【详解】如何编写 Linux 下 Nand Flash 驱动

版本: 1.7

作者：crifan

邮箱: green-waste(At)163.com

版本历史

版本	日期	内容说明
1.0	2009-07-21	简介如何在 Linux 下实现 Nand Flash 驱动
1.2	2011-03-15	整理了排版 添加了很多内容
1.3	2011-06-12	修正了 Nand Flash 行列地址的计算方法
1.7	2011-07-02	添加了 ONFI，LBA 规范的介绍 添加了 Unique ID 介绍 添加了对应的 MTD 中检测不同类型芯片的代码 增加了关于 Nand Flash 的软件和硬件的 ECC 算法的简介

目录

图表

缩略词

缩写	全称
BBM	Bad Block Management
BBT	Bad Block Table
EEPROM	Electrically Erasable Programmable Read-Only Memory
MLC	Multi Level Cell
MOSFET	Metal-Oxide -Semiconductor Field Effect Transistor 金属氧化物半导体场效应晶体管
SLC	Single Level Cell

1 正文之前

1.1 目的

本文的主要目的是，看了之后，你应该对 Nand Flash 的硬件特性以及对应的 Linux 下软件平台有了基本的认识，进一步地，对如何实现 Linux 下的 Nand Flash 的驱动，知道要做哪些事情了，以及大概是如何实现的。这样，如果有了对应的开发环境，你就可以自己去实现 Nand Flash 的驱动了。
不过额外提示一句的是，写出代码，并不代表你就完全搞懂了整个系统的流程。而且已经写好的代码，很可能有 bug，要你不断地调试，通过调试，你才会对整个系统以及 Nand Flash 的方方面面有个更深入的了解的。
而且，你会发现，为了写驱动那点代码之前，却要弄懂太多的东西，包括硬件的工作原理，软件的协议规范，软件的逻辑架构等等，最后才能去实现你的驱动，所以有人会说，你写驱动不是很简单嘛，不就是写那几行代码吗，对此，一个经典的回答就是，对于整个写驱动的工作的价值算作 100 元的话，写代码值 1 块钱，但是知道怎么写，值 99 块钱。^_^

1.2 目标读者和阅读此文的前提

正因为此文目的是让你搞懂如何在 Linux 下面实现 Nand Flash 的驱动，所以，目标读者就是，希望对 Nand Flash 硬件知识有一定了解，和想要在 Linux 下面实现 Nand Flash 驱动的读者。而阅读此文的前提，是要有一些基本的软硬件基础知识，和了解如何在 v2.6 内核之后 Linux 的下面开发驱动的流程。有了这些知识，再看本文，然后你才能清楚真正要去实现 Nand Flash 的驱动，是如何下手。

1.3 说明

本文的逻辑是，先介绍 Nand Flash 的一些基本的硬件知识，然后详细分析 Nand Flash 的 Read 操作的具体的流程，清楚硬件实现的逻辑，接着介绍软件平台，即 Linux 下面和 Nand Flash 相关的内容，这样，硬件和软件都清楚是怎么回事了，然后再介绍如何去在 Linux 的架构下，实现 Nand Flash 驱动。

之前写的版本，虽然前面关于 Nand Flash 的内容介绍的比较详细，但是后面关于相关的 MTD 知识，尤其是 Linux 的 MTD 的架构和如何实现具体的 Nand 的 Flash 的操作等部分的内容，写的很简略，导致有些读者看了后，觉得是，关于如何写驱动，和没说差不多，呵呵。
因此，现在继续更新，将更详细的解释，如何从硬件 Nand Flash 的规范，一步步映射到具体的软件实现的过程，这样，使得读者更明白其中的内在逻辑，然后接着再介绍如何在理解了软硬件各自的所具有的功能，以及 Linux 的 MTD 系统，已经帮你实现了哪些功能，然后才会更加明白，余下的要实现的软件部分，就是你所要实现的 Linux 下的 Nand Flash 的驱动部分了。

1.4 声明

2 编写驱动之前要了解的知识

2.1 一些相关的名词的解释

2.1.1 Non-Volatile Memory 非易失性存储器

NV （RAM）Memory，断电数据也不会丢失的存储器，比如 Nand Flash，Nor Flash，硬盘等等。于此相对的是，断电了数据会丢失的存储器，比如 DRAM 等。

2.1.2 OTP（One Time Programmable）一次性可编程存储器

OTP，一种非易失性存储器，但是只允许一次性写入数据，写入（或称烧写）数据之后，就不能修改了。
OTP 的好处或者说用途是，常用于写入一些和芯片相关的一些特定数据，用于加密的一些数据等。
与一次性写入数据的 OTP 相对应的是，像 Nand Flash，硬盘等存储器，可以被多次写入数据。只要硬盘这类的存储器没坏，你高兴写入几次就写入几次，而 OTP 就只能写入一次，就没法再修改里面的数据了。

2.1.3 NDA（None-Disclosure Agreement）

NDA，中文可以翻译为，非公开协议，保密协议。
说白了，还是一种协议，常用于这种情况：
某家厂商的某种技术或资料，是保密的，不希望公开的。
但是呢，如果你要用他家的芯片啊之类的东西，在开发过程中，又必须得到对应的技术和资料，才能开发产品，所以，他就会要求和你签订这样的 NDA 协议，意思就是，你可以用我的技术和资料，但是你不能公开给（我未授权的）其他人。如果非法泄露我的机密技术，那我肯定要走法律程序控告你，之类的。

2.1.4 Datasheet 数据手册和 Specification 规范

英文 datasheet，中文一般翻译为数据手册。
指的是对应某个硬件，多为芯片，的功能说明，定义了如何操作该硬件，达到你要的功能，这其中主要包括芯片中的相关寄存器的定义，如何发送命令，发送什么命令，以此来操作此硬件等等。

而英文 Specification，引文常缩写为 Spec.，中文一般翻译为规范。
多指某个组织（盈利的或非盈利的），定义了一些规矩，如果你要用某种东西，在计算机领域，常常指的是某硬件和相关的软件协议，就要按照此规矩来操作，人家这个组织呢，保证你只要实现了此规范，设备就能按照你所期望的运行，能够实现对应的功能，而你的芯片实现了此规范，就叫做，是和此规范兼容（compatible）的。

2.1.5 Nand Flash 相关的一些名词解释

2.1.5.1 （Bad） Block Management（坏）块管理

Nand Flash 由于其物理特性，只有有限的擦写次数，超过那个次数，基本上就是坏了。在使用过程中，有些 Nand Flash 的 block 会出现被用坏了，当发现了，要及时将此 block 标注为坏块，不再使用。
于此相关的管理工作，属于 Nand Flash 的坏块管理的一部分工作。

2.1.5.2 Wear-Levelling 负载平衡

Nand Flash 的 block 的管理，还包括负载平衡。
正是由于 Nand Flash 的 block，都是有一定寿命限制的，所以如果你每次都往同一个 block 擦除然后写入数据，那么那个 block 就很容易被用坏了，所以我们要去管理一下，将这么多次的对同一个 block 的操作，平均分布到其他一些 block 上面，使得在 block 的使用上，相对较平均，这样相对来说，可以更能充分利用 Nand Flash。
关于 wear-leveling 这个词，再简单解释一下，wear 就是穿（衣服）等，用（东西）导致磨损，而 leveling 就是使得均衡，所以放在一起就是，使得对于 Nand Flash 的那么多的 block 的使用磨损，相对均衡一些，以此延长 Nand Flash 的使用寿命或者说更加充分利用 Nand Flash。

2.1.5.3 ECC（Error Correction Code）错误校验（代码）

Nand Flash 物理特性上使得其数据读写过程中会发生一定几率的错误，所以要有个对应的错误检测和纠正的机制，于是才有此 ECC，用于数据错误的检测与纠正。Nand Flash 的 ECC，常见的算法有海明码和 BCH，这类算法的实现，可以是软件也可以是硬件。不同系统，根据自己的需求，采用对应的软件或者是硬件。
相对来说，硬件实现这类 ECC 算法，肯定要比软件速度要快，但是多加了对应的硬件部分，所以成本相对要高些。如果系统对于性能要求不是很高，那么可以采用软件实现这类 ECC 算法，但是由于增加了数据读取和写入前后要做的数据错误检测和纠错，所以性能相对要降低一些，即 Nand Flash 的读取和写入速度相对会有所影响。

其中，Linux 中的软件实现 ECC 算法，即 NAND_ECC_SOFT 模式，就是用的对应的海明码。而对于目前常见的 MLC 的 Nand Flash 来说，由于容量比较大，动辄 2GB，4GB，8GB 等，常用 BCH 算法。BCH 算法，相对来说，算法比较复杂。
笔者由于水平有限，目前仍未完全搞懂 BCH 算法的原理。
BCH 算法，通常是由对应的 Nand Flash 的 Controller 中，包含对应的硬件 BCH ECC 模块，实现了 BCH 算法，而作为软件方面，需要在读取数据后，写入数据之前，分别操作对应 BCH 相关的寄存器，设置成 BCH 模式，然后读取对应的 BCH 状态寄存器，得知是否有错误，和生成的 BCH 校验码，用于写入。
其具体代码是如何操作这些寄存器的，由于是和具体的硬件，具体的 nand flash 的 controller 不同而不同，无法用同一的代码。如果你是 nand flash 驱动开发者，自然会得到对应的起 nand flash 的 controller 部分的 datasheet，按照手册说明，去操作即可。
不过，额外说明一下的是，关于 BCH 算法，往往是要从专门的做软件算法的厂家购买的，但是 Micron 之前在网上放出一个免费版本的 BCH 算法。
想要下载此免费的 BCH 算法，可以在这里找到地址：
【下载】4 bits 纠错的 BCH 源代码（常用于 MLC nand Flash 的 ECC 算法）

http://hi.baidu.com/serial_story/blog/item/c4e826d8988aa13932fa1cc6.html

2.2 硬件特性

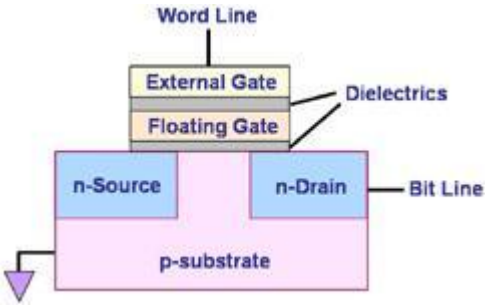
2.2.1 什么是 Flash

Flash 全名叫做 Flash Memory，从名字就能看出，是种数据存储设备，存储设备有很多类，Flash 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)。关于什么是非易失性/易失性，从名字中就可以看出，非易失性就是不容易丢失，数据存储在这类设备中，即使断电了，也不会丢失，这类设备，除了 Flash，还有其他比较常见的入硬盘，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。

2.2.1.1 Flash 的硬件实现机制

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

图表 1 典型的 Flash 内存单元的物理结构



数据在 Flash 内存单元中是以电荷(electrical charge) 形式存储的。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。而数据的表示，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，因此，Flash 的存储单元的默认值，不是 0（其他常见的存储设备，比如硬盘灯，默认值为 0），而是 1，而如果将电荷释放掉，电压降低到一定程度，表述数字 0。

2.2.2 什么是 Nand Flash

Flash 主要分两种，Nand Flash 和 nor flash。

关于 Nand Flash 和 Nor Flash 的区别，参见【附录 5】。

不过，关于两者区别，除了那个解释之外，这里再多解释解释：

1.Nor 的成本相对高，容量相对小，比如常见的只有 128KB，256KB，1MB，2MB 等等，优点是读写数据时候，不容易出错。所以在应用领域方面，Nor Flash 比较适合应用于存储少量的代码。

2.Nand flash 成本相对低，说白了就是便宜，缺点是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。但优点是，相对来说容量比较大，现在常见的 Nand Flash 都是 1GB，2GB，更大的 8GB 的都有了，相对来说，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，用大容量的 Nand Flash 做整个系统和用户数据的存储。
而一般的嵌入式平台的启动流程也就是，系统从装有启动代码的 Nor Flash 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 就分别存放的是 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。

2.2.2.1 Nand Flash 和 Nor Flash 的区别

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：

图表 2 Nand Flash 和 Nor Flash 的区别

	NOR	NAND	（备注）
接口	总线	I/O 接口	这个是两者物理结构上的最大区别
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（1）	即是否允许，芯片内执行 (XIP, eXecute In Place) (参见附录)
位反转(Bit twiddling/bit flip)	几乎无限制	1-4 次，也称作 “部分页编程限制”	也就是数据错误，0->1 或 1->0
在芯片出厂时候是否允许坏块	不允许	允许	

注：
（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。。。

2.2.2.2 Nand Flash 的详细分类

Nand Flash，按照硬件类型，可以分为

（1）**Bare NAND chips**：裸片，单独的 Nand Flash 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。

2.2.3 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。

2.2.3.1 SLC（Single Level Cell）

单个存储单元，只存储一位数据，表示 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，也就无法实现之前所说的的，Flash 的速度，即一闪而过的速度了，也就失去了 Flash 的众多特性了。

2.2.3.2 MLC（Multi Level Cell）

与 SLC 相对应的，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，关于这点，之前看 Nand flash 的数据手册（datasheet）的时候，差点搞晕了。

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。

2.2.3.3 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：

图表 3 Nand Flash 第 3 个 ID 的含义

	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				

2.2.4 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。
概念上，由大到小来说，就是：
Nand Flash -> Chip -> Plane -> Block -> Page -> oob

比如，型号为 K9K8G08U0A 这个芯片（chip），内部有两个 K9F4G08U0A，每个 K9F4G08U0A 包含了 2 个 Plane，每个 Plane 是 1Gb，所以 K9F4G08U0A 的大小是 1Gb×2＝2Gb＝256MB，因此，K9K8G08U0A 内部有 2 个 K9F4G08U0A，即 4 个 Plane，总大小是 4×256MB＝1GB。而型号是 K9WAG08U1A 的 Nand Flash，内部包含了 2 个 K9K8G08U0A，所以，总容量是 K9K8G08U0A 的两倍＝1GB×2＝2GB，类似地 K9NBG08U5A，内部包含了 4 个 K9K8G08U0A，总大小就是 4×1GB＝4GB。

【注意】
上面所说的 block，page 等 Nand Flash 的物理上的组织结构，是在 chip 的基础上来说的，但是软件编程的时候，除非你要用到 Multi Plane Program 和 Interleave Page Program 等，一般很少区分内部有几个 chip 以及每个 chip 有几个 plane，而最关心的只是 Nand Flash 的总体容

量 size 有多大，比如是 1GB 还是 2GB 等等。

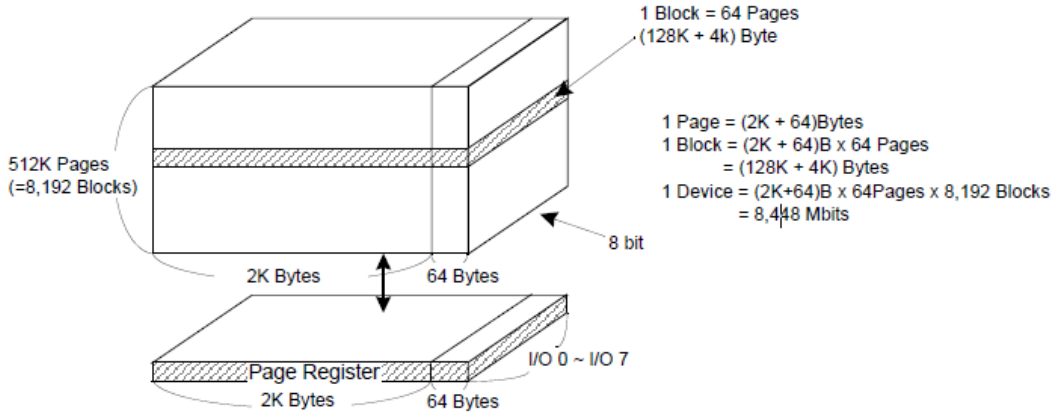
下面详细介绍一下，Nand Flash 的一个 chip 内部的硬件逻辑组织结构。

2.2.5 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：

图表 4 Nand Flash 物理存储单元的阵列组织结构

Figure 2. K9K8G08U0A Array Organization



上图是 K9K8G08U0A 的 datasheet 中的描述。
简单解释就是:

2.2.5.1 Block 块

一个 Nand Flash（的 chip，芯片）由很多个块（Block）组成，块的大小一般是 128KB，256KB，512KB，此处是 128KB。其他的小于 128KB 的，比如 64KB，一般都是下面将要介绍到的 small block 的 Nand Flash。

块 Block，是 Nand Flash 的擦除操作的基本/最小单位。

2.2.5.2 Page 页

每个块里面又包含了很多页（page）。每个页的大小，对于现在常见的 Nand Flash 多数是 2KB，最新的 Nand Flash 的是 4KB、8KB 等，这类的页大小大于 2KB 的 Nand Flash，被称作 big block 的 Nand Flash，对应的发读写命令地址，一共 5 个周期(cycle)，而老的 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。

页 Page，是读写操作的基本单位。

不过，也有例外的是，有些 Nand Flash 支持 subpage（1/2 页或 1/4 页）子页的读写操作，不过一般很少见。

2.2.5.3 oob / Redundant Area / Spare Area

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。

Oob 的读写操作，一般是随着页的操作一起完成的，即读写页的时候，对应地就读写了 oob。

关于 oob 具体用途，总结起来有：

- 标记是否是坏快
- 存储 ECC 数据
- 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，而 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。

2.2.6 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。所以一般将 Flash 翻译为 （快速）闪存。

2.2.7 Flash 相对于普通设备的特殊性

根据上面提到过的，Flash 最小操作单位，相对于普通存储设备，就显得有些特殊。
因为一般存储设备，比如硬盘或内存，读取和写入都是以位（bit）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，这点可以从前面的内部实现机制了解到，对于最初始值，都是 1，所以是 0xFFFFFFFF，而数据的写入，即是将对应的变成 0，而将数据的擦出掉，就是统一地，以 block 为单位，全部一起充电，所有位，都变成初始的 1，而不是像普通存储设备那样，每一个位去擦除为 0。而数据的写入，就是电荷放电的过程，代表的数据也从 1 变为了 0.

所以，总结一下 Flash 的特殊性如下：

图表 5 Flash 和普通设备相比所具有的特殊性

	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 0xFF ③
对于写操作	直接写即可	在写数据之前，要先擦除，然后再写

注：
① 之所以将写操作叫做编程，是因为 flash 是从之前的 EPROM、EEPROM 等继承发展而来，而之前的 EEPROM，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，所以叫做编程。
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，也就是将对应的位由 1 变成 0。

2.2.8 Nand Flash 引脚(Pin)的说明

图表 6 Nand Flash 引脚功能说明

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.
$\overline{CE}$ / $\overline{CE1}$	**CHIP ENABLE** The $\overline{CE}$ / $\overline{CE1}$ input is the device selection control. When the device is in the Busy state, $\overline{CE}$ / $\overline{CE1}$ high is ignored, and the device does not return to standby mode in program or erase operation. Regarding $\overline{CE}$ / $\overline{CE1}$ control during read operation , refer to 'Page Read' section of Device operation.
$\overline{CE2}$	**CHIP ENABLE** The $\overline{CE2}$ input enables the second K9K8G08U0A
$\overline{RE}$	**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/$\overline{B}$ / R/$\overline{B1}$	**READY/BUSY OUTPUT** The R/$\overline{B}$ / R/$\overline{B1}$ 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：

图表 7 Nand Flash 引脚功能的中文说明

引脚名称	引脚功能
I/O0 ~ I/O7	用于输入地址/数据/命令，输出数据
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
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
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
RE#	Read Enable，读使能，在读取数据之前，要先使 CE＃有效。
WE#	Write Enable，写使能，在写取数据之前，要先使 WE＃有效
WP#	Write Protect，写保护
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
Vcc	Power，电源
Vss	Ground，接地
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。

2.2.8.1 为何需要 ALE 和 CLE

硬件上，有了电源的 Vcc 和接地的 Vss 等引脚，很好理解，但是为何还要有 ALE 和 CLE 这样的引脚，为何设计这么多的命令,把整个系统搞这么复杂，关于这点，最后终于想明白了：设计命令锁存使能(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,也就无法实现正确的操作了.

2.2.8.2 Nand Flash 只有 8 个 I/O 引脚的好处

在 Nand Flash 的硬件设计中，你会发现很多个引脚。关于硬件上为何设计这样的引脚，而不是直接像其他存储设备，比如普通的 RAM，直接是一对数据线引出来，多么方便和好理解啊。

关于这样设计的好处：

2.2.8.2.1 减少外围连线：

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。

2.2.8.2.2 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。

说白了，对于旧的 Nand Flash 所实现的驱动，这些软件工作，在换新的硬件的 Nand Flash 的情况下，仍然可以工作，或者是通过极少的修改，就同样可以工作，使得软硬件兼容性大大提高。

2.2.9 Nand flash 的一些典型(typical)的特性

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。
6. Nand Flash 的编程/擦除的寿命：即，最多允许的擦除的次数。

以前老的 Nand Flash，编程/擦除时间比较短，比如 K9G8G08U0M，才 5K 次，而后来的多数也只有 10K=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，达到和接近于之前常见的 Nor Flash，几乎是同样的使用寿命了。

7. 封装形式：48 引脚的 TSOP1 封装 或 52 引脚的 ULGA 封装

2.2.10 Nand Flash 控制器与 Nand Flash 芯片

关于 Nand Flash 的控制器 Controller 和 Nand Flash 芯片 chip 之间的关系，觉得有必要解释一下：

首先，我们要知道的是，我们写驱动，是写 Nand Flash 控制器的驱动，而不是 Nand Flash 芯片的驱动，因为独立的 Nand Flash 芯片，一般来说，是很少直接拿来用的，多数都是硬件上有对应的硬件的 Nand Flash 的控制器，去操作和控制 Nand Flash，包括提供时钟信号，提供硬件 ECC 校验等等功能，我们所写的驱动软件，是去操作 Nand Flash 的控制器，然后由控制器去操作 Nand Flash 芯片，实现我们所要的功能。

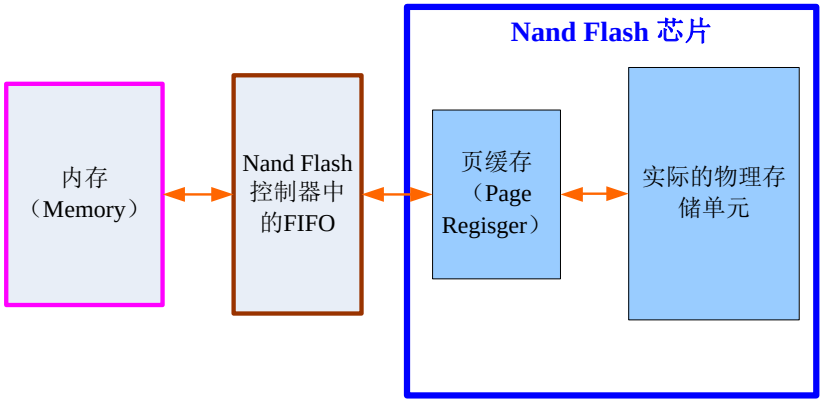
2.2.11 Nand Flash 中的特殊硬件结构

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，就有特殊的设计，就对应某些特殊的硬件特性，就有必要解释解释：

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（Plane），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，本质上就是一个缓存 buffer，但是只是此处 datasheet 里面把其叫

做页寄存器 page register 而已，实际将其理解为页缓存，更贴切原意。
而正是因为有些人不了解此内部结构，才容易产生之前遇到的某人的误解，以为内存里面的数据，通过 Nand Flash 的 FIFO，写入到 Nand Flash 里面去，就以为立刻实现了实际数据写入到物理存储单元中了，而实际上只是写到了这个页缓存中，只有当你再发送了对应的编程第二阶段的确认命令，即 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：

图表 8 Nand Flash 读写时的数据流向



2.2.12 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，就称为其为坏块 Bad Block。
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。与此对应的正常的块，肯定是写入读出都是正常的。

2.2.12.1 坏块的分类

坏块有两种：
（1） 出厂时就有存在的坏块：
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。
（2） 使用过程中产生的坏块：
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。即用坏了的块。

2.2.12.2 坏块的标记

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。
不过，对于现在新出的有些 Nand Flash，很多标记方式，有些变化，有的变成该坏块的第一个页或者第二个页，也有的是，倒数最后一个或倒数第二个页，用于标记坏块的。
具体的信息，请参考对应的 Nand Flash 的数据手册，其中会有说明。

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，在实际的驱动编程开发过程中，为了方便起见，我倒是经常用，其实也没啥大碍，呵呵。不过呢，其实最好的做法是，用“nand erase”只擦除好的块，对于已经标记坏块的块，不要轻易擦除掉，否则就很难区分哪些是出厂时就坏的，哪些是后来使用过程中用坏的了。

2.2.12.3 坏块的管理

对于坏块的管理，在 Linux 系统中，叫做坏块管理（BBM，Bad Block Management），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。

2.2.12.4 坏块的比例

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。

2.2.13 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，。。。

2.2.14 常见的 Nand Flash 的操作

要实现对 Nand Flash 的操作，比如读取一页的数据，写入一页的数据等，都要发送对应的命令，而且要符合硬件的规定，如图：

图表 9 Nand Flash K9K8G08U0A 的命令集合

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(4)	80h---11h	81h---10h	
Copy-Back Program	85h	10h	
Two-Plane Copy-Back Program(4)	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
Chip1 Status(3)	F1h		O
Chip2 Status(3)	F2h		O

从上图可以看到，如果要实现读一个页的数据，就要发送 Read 的命令，而且是分两个周期（Cycle），即分两次发送对应的命令，第一次是 0x00h,第二次是 0x30h，而两次命令中间，需要发送对应的你所要读取的页的地址，关于此部分详细内容，留待后表。
对应地，其他常见的一些操作，比如写一个页的数据(Page Program)，就是先发送 0x80h,然后发生要写入的地址，再发送 0x10h。

【提示】
对于不同厂家的不同型号的 Nand Flash 的基本操作，即读页数据 Read Page，写页数据（对页进行编程）Page Program，擦除整个块的数据 Erase Block 等操作，所用的命令都是一样的，但是针对一些 Nand Flash 的高级的一些特性，比如交错页编程（Interleave Page Program），多片同时编程(Simultaneously Program Multi Plane)等，所用的命令，未必一样，不过对于同一厂家的 Nand Flash 的芯片，那一般来说，都是统一的。

关于一些常见的操作，比如读一个页的 Read 操作和写一个页的 Page Program，下面开始更深入的介绍。

2.2.14.1 页编程（Page Program）注意事项

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（Partial Page Program），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，都是以页为单位进行编程操作的，很少使用到部分页编程。

关于这个部分页编程，本来是一个页的写操作，却用两个命令或更多的命令去实现，看起来是操作多余，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。
这种完成单个操作要分两步发送命令的设计，即先开始设置，再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，由于某种噪音，而产生了 0x60 命令，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。

2.2.14.2 读（Read）操作过程详解

下面以最简单的 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：

从 Nand Flash 的某个页 Page 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，即使我们不太懂 Nand Flash 的细节，但是通过前面的基本知识的介绍，那么以我们的常识，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：

2.2.14.2.1 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：

根据前面关于 Nand Flash 的命令集合介绍，我们知道，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。

2.2.14.2.2 发送命令前的准备工作以及时序图各个信号的具体含义

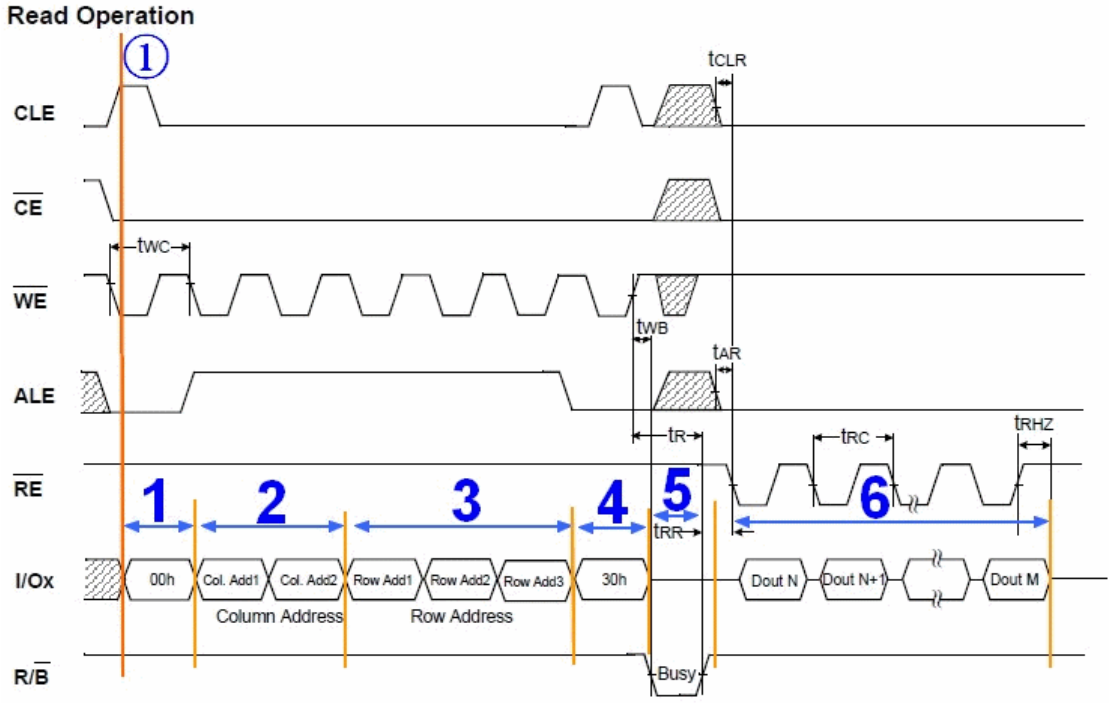
知道了用何命令后，再去了解如何发送这些命令。

【小常识】

在开始解释前，关于”使能”这个词要罗嗦一下，以防有些读者和我以前一样，在听这类词语的时候，属于初次接触，或者接触不多的，就很容易被搞得一头雾水的（虽然该词汇对于某些专业人士说是属于最基本的词汇了，囧）。

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。

图表 10 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（此时是发送图示所示的读命令第二周期的 0x30），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。

2.2.14.2.3 如何计算出我们要传入的行地址和列地址

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：

图表 11 Nand Flash 的地址周期组成

	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	A30	*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 required.

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。
（1）对应地，列地址 A0~A10，就是页内地址，地址范围是从 0 到 2047。
细心的读者可能注意到了，为何此处多出来个 A11 呢？
这样从 A0 到 A11，一共就是 12 位，可以表示的范围就是 0~2^12，即 0~4096 了。
实际上，由于我们访问页内地址，可能会访问到 oob 的位置，即 2048-2111 这 64 个字节的范围内，所以，此处实际上只用到了 2048～2111，用于表示页内的 oob 区域，其大小是 64 字节。
（2）对应地，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。

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。
注意，下面这样的方法，是错误的：
0x36B204B8 = **11 0110 1011 0010 0000** **0**100 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 ： **0**100 = 0x04
3rd 周期，A19～A12 ： **0010 0000** = 0x20

4th 周期，A27～A20　：**0110 1011** = 0x6B
5th 周期，A30～A28　：　　**11** = 0x03

注意：
与图 7 中对应的，*L，意思是地电平，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。

而至于上述计算方法为何是错误的，那是因为上面计算过程中，把第 11 位的值，本来是属于页号的位 A11，给算成页内地址里面的值了。

应该是这样计算，才是对的：
0x36B204B8 = **11 0110 1011 0010 0000 0**100 1011 1000
1st 周期，A7～A0　：1011 1000 = 0x B8
2nd 周期，A10～A8　：　　100 = 0x04
3rd 周期，A19～A12　：**010 0000 0** = 0x40
4th 周期，A27～A20　：**110 1011 0** = 0xD6
5th 周期，A30～A28　：　　**11 0** = 0x06

那有人会问了，上面表 11 中，不是明明写的 A0 到 A30，其中包括 A11，不是正好对应着此处地址中的 bit0 到 bit30 吗？
其实，我开始也是犯了同样的错误，误以为我们要传入的地址的每一位，就是对应着表 11 中的 A0 到 A30 呢，实际上，表 11 中的 A11，是比较特殊的，只有当我们访问页内地址处于 oob 的位置，即属于 2048~2111 的时候，A11 才会其效果，才会用 A0-A11 用来表示对应的某个属于 2048~2111 的某个值，属于 oob 的某个位置。
而我们此处的页内地址为 2108，还没有超过 2047 呢，所以 A11 肯定是 0.

这么解释，显得很绕，很难看懂。
换种方式来解释，就容易听懂了：
说白了，我们就是要访问第 7000 个块中的第 25 页中的 1208 字节处，对应着
页内地址
=1208
=0x4B8

页号
=块数×页数/块 ＋ 块内的页号
= 7000×（128K/2K） ＋25
= 7000×64 + 25
= 448025
=0x6D619

也就是，我们要访问 0x6D619 页内的 0x4B8 地址，这样很好理解吧,^_^。
然后对应的：
页内地址=0x4B8，分成两个对应的列地址，就变成
0x4B8 ：列地址 1=0xB8，列地址 2=0x04

页号=0x6D619，分成三个行号就是：
0x6D619：行号 1=0x19，行号 2=0xD6，行号 3=0x06

再回头看看上面的计算方法，
最开始计算出来的：
列地址 1=0xB8
列地址 2=0x04
行号 1=0x20
行号 2=0x6B
行号 3=0x03
是错误的。

而第二次计算正确的：
列地址 1=0xB8
列地址 2=0x04
行号 1=0x19
行号 2=0xD6
行号 3=0x06
才是对的，也和我们此处自己手动计算，是一致的。

第一次之所以计算错，就是错误的把行地址的最低一位 A11，放到列地址中的最高位了。

至此，才算把如何手动计算行地址和列地址，解释明白和正确了。

对应的，Linux 的源码\drivers\mtd\nand\nand_base.c 中，也是这样处理的：

```
static void nand_command_lp(struct mtd_info *mtd, unsigned int command,
              int column, int page_addr)
{
。。。
       /* Serially input address */
       if (column != -1) {
。。。
           chip->cmd_ctrl(mtd, column, ctrl); /* 发送 Col Addr 1 */
           ctrl &= ~NAND_CTRL_CHANGE;
           chip->cmd_ctrl(mtd, column >> 8, ctrl); /* 发送 Col Addr 2 */
       }
       if (page_addr != -1) {
           chip->cmd_ctrl(mtd, page_addr, ctrl); /* 发送 Row Addr 1 */
           chip->cmd_ctrl(mtd, page_addr >> 8, /* 发送 Row Addr 2 */
                     NAND_NCE | NAND_ALE);
           /* One more address cycle for devices > 128MiB */
           if (chip->chipsize > (128 << 20))
               chip->cmd_ctrl(mtd, page_addr >> 16, /* 发送 Row Addr 3 */
```

```
                        NAND_NCE | NAND_ALE);
        }
    }
```

其中，上述源码中的 column，即页内地址，多数情况下，都是 0，即使不是 0，也可以直接通过将传入的地址除于页地址所得的余数，就是列地址了。
而 page_addr 即页号，也很简单，就是通过要访问的地址，除于页大小，即可得到。

因此，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：
列地址 1=0xB8
列地址 2=0x04
然后再传 3 个行地址的：
行号 1=0x19
行号 2=0xD6
行号 3=0x06
这样硬件才能识别。
而接下来的内容，也就是介绍硬件是如何处理这些输入的。

2.2.14.2.4 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。
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。
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。
接下来再传入三个行地址。对应的也就是页号。
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。
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 1208 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。
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。

2.2.15 Nand Flash 的一些高级特性

2.2.15.1 Nand Flash 的 Unique ID

2.2.15.1.1 什么是 Unique ID 唯一性标识

Unique ID，翻译为中文就是，独一无二的 ID，唯一性标识。
很明显，这个 Unique ID 是为了用来识别某些东西的，每一个东西都拥有一个独一无二的标识信息。
在 Nand Flash 里面的 Unique ID，主要是某个 ID 信息，保证每个 Nand Flash 都是独一无二的。主要用于其它的使用 Nand Flash 的用户，根据此 unique id 去做加密等应用，实现某些安全方面的应用。
简而言之，就是用 Nand Flash 的 Unique ID 来实现安全相关的应用，比如加密，版权保护等等。

2.2.15.1.2 不同 Nand Flash 厂商的对 Unique ID 的不同的实现方法

此处，继续解释之前，还要再次赘述一下：
目前 Nand Flash 的厂家有 samsung，Toshiba，Intel， Hynix，Micron，Numonyx，Phison ，SanDisk，Sony，Spansion 等。
由于前面所说的 Nand Flash 的规范之争，即 Toshiba & Samsung 和 Intel + 其它厂商（Hynix，Micron，Numonyx，Phison ，SanDisk，Sony，Spansion 等）之争，导致对于 Unique ID 这么个小的功能点（feature），不同的方面，弄出了不同的实现（做法）。

下面就来解释一下各个厂家关于 Unique ID 的实现方法，以及如何读取对应的 Unique ID：

2.2.15.1.2.1 Toshiba 东芝的 Nand 的 Unique ID

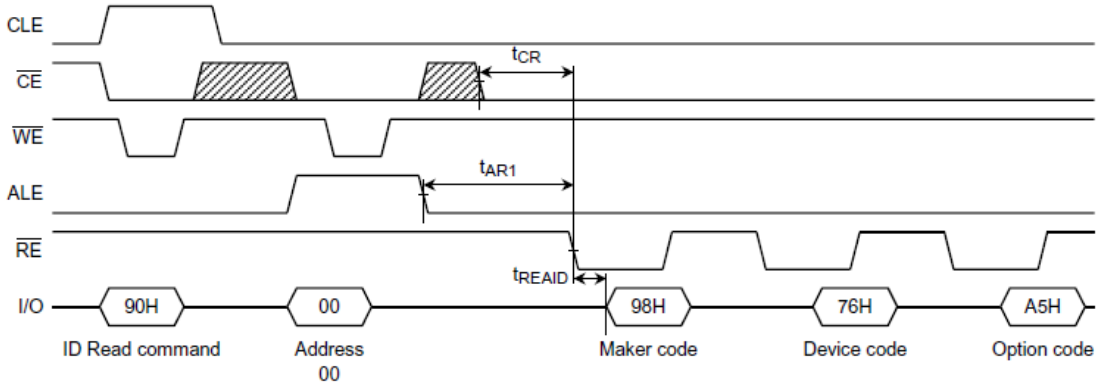
网上找到一个 datasheet：
Toshiba TH58NS512DC
http://datasheet.elcodis.com/pdf/11/23/112371/th58ns512dc-to51y.pdf
中有提到：
P1:
“The TH58NS512DC is a SmartMediaTM with ID and each device has 128 bit unique ID number embedded in the device. This unique ID number is applicable to image files, music files, electronic books, and so on where copyright protection is required.”
即每个 Toshiba 的 TH58NS512DC 中，都有一个 128 bit=16 byte 的 Unique ID，可用于图片，音乐，电子书等应用中的版权保护。

P24：

图表 12 Toshiba 的 Unique ID

ID Read

The TH58NS512 contains ID codes which identify the device type and the manufacturer. The ID codes can be read out under the following timing conditions:



For the specifications of the access times t_{REAID}, t_{CR} and t_{AR1} refer to the AC Characteristics.

Figure13. ID Read timing

Table 6. Code table

	I/O8	I/O7	I/O6	I/O5	I/O4	I/O3	I/O2	I/O1	Hex Data
Maker code	1	0	0	1	1	0	0	0	98H
Device code	0	1	1	1	0	1	1	0	76H
Option code	1	0	1	0	0	1	0	1	A5H*

* The A5H for the 3rd byte of ID read means the existence of 128 bit unique ID number in the device.

How to read out unique ID number

The 128 bit unique ID number is embedded in the device. The procedure to read out the ID number is available using special command which is provided under a non-disclosure agreement.

2.2.15.1.2.2 读取 Toshiba 的 Nand 的 Unique ID

从上面可以看出，Toshiba 的 Nand 中，关于 Unique ID，是需要先通过普通的 0x90，即 Read ID 的命令，去读取 Nand 的 ID，找到第三个字节（3rd byte），然后判断其是否是 0xA5，如果是 0xA5，然后才能确定此 Nand 里面是有 Unique ID 的，然后才有去读取 Unique ID 这一说。

而关于如何独缺 Unique ID，则需要和自己去联系 Toshiba，和其签订 NDA 协议后，才可得知读取 Nand 的 Unique ID 的方法。

2.2.15.1.3 Samsung 三星的 Nand 的 Unique ID

网上找到的某款三星的 Nand 的 datasheet：

Samsung K9F5608U0B

http://hitmen.c02.at/files/docs/psp/ds_k9f5608u0b_rev13.pdf

“6. Unique ID for Copyright Protection is available

- The device includes one block sized OTP (One Time Programmable), which can be used to increase system security or to provide identification capabilities. Detailed information can be obtained by contact with Samsung”

即，Samsung 的 Nand 的 Unique ID，也和 Toshiba 的用途类似，也主要是用于版权保护，但是其实现却不同。

Samsung 的 Unique ID 的实现，是专门在 Nand 里面配备了一个 OTP 的 Block，而此 Nand 芯片的 Block 大小是 16KB。

而关于如何操作此 OTP，即如何写入数据和读取数据，此处未说明。

欲知详情，请联系三星。

不过个人理解，应该和普通的 block 的操作类似，即普通的 block，包含很多 page，每个 page 的操作，有对应的 page read，用对应的 page read 命令来读取此特殊的 OTP 的 block 里面的数据。

而此 OTP 的 block 里面的数据是什么，完全取决于自己最开始往里面写入了什么数据。说白了就是，你根据自己需求，在你的产品出厂的时候，写入对应的数据，比如该款产品的 SN 序列号等数据，然后自己在用 page read 读取出相应数据后，自己解析，得到自己要的信息，用于自己的用途，比如版权保护等。

2.2.15.1.3.1 读取 Samsung 的 Nand 的 Unique ID

如前所述：

关于如何操作此 OTP 的 block，即如何写入数据和读取数据，此处未说明。

即想要知道如何读取 Samsung 的 Nand 的 Unique ID，请自己去问三星。

2.2.15.1.4 遵循 ONFI 规范的厂商的 Nand 的 Unique ID

主要指的是 Intel 英特尔，Hynix 海力士，Micron 美光，Numonyx 恒亿，Spansion 飞索等公司。

对应的 Nand 的 Unique ID 的相关定义，ONFI 的规范中都有，现简要摘录如下：

ONFI 2.2

http://onfi.org/wp-content/uploads/2009/02/ONFI%202_2%20Gold.pdf

ONFI 规范中，在“5.7.1. Parameter Page Data Structure Definition”中，如图：

图表 13 ONFI 的参数页数据结构定义

Byte	O/M	Description
		Revision information and features block
0-3	M	Parameter page signature Byte 0: 4Fh, "O" Byte 1: 4Eh, "N" Byte 2: 46h, "F" Byte 3: 49h, "I"
4-5	M	Revision number 5-15 Reserved (0) 4 1 = supports ONFI version 2.2 3 1 = supports ONFI version 2.1 2 1 = supports ONFI version 2.0 1 1 = supports ONFI version 1.0 0 Reserved (0)
6-7	M	Features supported 9-15 Reserved (0) 8 1 = supports program page register clear enhancement 7 1 = supports extended parameter page 6 1 = supports interleaved read operations 5 1 = supports source synchronous 4 1 = supports odd to even page Copyback 3 1 = supports interleaved program and erase operations 2 1 = supports non-sequential page programming 1 1 = supports multiple LUN operations 0 1 = supports 16-bit data bus width
8-9	M	Optional commands supported 10-15 Reserved (0) 9 1 = supports Reset LUN 8 1 = supports Small Data Move 7 1 = supports Change Row Address 6 1 = supports Change Read Column Enhanced 5 1 = supports Read Unique ID 4 1 = supports Copyback 3 1 = supports Read Status Enhanced 2 1 = supports Get Features and Set Features 1 1 = supports Read Cache commands 0 1 = supports Page Cache Program command
10-11		Reserved (0)

定义了一个 page 的数据，用于存储对应的 Nand 的各种参数，其中，有一个第 8 个字节的 bit5==1 的时候，表示支持“Read Unique ID”的命令，即说明此 Nand 芯片支持此命令，如果 byte8 的 bit5==0，那么说明不支持，也就没法去读 Unique ID 了。

2.2.15.1.4.1 读取遵循 ONFI 的厂商的 Nand 的 Unique ID

如果经过上述判断，此符合 ONFI 的 Nand Flash 支持 Read Unique ID 命令，次此时就可以通过该命令来读取对应的 Nand Flash 的 Unique ID 了。

此 Read Unique ID 的详细解释为：

“**5.8. Read Unique ID Definition**

The Read Unique ID function is used to retrieve the 16 byte unique ID (UID) for the device. The unique ID when combined with the device manufacturer shall be unique.

The UID data may be stored within the Flash array. To allow the host to determine if the UID is

without bit errors, the UID is returned with its complement, as shown in Table 47. If the XOR of the UID and its bit-wise complement is all ones, then the UID is valid.”

即用 Read Unique ID 命令来读取 128bit=16 字节的 Unique ID，但是呢，为了用于防止写入的 Unique ID 有误，因此在 16 字节后面又添了个对应的补码，即每位取反的结果，这样前 16 字节的 Unique ID 和后 16 字节的 Unique ID 的补码，构成了 32 字节，算作一组，如下图所示：

图表 14 ONFI 中 Unique ID 的结构

Bytes	Value
0-15	UID
16-31	UID complement (bit-wise)

Table 47 UID and Complement

“To accommodate robust retrieval of the UID in the case of bit errors, sixteen copies of the UID and the corresponding complement shall be stored by the target. For example, reading bytes 32-63 returns to the host another copy of the UID and its complement. Read Status Enhanced shall not be used during execution of the Read Unique ID command.

Figure 57 defines the Read Unique ID behavior. The host may use any timing mode supported by the target in order to retrieve the UID data.”

而为了进一步防止出错，将上面 32 字节算一组，重复了 16 次，将这 16 个 32 字节的数据，存在 Nand Flash 里面，然后用 Read Unique ID 命令去读取出来，取得其中某个 32 字节即可，然后判断前 16 字节和后 16 字节取反，如果结果所有位都是 1，那么结果即为 16 个 0xFF，那么说明 Unique ID 是正确的。否则说明有误。

Read Unique ID 的命令的详细格式如下图所示：

图表 15 ONFI 中 Read Unique ID 命令的时序图

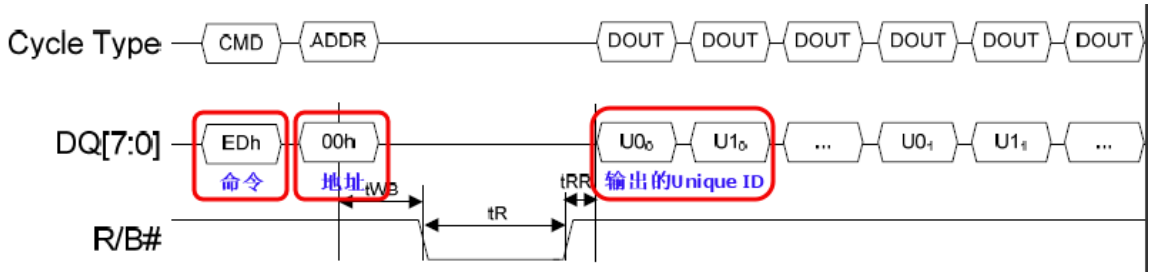


Figure 57 Read Unique ID command timing

$U0_k$-Un_k The kth copy of the UID and its complement. Sixteen copies are stored. Reading beyond 512 bytes returns indeterminate values.

即先发送 0xED 命令，再发送 0x00 地址，然后等待 Nand Flash 的 busy 状态结束，就可以读取出来的那 16 组的 32 个字节了，然后用上面办法去判断，找到前 16 字节和后 16 字节异或得到结果全部 16 字节都为 0xFF，即说明得到了正确的 Unique ID。至此，符合 ONFI 规范的 Unique ID，就读取出来了。

2.2.15.2 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。

对此也许有人会问了，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，但是芯片内部却仍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。

总的来说就是：由于某些外部应用所需要的访问 Nand Flash 的频率比较低，而 Nand Flash 内部操作速度比较快，所以在针对 Nand Flash 的读或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。

【提示】

1. 如果想要操作硬件 Nand Flash 芯片，先要将对应的 CE#（低有效）片选信号拉低，选中该芯片，然后才能做接下来的读写操作所要做的发命令，发数据等动作。

2．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，也许你对 5mA 没太多概念，给你说个数据你就知道了：当时为了针对 MP3 播放功耗进行优化，整个系统优化之后的待机功耗，也才 10 个 mA 左右的，所以节省 5mA 已经算是很不错的功耗优化了。

2.2.15.3 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。

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，而这部分操作，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。

2.2.15.4 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。

2.2.15.5 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。

2.2.15.6 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。

2.3 软件方面

如果想要在 Linux 下编写 Nand Flash 驱动，那么就先要搞清楚 Linux 下，关于此部分的整个框架。弄明白，系统是如何管理你的 Nand Flash 的，以及，系统都帮你做了那些准备工作，而剩下的，驱动底层实现部分，你要去实现哪些功能，才能使得硬件正常工作起来。

2.3.1 Nand Flash 相关规范 – ONFI 和 LBA

在介绍 Nand Flash 的软件细节方面之前，先来介绍一下 Nand Flash 的两个相关的规范：ONFI 和 LBA。

2.3.1.1 ONFI 是什么

ONFI 规范，即 Open Nand Flash Interface specification。

ONFI 是 Intel 主导的，其他一些厂家（Hynix，Micron，Numonyx，Phison ，SanDisk，Sony，Spansion 等）参与制定的，统一了 Nand Flash 的操作接口。
所谓操作接口，就是那些对 Nand Flash 操作的命令等内容。
而所谓统一，意思是之前那些 Nand Flash 的操作命令等，都是各自为政，虽然大多数常见的 Nand Flash 的操作，比如 page read 的命令是 0x00,0x30，page write 的命令是 0x80,0x10 等，但是有些命令相关的内容，很特别且很重要的一个例子就是，每个厂家的 Nand Flash 的 read id 的命令，虽然都是 0x90，但是读取出来的几个字节的含义，每个厂家定义的都不太一样。
因此，才有统一 Nand Flash 的操作接口这一说。

ONFI 规范，官网可以下载的到：
http://onfi.org/specifications/
比如：
ONFI 2.2 Spec
http://onfi.org/wp-content/uploads/2009/02/ONFI%202_2%20Gold.pdf

ONFI 规范中定义的 Nand Flash 的命令集合为：

图表 16 ONFI 中的 Nand Flash 的命令集合

Command	O/M	1st Cycle	2nd Cycle	Acceptable while Accessed LUN is Busy	Acceptable while Other LUNs are Busy	Target level commands
Read	M	00h	30h		Y	
Interleaved	O	00h	32h		Y	
Copyback Read	O	00h	35h		Y	
Change Read Column	M	05h	E0h		Y	
Change Read Column Enhanced	O	06h	E0h		Y	
Read Cache Random	O	00h	31h		Y	
Read Cache Sequential	O	31h			Y	
Read Cache End	O	3Fh			Y	
Block Erase	M	60h	D0h		Y	
Interleaved	O	60h	D1h		Y	
Read Status	M	70h		Y	Y	
Read Status Enhanced	O	78h		Y	Y	
Page Program	M	80h	10h		Y	
Interleaved	O	80h	11h		Y	
Page Cache Program	O	80h	15h		Y	
Copyback Program	O	85h	10h		Y	
Interleaved	O	85h	11h		Y	
Small Data Move[2]	O	85h	11h		Y	
Change Write Column[1]	M	85h			Y	
Change Row Address[1]	O	85h			Y	
Read ID	M	90h				Y
Read Parameter Page	M	ECh				Y
Read Unique ID	O	EDh				Y
Get Features	O	EEh				Y
Set Features	O	EFh				Y
Reset LUN	O	FAh		Y	Y	
Synchronous Reset	O	FCh		Y	Y	Y
Reset	M	FFh		Y	Y	Y

NOTE:
1. Change Write Column specifies the column address only. Change Row Address specifies the row address and the column address. Refer to the specific command definitions.
2. Small Data Move's first opcode may be 80h if the operation is a program only with no data output. For the last second cycle of a Small Data Move, it is a 10h command to confirm the Program or Copyback operation.

可以看到，其中常见的一些命令，比如 page read（0x00,0x30），和 page write（0x80,0x10），block erase（0x60,0xD0），Reset（0xFF）等等命令，都是和普通的 Nand Flash 的命令是一样的，而额外多出一些命令，比如 Read Unique ID（0xED）等命令，是之前某些 Nand Flash 命令所不具有的。

如此，定义了 Nand Flash 的操作的命令的集合以及发送对应命令所遵循的时序等内容。

2.3.1.1.1 ONFI Block Abstracted NAND

ONFI 还定义了另外一个规范：
ONFI Block Abstracted Nand Specification

http://onfi.org/wp-content/uploads/2009/02/BA_NAND_rev_1_1_Gold.pdf
即 ONFI LBA Nand，简单说就是，逻辑块寻址的 Nand。其含义和 Toshiba 的 LBA，基本没有太多区别。

2.3.1.1.2 ONFI 的好处

ONFI 规范定义了之后，每家厂商的 Nand Flash，只要符合这个 ONFI 规范，然后上层 Nand Flash 的软件，就可以统一只用一种了，换句话说，我写了一份 Nand Flash 的驱动后，就可以操作所有和 ONFI 兼容的 Nand Flash 了，整个 Nand Flash 的兼容性，上层软件的兼容性，互操作性，就大大提高了。
而且，同样的，由于任何规范在定义的时候，都会考虑到兼容性和扩展性，ONFI 也不例外。针对符合 ONFI 规范的，写好的软件，除了可以操作多家与 ONFI 兼容的 Nand Flash 之外，而对于以后出现的新的技术，新制程的 Nand Flash，只要符合 ONFI 规范，也同样可以支持，可以在旧的软件下工作，而不需要由于 Nand Flash 的更新换代，而更改上层软件和驱动，这个优势，由于对于将 Nand Flash 芯片集成到自己系统中的相关开发人员来说，是个好消息。

2.3.1.2 LBA 规范是什么

LBA Nand Flash，Logical Block Address，逻辑块寻址的 Nand Flash，是 Nand Flash 大厂之一的 Toshiba，自己独立设计出来的新一代的 Nand Flash 的规范。

之所以叫做逻辑块寻址，是相对于之前常见的，普通的 Nand Flash 的物理块的寻址来说的。常见的 Nand Flash，如果要读取和写入数据，所用的对应的地址是对应的：block 地址+block 内的 Page 地址+Page 内的偏移量 = 绝对的物理地址，
此物理块寻址，相对来说有个缺点，那就是，由于之前提到的 Nand Flash 会出现使用过程中出现坏块，所以，遇到这样的坏块，首先坏块管理要去将此坏块标记，然后将坏块的数据拷贝到另一个好的 block 中，再继续访问新的 block。
而且数据读写过程中，还要有对应的 ECC 校验，很多情况下，也都是软件来实现这部分的工作，即使是硬件的 ECC 校验，也要写少量的软件，去操作对应寄存器，读取 ECC 校验的结果，当然别忘了，还有对应的负载平衡等工作。
如此的这类的坏块管理工作，对于软件来说，很是繁重，而且整个系统实现起来也不是很容易，所以，才催生了一个想法，是否可以把 ECC 校验，负载平衡，坏块管理，全部都放到硬件实现上，而对于软件来说，我都不关心，只关心有多少个 Block 供我使用，用于数据读写。
针对于此需求，Toshiba 推出了 LBA 逻辑块寻址的 Nand Flash，在 Nand Flash 存储芯片之外，加了对应一个硬件控制权 Controller，实现了上述的坏块管理，ECC 校验，负载平衡等工作，这样使得人家想要用你 LBA 的 Nand Flash 的人，去开发对应的软件来驱动 LBA Nand Flash 工作，相对要做的事情，就少了很多，相对来说就是减轻了软件系统集成方面的工作，提高了开发效率，缩短了产品上市周期。

LBA Nand，最早放出对应的样片（sample）是在 2006 年 8 月。
网上找到一个 LBA Nand Flash 的简介：
http://www.toshiba-components.com/prpdf/5678E.pdf
现早已经量产，偶在之前开发过程中，就用过其某款 LBA 的 Nand Flash。

目前网上还找不到免费的 LBA 的规范。除非你搞开发，和 Toshiba 签订 NDA 协议后，才可

以拿到对应的 specification。

关于 Toshiba LBA Nand 规范，在此多说一点（参考附录中：lba-core.c）：
LBA Nand 分为 PNP，VFP 和 MDP 三种分区。
PNP 主要用于存放 Uboot 等启动代码；
VFP 主要用于存放 uImage 等内核代码；
MDP 主要用于存放用户的数据，以及 rootfs 等内容；

2.3.1.3 为何会有 ONFI 和 LBA

在解释为何会有 ONFI 和 LBA 之前，先来个背景介绍：
目前 Nand Flash 的厂家有 Samsung，Toshiba，Intel， Hynix，Micron，Numonyx，Phison ，SanDisk，Sony，Spansion 等。

2.3.1.3.1 技术层面的解释

ONFI 的出现，上面已经解释过了，就是为了统一 Nand Flash 的接口，使得软件兼容性更好；而 LBA 的出现，是为了减轻软件方面对 Nand Flash 的各种管理工作。上面这些解释，其实只是技术上解释。

2.3.1.3.2 现实层面的解释

而现实方面是，ONFI 是 Intel 主导的，其他一些 Nand Flash 厂商参与制定出来的一套 Nand Flash 的规范，但是却没有得到 Nand Flash 的两个大厂家的认可，一个是以第一厂商自居 samsung，另一个是 Nand Flash 技术引导者的 Toshiba。所以，可以算是在 Nand Flash 领域里，老三带着一帮小的，定了一个规范，但是老大和老二却不买账。因此，技术上的 Nand 的老大 Toshiba 联手产量上的老大，自己去推出了另外一套规范 LBA。
这可以称得上是典型的规范之争吧。

2.3.1.4 ONFI 和 LBA 的区别和联系

总的来说，ONFI 在对于旧的 Nand Flash 的兼容上，都是相对类似的。

2.3.1.4.1 ONFI 和 LBA 的区别

总的来说：
ONFI 规范，更注重对于 Nand Flash 的操作接口方面的定义 + ONFI LBA Nand 的定义，而 Toshiba LBA 规范，主要侧重于 LBA 的 Nand 的定义。

2.3.1.4.2 ONFI 和 LBA 的联系

ONFI Block Abstracted NAND Specification，基本上和 Toshiba 的 LBA，没太多区别，只是 Toshiba 的 LBA 规范中，又多了些其他模式和应用类别。

总的来说，可以这么划分：
ONFI = Nand Flash 操作接口的统一 + ONFI 的 LBA Nand
Toshiba LBA = 等价于 ONFI 的 LBA Nand + 多种模式和对应的不同应用

2.3.2 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 子系统的好处，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。

图表 17 MTD 设备和硬盘设备之间的区别

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.

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附录中 MTD 设备的文章，该文章是比较详细地介绍了整个 MTD 框架和流程，方便大家理解整个 mtd 框架和 Nand Flash 驱动。

2.3.2.1 Linux MTD 中检测不同类型 Nand Flash 的 ID 部分的代码

关于 nand flash，由于各个厂家的 read id 读出的内容的定义，都不同，导致，对于读出的 id，分别要用不同的解析方法，下面这段代码，是我之前写的，本来打算自己写信去推荐到 Linux MTD 内核源码的，不过后来由于没搞懂具体申请流程，就放弃了。不过，后来，看到 Linux 的 MTD 部分更新了，加了和下面类似的做法。

此处只是为了记录下来，也算给感兴趣的人一个参考吧。

文件：**\linux-2.6.28.4\drivers\mtd\nand\ nand_base.c**


```
/*
 * Get the flash and manufacturer id and lookup if the type is supported
 */
static struct nand_flash_dev *nand_get_flash_type(struct mtd_info *mtd,
                      struct nand_chip *chip,
                      int busw, int *maf_id)
{
... ...
    chip->chipsize = (uint64_t)type->chipsize << 20;

/* 针对不同的 MLC 和 SLC 的 nand flash，添加了不同的解析其 ID 的方法 */
    /* Newer devices have all the information in additional id bytes */
    if (!type->pagesize) {
        int erase_bits, page_base, block_base, old_50nm, new_40nm;
        uint8_t id3rd, id4th, id5th, id6th, id7th;

        /* The 3rd id byte holds MLC / multichip data */
        chip->cellinfo = id3rd = chip->read_byte(mtd);
        /* The 4th id byte is the important one */
        id4th = chip->read_byte(mtd);
        id5th = chip->read_byte(mtd);
        id6th = chip->read_byte(mtd);
        id7th = chip->read_byte(mtd);
        /* printk(KERN_INFO " (ID:%02x %02x %02x %02x %02x %02x %02x) ",
            id1st, id2nd, id3rd, id4th, id5th, id6th, id7th); */

        if (nand_is_mlc(chip->cellinfo)) {
            /*
             * MLC:
             * 50nm has 5 bytes ID, further read ID will periodically output
             * 40nm has 6 bytes ID
             */

            /*
             * the 4th byte is not the same meaning for different manufature
            */
            if (NAND_MFR_SAMSUNG == *maf_id) {
                /* samsung MLC chip has several type ID meanings:
                (1)50nm serials, such as K9GAG08U0M
                (2)40nm serials, such as K9LBG08UXD
                */

                /* old 50nm chip will periodically output if read further
ID */
```

```
                old_50nm = (id1st == id6th) && (id2nd == id7th);
                /* is 40nm or newer */
                new_40nm = id6th & 0x07;
                if ((!old_50nm) && new_40nm) {
                    /*
                     * Samsang
                     * follow algorithm accordding to datasheets of:
                     * K9LBG08UXD_1.3 (40nm),
                     * ID(hex): EC D7 D5 29 38 41
                     * this algorithm is suitable for new chip than 50nm
                     * such as K9GAG08u0D,
                     * ID(hex): EC D5 94 29 B4 41
                     */

                    int bit236;

                    /* Calc pagesize, bit0,bit1: page size */
                    page_base = (1 << 11);  /* 2KB */
                    mtd->writesize = page_base * (1 << (id4th & BIT01));
                    block_base = (1 << 17); /* 128 KB */
                    /* Calc block size, bit4,bit5,bit7: block size */
                    erase_bits = (id4th >> 4) & BIT01; /* get bit4,bit5 */
                    erase_bits |= (id4th >> 5) & BIT(2); /* get bit7 and
combine them */
                    mtd->erasesize = block_base * (1 << erase_bits);
                    /* Calc oobsize, bit2,bit3,bit6: oob size */
                    bit236 = (id4th >> 2) & BIT01; /* get bit2,bit3 */
                    bit236 |= (id4th >> 4) & BIT(2); /* get bit6 and combine
them */
                    switch (bit236) {
                    case 0x01:
                        mtd->oobsize = 128;
                        break;
                    case 0x02:
                        mtd->oobsize = 218;
                        break;
                    default:
                        /* others reserved */
                        break;
                    }
                }
                else {
                    /*
                     * Samsang
```

```
                * follow algorithm accordding to datasheets of:
                * K9GAG08U0M (50nm)
                * this algorithm is suitable for old 50nm chip
                */

               goto slc_algorithm;
           }
       }
       else if (NAND_MFR_TOSHIBA == *maf_id) {
           /*
            * Toshiba
            * follow algorithm guess from ID of TC58NVG3D1DTG00:
            * Toshiba MLC TC58NVG3D1DTG00 1GB 8bit 1chip
            * 4K+218 512K+27K 3.3V, (ID:98 D3 94 BA 64 13 42)
            */
           int bit23;

           /* Calc pagesize, bit0,bit1: page size */
           page_base = (1 << 10);  /* 1KB */
           mtd->writesize = page_base * (1 << (id4th & BIT01));
           block_base = (1 << 16); /* 64 KB */
           /* Calc block size, bit4,bit5: block size */
           erase_bits = (id4th >> 4) & BIT01; /* get bit4,bit5 */
           mtd->erasesize = block_base * (1 << erase_bits);
           /* Calc oobsize, use spare/redundant area bit */
           bit23 = (id4th >> 2) & BIT01; /* get bit2,bit3 */
           switch (bit23) {
           case 0x01:
               mtd->oobsize = 128;
               break;
           case 0x02:
               mtd->oobsize = 218;
               break;
           default:
               /* others reserved */
               break;
           }
           /* Get buswidth information: x8 or x16 */
           busw = ((id4th >> 6) & BIT(0)) ? NAND_BUSWIDTH_16 : 0;
       }
       else if (NAND_MFR_MICRON == *maf_id) {
           /*
            * Micron
            * follow algorithm accordding to datasheets of:
```

```
                 * 29F32G08CBAAA
                 */
                int spare_area_size_bit;

                /* Calc pagesize, bit0,bit1: page size */
                page_base = (1 << 10);  /* 1KB */
                mtd->writesize = page_base * (1 << (id4th & 0x03));
                block_base = (1 << 16); /* 64 KB */
                /* Calc block size, bit4,bit5: block size */
                erase_bits = (id4th >> 4) & BIT01; /* get bit4,bit5 */
                mtd->erasesize = block_base * (1 << erase_bits);
                /* Calc oobsize, use spare/redundant area bit */
                spare_area_size_bit = (id4th >> 2) & BIT(0);
                if (spare_area_size_bit) /* special oob */
                    mtd->oobsize = 218;
                else /* normal */
                    mtd->oobsize = mtd->writesize / NAND_PAGE_OOB_RATIO;
                /* Get buswidth information: x8 or x16 */
                busw = ((id4th >> 6) & BIT(0)) ? NAND_BUSWIDTH_16 : 0;
            }
            else {
                /*
                 * Others
                 * FIXME: update follow algrithm,
                 * according to different manufacture's chip's datasheet
                 */

                goto slc_algorithm;
            }
        }
        else {
            /*
             * SLC, only has 4 bytes ID, further read will output
periodically, such as:
             * Hynix : HY27UG084G2M, only has 4 byte ID,
             * following read ID is periodically same as the 1st ~ 4th byte,
             * for HY27UG084G2M is : 0xAD 0xDC 0x80 0x15 0xAD 0xDC 0x80
0x15 .....
             */
slc_algorithm:
            /* Calc pagesize, bit0,bit1: page size */
            page_base = (1 << 10);  /* 1KB */
            mtd->writesize = page_base * (1 << (id4th & BIT01));
            block_base = (1 << 16); /* 64 KB */
```

```
            /* Calc block size, bit4,bit5: block size */
            erase_bits = (id4th >> 4) & BIT01; /* get bit4,bit5 */
            mtd->erasesize = block_base * (1 << erase_bits);
            /* Calc oobsize, use fixed ratio */
            mtd->oobsize = mtd->writesize / NAND_PAGE_OOB_RATIO;
            /* Get buswidth information: x8 or x16 */
            busw = ((id4th >> 6) & BIT(0)) ? NAND_BUSWIDTH_16 : 0;
        }
    } else {
        /*
         * Old devices have chip data hardcoded in the device id table
         */
        mtd->erasesize = type->erasesize;
        mtd->writesize = type->pagesize;
        mtd->oobsize = mtd->writesize / NAND_PAGE_OOB_RATIO;
        busw = type->options & NAND_BUSWIDTH_16;
    }
/*以上内容，主要是更加不同厂家的 nand flash 的 datasheet，一点点总结出来的算法。
最新的 Linux 的 MTD 部分，已经添加了类似如上部分的代码。此处贴出来，仅供参考。*/
    /*
     * Check, if buswidth is correct. Hardware drivers should set
     * chip correct !
     */
    if (busw != (chip->options & NAND_BUSWIDTH_16)) {
        printk(KERN_INFO "NAND device: Manufacturer ID:"
               " 0x%02x, Chip ID: 0x%02x (%s %s)\n", *maf_id,
               dev_id, nand_manuf_ids[maf_idx].name, mtd->name);
        printk(KERN_WARNING "NAND bus width %d instead %d bit\n",
               (chip->options & NAND_BUSWIDTH_16) ? 16 : 8,
               busw ? 16 : 8);
        return ERR_PTR(-EINVAL);
    }
... ...
}
```

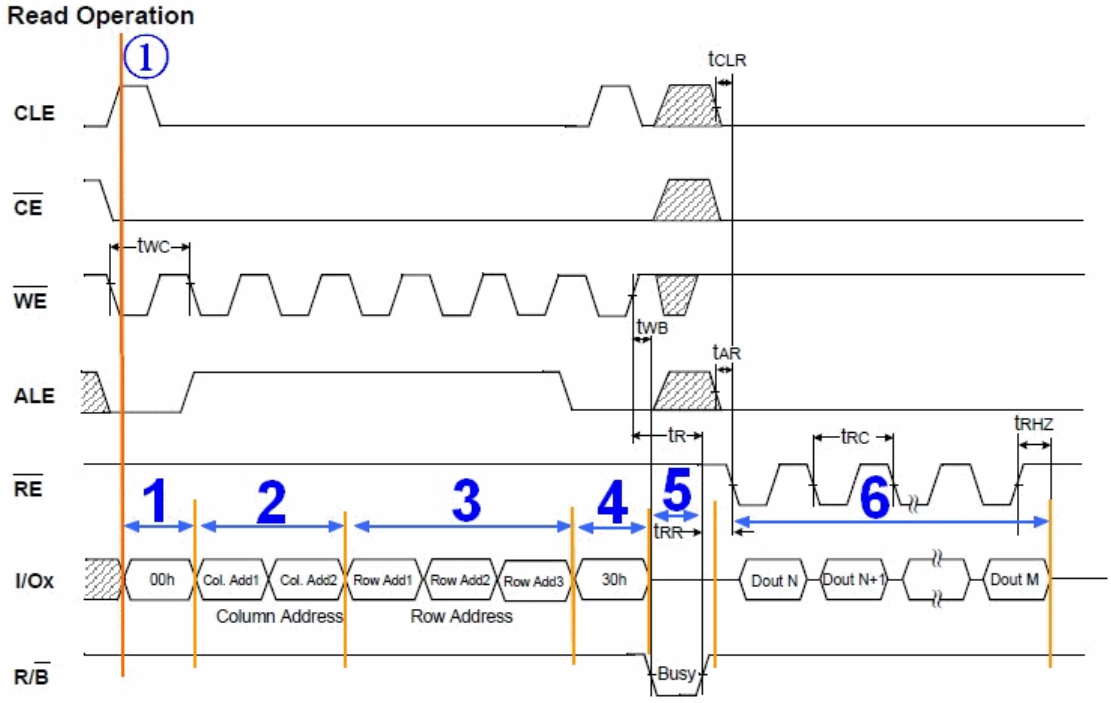
2.3.3 读操作的硬件到软件的映射

下面这部分主要介绍一下，关于硬件的设计和规范，是如何映射到具体的软件实现的，看了这部分内容之后，你对如何根据硬件的规范去用软件代码实现对应的功能，就有了大概的了解了，然后去实现对应的某硬件的驱动，就有了大概的脉络了。

关于硬件部分的细节，前面其实已经介绍过了，但是为了方便说明，此处还是以读操作为例去讲解硬件到软件是如何映射的。

再次贴出上面的那个图：

图表 18 Nand Flash 数据读取操作的时序图



对于上面的从①到⑥，每个阶段所表示的含义，再简单解释一下：

①：此阶段，是读命令第一个周期，发送的命令为 0x00。
②：此阶段，依次发送列地址，关于这些行地址，列地址等是如何计算出来的，后面的内容会有详细解释。
③：此阶段是发送对应的行地址
④：此阶段是发送读命令第二周期 2nd cycle 所对应的命令，0x30
⑤：此阶段是等待时间，等待 Nand Flash 硬件上准备好对应的数据，以便后续读出。
⑥：此阶段，就是一点点地把所需要的数据读出来。

上面的是内容，说的是硬件是如何设计的，而这硬件的设计，即硬件的逻辑时序是如何规定的，对应的软件实现，也就要如何实现。不过可以看出的是，其中很多步骤，比如步骤 1 和步骤 4，那都是固定的，而且，即使其中的步骤 2 和步骤 3，即使是不同厂家和不同的 Nand Flash 芯片，除了要写入的列地址和行地址可能不同之外，也都是逻辑一样的，同样地，步骤 5 和 6，也都是一样的，唯一不同的，是每家不同的 Nand Flash 控制器不同，所以具体到步骤 6 的时候，去读出数据的方式不同，所以，那一部分肯定是你要实现 Nand Flash 驱动的时候要自己实现的，而对应的其他几个公有的步骤呢，就有了 Linux 的 MTD 层帮你实现好了，所以，下面就来介绍一下，关于读取一个 Nand Flash 的页 Page，Linux 的 MTD 层，是如何具体的帮你实现的：

关于 Nand Flash 的读操作，即读取一页的数据，这样的读数据的操作，很明显，是从上层文件系统传递过来的，其细节我们在此忽略，但是要知道，上层读取数据的请求，传递到了 MTD 这一层，其入口是哪个函数，然后我们才能继续往下面分析细节。

关于下面所要的介绍的代码，如果没有明确指出，都是位于此文件：
代码位置： **\drivers\mtd\nand\nand_base.c**

MTD 读取数据的入口是 nand_read，然后调用 nand_do_read_ops，此函数主体如下：

```
static int nand_do_read_ops(struct mtd_info *mtd, loff_t from,
struct mtd_oob_ops *ops)
{
  。。。。。。
  while(1) {
        。。。
      /* ㈠：要读取数据，肯定是要先发送对应的读页（read page）的命令 */
        chip- >cmdfunc(mtd, NAND_CMD_READ0, 0x00, page);
        。。。
      /* ㈡:发送完命令,接着就可以去调用read_page函数读取对应的数据了 */
        ret = chip- >ecc.read_page(mtd, chip, bufpoi,
        buf += bytes;
        。。。

      readlen - = bytes;
      if (! readlen)
      break;
      / * For subsequent reads align to page boundary. */
      col = 0;
      / * Increment page address */
      realpage++;
      page = realpage & chip- >pagemask;
      。。。
      }
      。。。
  } ? end while 1 ?

  。。。。。。
  return mtd- >ecc_stats.corrected - stats.corrected ? - EUCLEAN : 0;
} ? end nand_do_read_ops ?
```

对于上述中步骤㈠的函数 cmdfunc，一般来说可以不用自己的驱动中实现，而直接使用 MTD 层提供的已有的函数，nand_command_lp，其细节如下：

```
static void nand_command_lp(struct mtd_info *mtd, unsigned int command,
int column, int page_addr)
{
  。。。
  / * Command latch cycle */
  /* 此处就是就是发送读命令的第一个周期1st Cycle的命令，即0x00，对应着上
  述步骤中的① */
```

```
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;
  / * Serially input address */
  if (column ! = - 1) {
    / * Adjust columns for 16 bit buswidth */
    if (chip- >options & NAND_BUSWIDTH_16)
      column >>= 1;
/* 接下来是发送两个column，列地址，对应着上述步骤中的② */
    chip- >cmd_ctrl(mtd, column, ctrl);
    ctrl &= ~NAND_CTRL_CHANGE;
    chip- >cmd_ctrl(mtd, column >> 8, ctrl);
  }
  if (page_addr ! = - 1) {
/* 然后发送三个row行地址，对应着上述步骤中的③ */
    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);
  }
} ? end if column! =- 1| | page_addr... ?
chip- >cmd_ctrl(mtd, NAND_CMD_NONE, NAND_NCE |
NAND_CTRL_CHANGE);
/ *
* program and erase have their own busy handlers
* status, sequential in, and deplete1 need no delay
*/
switch (command) {

  。 。 。
case NAND_CMD_READ0:
/* 接下来发送读命令的第二个周期2nd Cycle的命令，即0x30，对应着上述步骤
中的④ */
  chip- >cmd_ctrl(mtd, NAND_CMD_READSTART,
  NAND_NCE | NAND_CLE | NAND_CTRL_CHANGE);
  chip- >cmd_ctrl(mtd, NAND_CMD_NONE,
  NAND_NCE | NAND_CTRL_CHANGE);
  / * This applies to read commands */
default:
  。 。 。
} ? end switch command ?
```

```
  / * Apply this short delay always to ensure that we do wait tWB in
  * any case on any machine. */
  /* 此处是对应着④中的tWB的等待时间*/
  ndelay(100);

  /* 接下来就是要等待一定的时间，使得Nand Flash硬件上准备好数据，以供你之后读取，即对应着步骤⑤ */
  nand_wait_ready(mtd);
} ? end nand_command_lp ?
```

对于之前的步骤㈢的函数 read_page，一般来说也可以不用自己的驱动中实现，而直接使用 MTD 层提供的已有的函数，nand_read_page_hwecc，该函数所要实现的功能，正是上面余下没介绍的步骤⑥，即一点点的读出我们要的数据：

```
static int nand_read_page_hwecc(struct mtd_info *mtd, struct nand_chip
*chip,
uint8_t *buf, int page)
{
。。。
  for (i = 0; eccsteps; eccsteps- - , i += eccbytes, p += eccsize) {
    chip- >ecc.hwctl(mtd, NAND_ECC_READ);

  /* 真正的数据读取，就是下面这个read_buf函数了 */
    chip- >read_buf(mtd, p, eccsize);

    chip- >ecc.calculate(mtd, p, &ecc_calc[i]);
  }


  。。。
} ? end nand_read_page_hwecc ?
```

上面的 read_buf，就是真正的去读取数据的函数了，由于不同的 Nand Flash controller 控制器所实现的方式不同，所以这个函数必须在你的 Nand Flash 驱动中实现，即 MTD 层，能帮我们实现的都实现了，不能实现的，那肯定要你的驱动自己实现。

对于我们这里的 s3c2410 的例子来说，就是 s3c2410_nand_read_buf：

文件位置：**\drivers\mtd\nand\s3c2410.c**

```
static void s3c2410_nand_read_buf(struct mtd_info *mtd, u_char *buf, int len)
{
  struct nand_chip *this = mtd->priv;
  /* 到真正的地址去读取数据 */
  readsb(this->IO_ADDR_R, buf, len);
}
```

可以看出，此处的实现相当地的简单，就是读取对应的 IO 的地址，然后就可以把数据读出

来就可以了。不过，要注意的是，并不是所有的驱动都是这么简单，具体情况则是不同的 Nand Flash 控制器对应不同实现方法。

至此，关于整个的 Nand Flash 的读取一页的数据的操作，是如何将硬件的逻辑时序图，映射到对应的软件的实现的，就已经介绍完了。而看懂了这个过程，你才会更加明白，原来 MTD 层，已经帮助我们实现了很多很多通用的操作所对应的软件部分，而只需要我们实现剩下那些和具体硬件相关的操作的函数，就可以了，可以说大大减轻了驱动开发者的工作量。因为，如果没了 MTD 层，那么上面那么多的函数，几乎都要我们自己实现，单单是代码量，就很庞大，而且再加上写完代码后的驱动测试功能是否正常，使得整个驱动开发，变得难的多得多。

2.3.4 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。
在 Nand Flash 驱动加载后，第一步，就是去调用对应的 init 函数，s3c2410_nand_init,去将在 Nand Flash 驱动注册到 Linux 驱动框架中。
驱动本身，真正开始，是从 probe 函数，s3c2410_nand_probe->s3c24xx_nand_probe,
在 probe 过程中，去用 clk_enable 打开 Nand Flash 控制器的 clock 时钟，用 request_mem_region 去申请驱动所需要的一些内存等相关资源。然后，在 s3c2410_nand_inithw 中，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。
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 falsh 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);
/* scan tail，从名字就可以看出来，是扫描的后一阶段，此时，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++;
    }
```

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。

3 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。
估计很多人就会问了，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。

3.1 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。

3.2 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,
                                   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的 ECC 的话，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;

                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;
    }

    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_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。

（6）**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。

（7）**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。

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。

4 引用文章

1. Brief Intro of Nand Flash
http://hi.baidu.com/serial_story/blog/item/3f1635d1dc041cd7562c84a1.html

2. Samsung 的型号为 K9G8G08U0M 的 Nand Flash 的数据手册
要下载数据手册，可以去这里介绍的网站下载：
samsung 4K pagesize SLC Nand Flash K9F8G08U0M datasheet + 推荐一个 datasheet 搜索的网站
http://hi.baidu.com/serial_story/blog/item/7f25a03def1de309bba167c8.html

3. Nand Falsh Read Operation
http://hi.baidu.com/serial_story/blog/item/f06db3546eced11a3b29356c.html

4. Memory Technology Device (MTD) Subsystem for Linux.
http://www.linux-mtd.infradead.org/index.html

5. NAND 和 NOR 的比较
http://hi.baidu.com/serial_story/blog/item/c669991efe491e1b403417e5.html

6. 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

7. **芯片内执行(XIP, eXecute In Place)**:
http://hi.baidu.com/serial_story/blog/item/adb20a2a3f8ffe3c5243c1df.html

8. ONFI 官网
http://onfi.org/specifications/

9. ONFI 2.2 的 Spec
http://onfi.org/wp-content/uploads/2009/02/ONFI%202_2%20Gold.pdf

10. ONFI Block Abstracted Nand 的 Spec
http://onfi.org/wp-content/uploads/2009/02/BA_NAND_rev_1_1_Gold.pdf

11. LBA Nand Flash 的简介
http://www.toshiba-components.com/prpdf/5678E.pdf

12. Toshiba TH58NS512DC
http://datasheet.elcodis.com/pdf/11/23/112371/th58ns512dc-to51y.pdf

13. Samsung K9F5608U0B

http://hitmen.c02.at/files/docs/psp/ds_k9f5608u0b_rev13.pdf

14. NAND 接口规格之争 东芝三星 LBA 将对抗英特尔 ONFI
http://stor-age.zdnet.com.cn/stor-age/2007/0704/416025.shtml

15. lba-core.c
http://linux-fsl-imx51.sourcearchive.com/documentation/2.6.31-605.7/lba-core_8c-source.html

16. 【下载】4 bits 纠错的 BCH 源代码（常用于 MLC nand Flash 的 ECC 算法）
http://hi.baidu.com/serial_story/blog/item/c4e826d8988aa13932fa1cc6.html