs;

chip->write\_buf = s3c2410\_nand\_write\_buf;  
chip->read\_buf = s3c2410\_nand\_read\_buf;  
chip->select\_chip = s3c2410\_nand\_select\_chip;  
chip->chip\_delay = 50;  
chip->priv = nmtd;  
chip->options = 0;  
chip->controller = &info->controller;

switch (info->cpu\_type) {  
case TYPE\_S3C2410:

/\* nand flash控制器中，一般都有对应的数据寄存器，用于给你往里面写数据，表示将要读取或写入多少个字节(byte,u8)/字(word,u32) ，所以，此处，你要给出地址，以便后面的操作所使用 \*/

chip->IO\_ADDR\_W = regs + S3C2410\_NFDATA;  
info->sel\_reg = regs + S3C2410\_NFCONF;  
info->sel\_bit = S3C2410\_NFCONF\_nFCE;  
chip->cmd\_ctrl = s3c2410\_nand\_hwcontrol;  
chip->dev\_ready = s3c2410\_nand\_devready;

break;

。。。。。。

}

chip->IO\_ADDR\_R = chip->IO\_ADDR\_W;

nmtd->info = info;  
nmtd->mtd.priv = chip;  
nmtd->mtd.owner = THIS\_MODULE;  
nmtd->set = set;

if (hardware\_ecc) {  
chip->ecc.calculate = s3c2410\_nand\_calculate\_ecc;  
chip->ecc.correct = s3c2410\_nand\_correct\_data;

/\* 此处，多数情况下，你所用的Nand Flash的控制器，都是支持硬件ECC的，所以，此处设置硬件ECC(HW\_ECC) ，也是充分利用硬件的特性，  
\* 而如果此处不用硬件去做ECC话，那么下面也会去设置成NAND\_ECC\_SOFT，系统会用默认的软件去做ECC校验，相比之下，比硬件ECC的效率就低很多，而你的nand flash的读写，也会相应地要慢不少 \*/

chip->ecc.mode = NAND\_ECC\_HW; //设置成了硬件方式校验ecc

switch (info->cpu\_type) {

case TYPE\_S3C2410:

chip->ecc.hwctl = s3c2410\_nand\_enable\_hwecc;  
chip->ecc.calculate = s3c2410\_nand\_calculate\_ecc;

break;  
。。。。。  
}

} else {  
chip->ecc.mode = NAND\_ECC\_SOFT; //也就是说，怎么搞也得校验了  
}

if (set->ecc\_layout != NULL)  
chip->ecc.layout = set->ecc\_layout;

if (set->disable\_ecc)  
chip->ecc.mode = NAND\_ECC\_NONE;  
}

而我们要实现的底层函数，也就是上面蓝色标出来的一些函数而已：

（1）**s3c2410\_nand\_write\_buf**和 **s3c2410\_nand\_read\_buf**：这是两个最基本的操作函数，其功能，就是往你的nand flash的控制器中的FIFO读写数据。一般情况下，是MTD上层的操作，比如要读取一页的数据，那么在发送完相关的读命令和等待时间之后，就会调用到你底层的read\_buf，去nand Flash的FIFO中，一点点把我们要的数据，读取出来，放到我们制定的内存的缓存中去。写操作也是类似，将我们内存中的数据，写到Nand Flash的FIFO中去。  
（2）**s3c2410\_nand\_select\_chip** ： 实现Nand Flash的片选。  
（3）**s3c2410\_nand\_hwcontrol**： 给底层发送命令或地址，或者设置具体操作的模式，都是通过此函数。  
（4）**s3c2410\_nand\_devready**： Nand Flash的一些操作，比如读一页数据，写入（编程）一页数据，擦除一个块，都是需要一定时间的，在命令发送完成后，就是硬件开始忙着工作的时候了，而硬件什么时候完成这些操作，什么时候不忙了，变就绪了，就是通过这个函数去检查状态的。一般具体实现都是去读硬件的一个状态寄存器，其中某一位是否是1，对应着是出于“就绪/不忙”还是“忙”的状态。这个寄存器，也就是我们前面分析时序图中的R/B#。  
（5）**s3c2410\_nand\_calculate\_ecc**：如果是上面提到的硬件ECC的话，就不用我们用软件去实现校验算法了，而是直接去读取硬件产生的ECC数值就可以了。  
（6）**s3c2410\_nand\_correct\_data**： 当实际操作过程中，读取出来的数据所对应的硬件或软件计算出来的ECC，和从oob中读出来的ECC不一样的时候，就是说明数据有误了，就需要调用此函数去纠正错误。对于现在SLC常见的ECC算法来说，可以发现2位，纠正1位。如果错误大于1位，那么就无法纠正回来了。一般情况下，出错超过1位的，好像几率不大。至少我看到的不是很大。更复杂的情况和更加注重数据安全的情况下，一般是需要另外实现更高效和检错和纠错能力更强的ECC算法的。  
（7）**s3c2410\_nand\_enable\_hwecc**： 在硬件支持的前提下，前面设置了硬件ECC的话，要实现这个函数，用于每次在读写操作前，通过设置对应的硬件寄存器的某些位，使得启用硬件ECC，这样在读写操作完成后，就可以去读取硬件校验产生出来的ECC数值了。

当然，除了这些你必须实现的函数之外，在你更加熟悉整个框架之后，你可以根据你自己的nand flash的特点，去实现其他一些原先用系统默认但是效率不高的函数，而用自己的更高效率的函数替代他们，以提升你的nand flash的整体性能和效率。