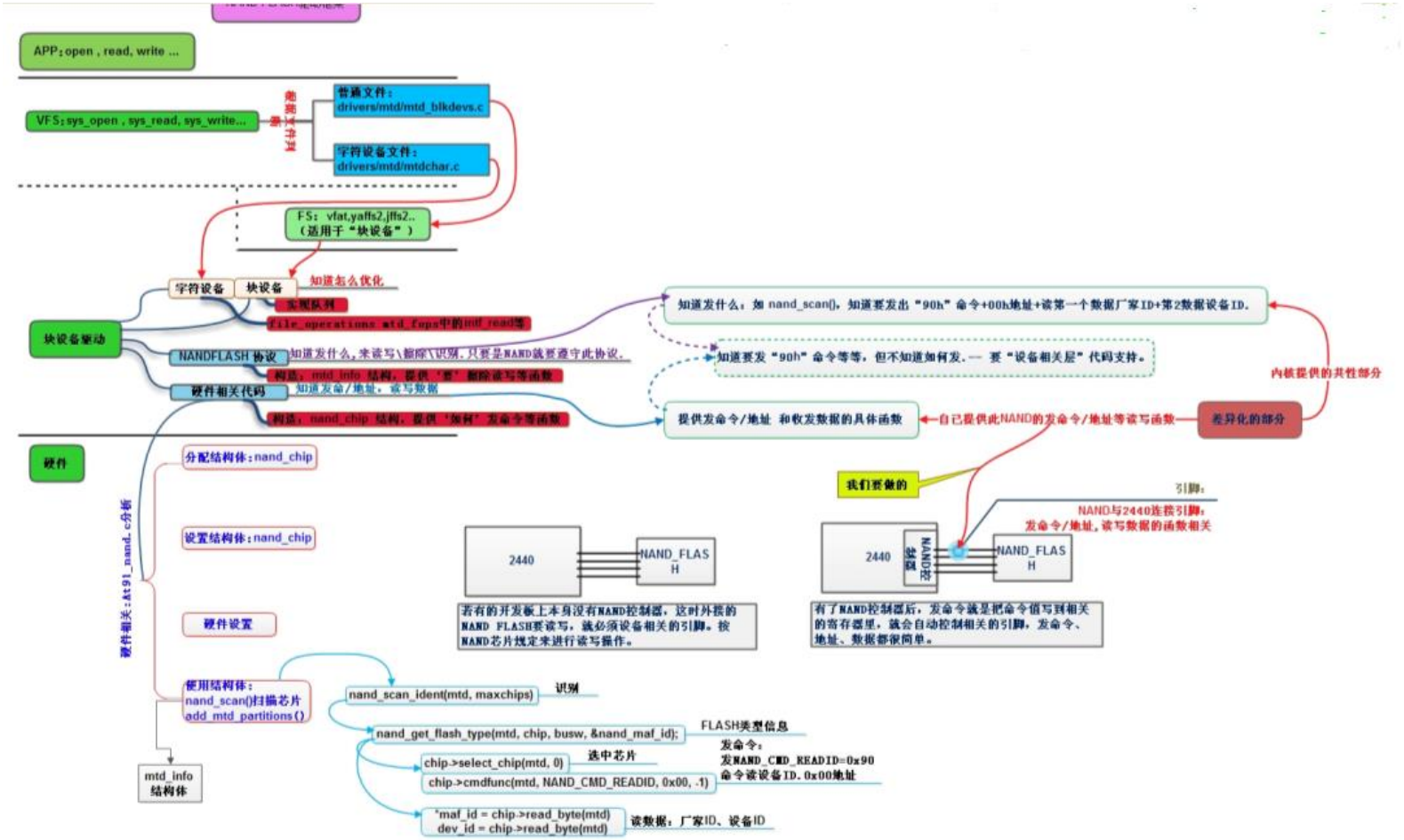
网名“鱼树”的学员聂龙浩，

学习“韦东山 Linux 视频第 2 期”时所写的笔记很详细，供大家参考。

也许有错漏，请自行分辨。

目录

NandFlash 框架图



NAND 原理及硬件操作：

APP:　　open,read,write "1.txt"
---------- 文件读写
文件系统：vfat,ext2,ext3,yaffs　　（把文件的读写转换成对扇区的读写）
----------ll_rw_block---------- 扇区读写
　　块设备驱动程序

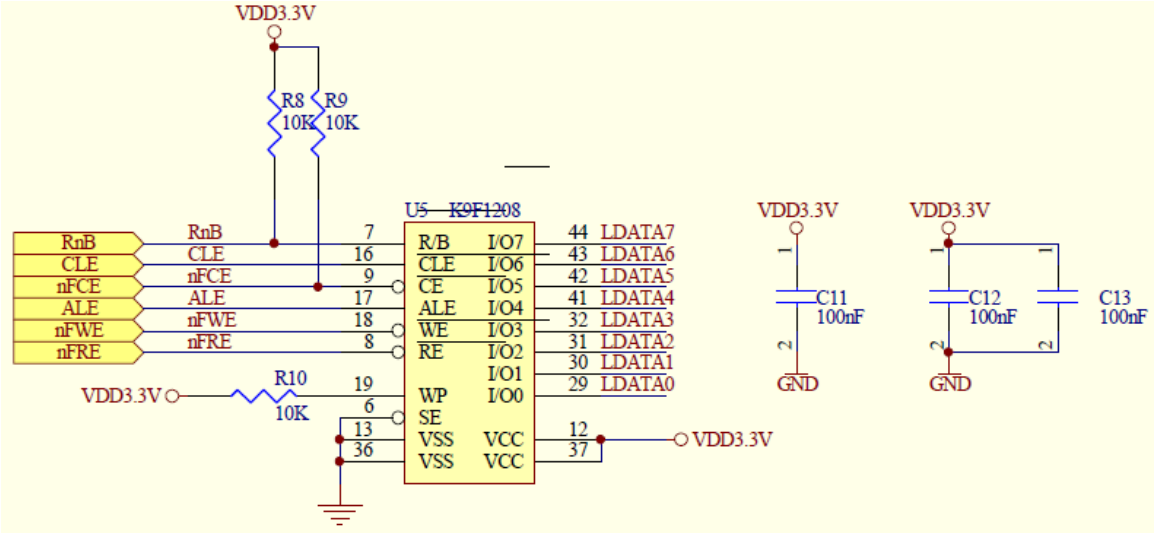
硬件：硬盘、FLASH

写块设置驱动程序步骤：

1，以面向对象的思想分配 gendisk 结构体。用 alloc_disk 函数。
2，设置 gendisk 结构体。
①，分配/设置一个队列：request_queue_t. （提供读写能力）用 blk_init_queue 函数。
②，设置 gendisk 其他信息。（提供磁盘属性：磁盘容量，扇区大小等）
3，注册 gendisk 结构体。用 add_disk 函数。

操作不用我们关心，格式化、读写文件等都是由“文件系统”这一层将文件的读写转换成对扇区的读写的。调用“ll_rw_block”会把读写放到你的队列中去。会调用你“队列请求处理函数”来处理。只要你写好“队列请求处理”函数即可。

一,NANA 接线图 上相关的 引脚含义：



和 2440 有接数据线“LDATA0 ~ 7”8 个数据线。

NAND FLASH 是一个存储芯片
那么：这样的操作很合理"读地址 A 的数据，把数据 B 写到地址 A"

A:地址线，数据线，命令 与 ALE,CLE 信号
问 1. 原理图上 NAND FLASH 和 S3C2440 之间只有数据线(LDATA0~7),没有看到地址引脚。怎么传输地址（如何将地址信号告诉 NAND）？答案：复用。
答 1. 在 DATA0～DATA7 上既传输数据，又传输地址。用一个信号 ALE 分辨是“地址”还是“信号”，当 ALE 为高电平时传输的是
地址。当 ALE 为低电平是传输的是数据。

在这个 DATA0~7 的数据线上，是否只传输数据或是地址呢？看 NAND 的芯片手册《K9F2G08U0A.pdf》上得知对 NAND 的操作还要发出命令。

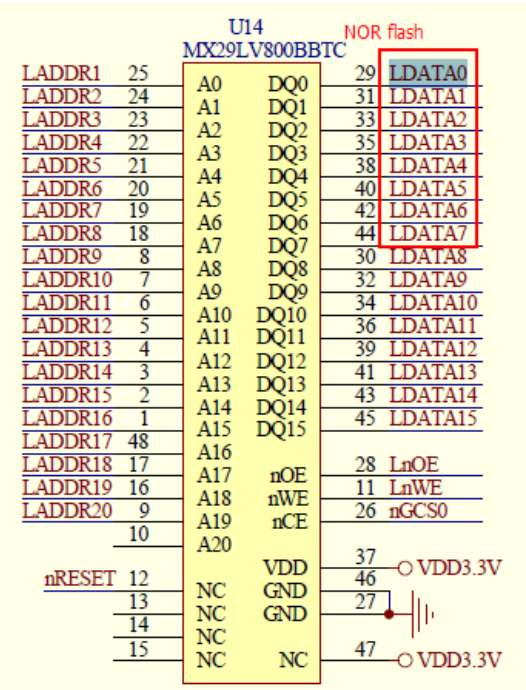
Table 1. Command Sets

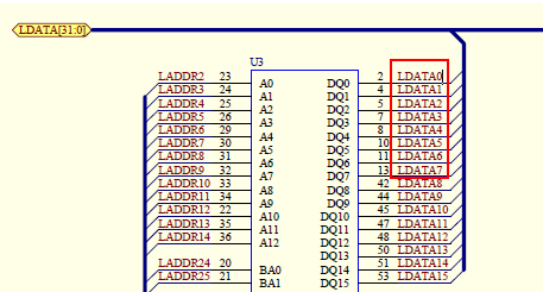
Function	1st Cycle	2nd Cycle	Acceptable Command during Busy
Read	00h	30h	
Read for Copy Back	00h	35h	
Read ID	90h	-	
Reset	FFh	-	O
Page Program	80h	10h	
Copy-Back Program	85h	10h	
Block Erase	60h	D0h	
Random Data Input(1)	85h	-	
Random Data Output(1)	05h	E0h	
Read Status	70h		O
Read EDC Status(2)	7Bh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

NOTE : 1. Random Data Input/Output can be executed in a page.
2. Read EDC Status is only available on Copy Back operation.

如上，想“read”读时，要先发出命令“00h”.等等操作都要先发出“命令”。
问 2. 从 NAND FLASH 芯片手册可知，要操作 NAND FLASH 需要先发出命令，只有 8 条 DATA0~7 的数据线 怎么传入命令？
答 2. 在 DATA0～DATA7 上既传输数据，又传输地址，也传输命令。
当 ALE 为高电平时传输的是地址，
当 CLE 为高电平时传输的是命令，
当 ALE 和 CLE 都为低电平时传输的是数据。
以上便清楚了“CLE”、“ALE”。

B:片选信号“CE”
原理图上还有个“CE - 片选信号”，如何理解“片选信号”：
查 DATA0 接有哪里：





LDATA0 还有接到“NOR flash”上。
LDATA 0 还有接到“内存”上。
其实 LDATA0 还有接到如网卡等设备上。

问 3. 数据线既接到 NAND FLASH，也接到 NOR FLASH，还接到 SDRAM、DM9000 等等
那么怎么避免干扰？
答 3. 这些设备，要访问之前必须"选中"，没有选中的芯片不会工作，相当于没接一样。
要“选中”这就是它们都有“片选”
信号。
比如 NAND，要让引脚“nFCE”变成低电平选中。内存要选中，则片选引脚“nSCS”要变成低电平。NOR 是“nCE”片选信号变成低电平选中。

C：RnB - 状态引脚
问 4. 假设烧写 NAND FLASH，把命令、地址、数据发给它之后，
NAND FLASH 肯定不可能瞬间完成烧写的，
怎么判断烧写完成？
答 4. 通过状态引脚 RnB 来判断：它为高电平表示就绪，它为低电平表示正忙。

D,nFRE nFWE 这是读写信号。

二，如何操作 NAND FLASH:

问 5. 怎么操作 NAND FLASH 呢？

答 5. 根据 NAND FLASH 的芯片手册，一般的过程是：

发出命令

发出地址

发出数据 或 读数据

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(3)	80h---11h	81h---10h	
Copy-Back Program	85h	10h	
Two-Plane Copy-Back Program(3)	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

NOTE : 1. Random Data Input/Output can be executed in a page.

2. Read EDC Status is only available on Copy Back operation.

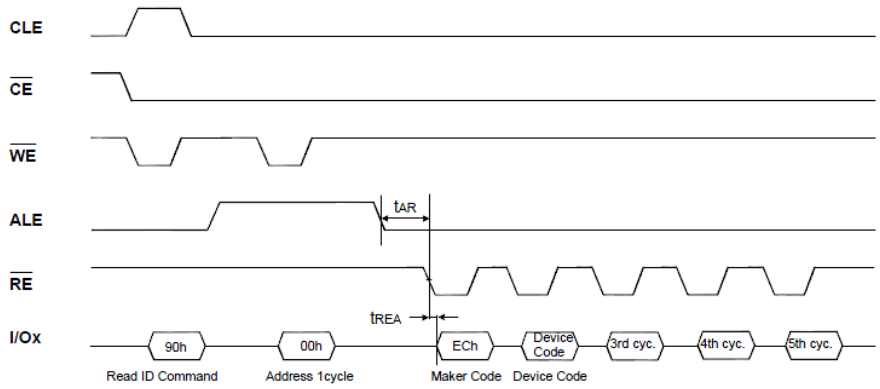
3. Any command between 11h and 81h is prohibited except 70h and FFh.

4. K9F2G08R0A does not support Two-Plane operation.

（命令表格）

看“命令”不容易看，就看“时序图”，比如命令表格中的“read ID”.所谓“时序图”，就是“横轴”是时间。纵轴上各“信号”为不同电平时，横轴上相同“列--纵轴”所对应的功能动作。

“Read ID Operation”：



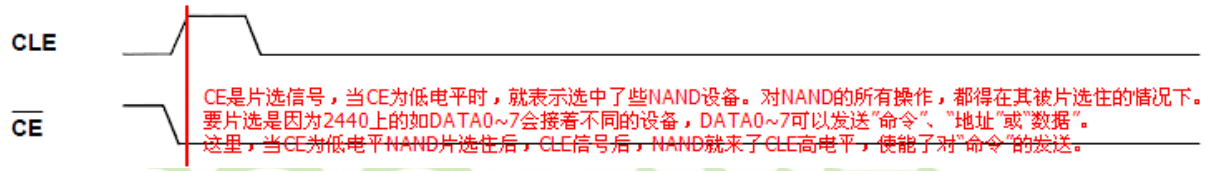
Device	Device Code (2nd Cycle)	3rd Cycle	4th Cycle	5th Cycle
K9F1G08U0B	F1h	00h	95h	40h

每个 NAND FLASH，都内嵌了一些 ID，如厂家 ID，设备 ID。
“时序图”从左往右看，横轴是时间。纵方向则是“一列一列”的看。

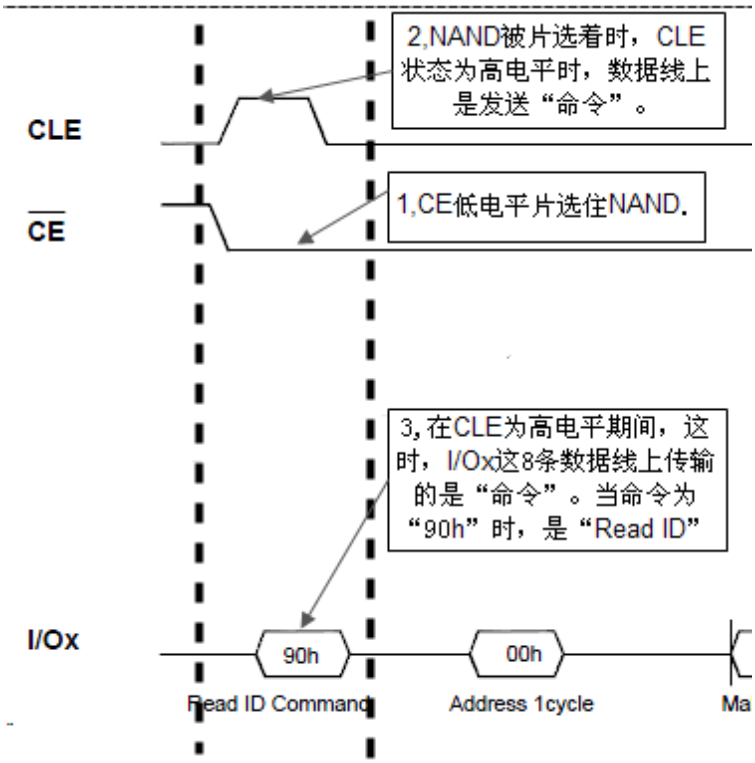
CE:片选，低电平时为选中芯片。这里是选中 NAND 芯片（因为 8 条 DATA0~7 接到不同设备，要用片选来选中某设备）。
CLE:当 CLE 为高电平时传输的是“命令”。
当 ALE 为高电平时传输的是地址。
WE：写脉冲。
RE：读脉冲。
I/Ox:是 8 条数据线。在 DATA0～DATA7 上既传输数据，又传输地址，也传输命令。当 ALE 和 CLE 都为低电平时传输的是数据。

“读 ID 操作”：

0，CE 为低电平，选中此 NAND 设备：

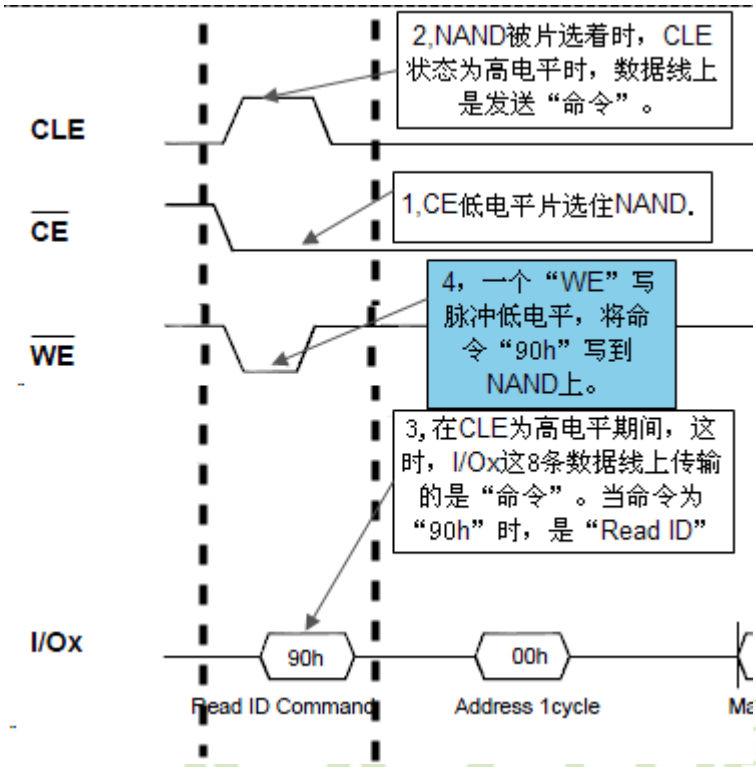


1,CLE 为高电平，在数据线“I/Ox”上输出“90h”。
发出“90h”命令，就是在这个 8 条数据线上“I/Ox”发出“90h”值。如何知道它是“命令”？则往纵轴看，"90h"这一列上面“CLE”为“高电平”了。（在上面的 NAND 与 2440 连接引脚中知道：当 CLE 为高电平时传输的是命令。）



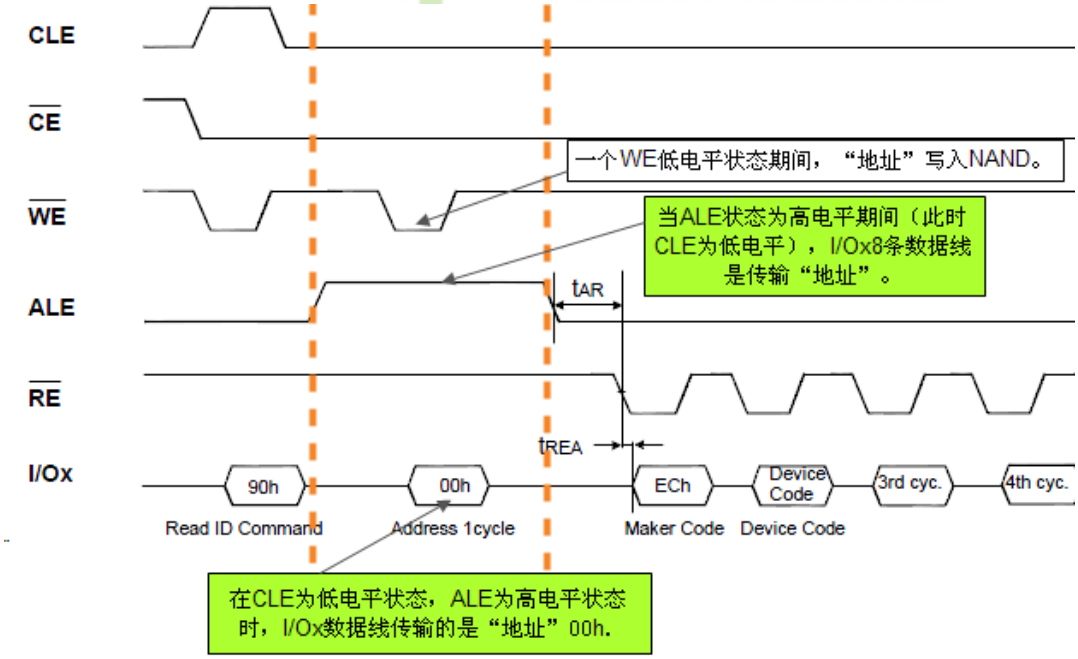
2，再给一个“写脉冲 - WE”。

在“WE”的上升沿，NAND 就把“I/Ox”数据线上输入的“90h”存进来。



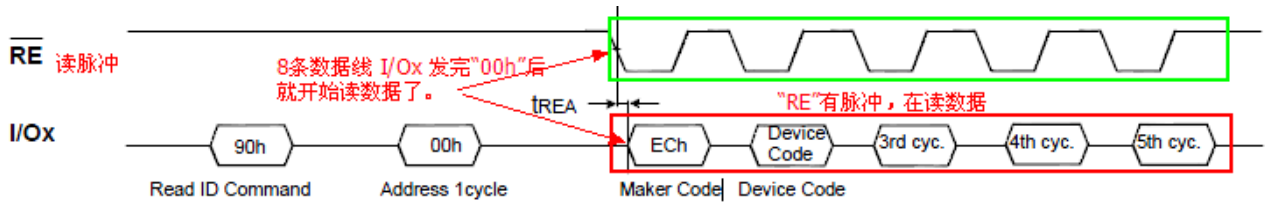
3，"I/Ox"数据线上接着发出一个“0 地址 - 00h”(Address 1cycle).

然后就可以读数据了。



4，读数据，从"I/Ox"这 8 条数据线上读到第一个数据是“ECh”。

第二个值是一个“设备 ID - Device Code”



2440 内部集成了一个 NAND 控制器

	NAND FLASH	S3C2440
发命令	选中芯片	
	CLE 设为高电平	NFCMMD=命令值 （命令值写 NFCMMD 寄存器，会自动完成左边步骤）
	在 DATA0~DATA7 上输出命令值	
	发出一个写脉冲（WE）	
发地址	选中芯片	NFADDR=地址值
	ALE 设为高电平	
	在 DATA0~DATA7 上输出地址值	
	发出一个写脉冲	
发数据	选中芯片	NFDATA=数据值
	ALE,CLE 设为低电平	
	在 DATA0~DATA7 上输出数据值	
	发出一个写脉冲	
读数据	选中芯片	val=NFDATA（读 NFDATA 寄存器）
	发出读脉冲	
	读 DATA0~DATA7 的数据	

用 UBOOT 来体验 NAND FLASH 的操作：

1. 读 ID

	S3C2440	u-boot
选中	NFCONT 的 bit1 设为 0	md.l 0x4E000004 1; mw.l 0x4E000004 1
发出命令 0x90	NFCMMD=0x90	mw.b 0x4E000008 0x90
发出地址 0x00	NFADDR=0x00	mw.b 0x4E00000C 0x00
读数据得到 0xEC	val=NFDATA	md.b 0x4E000010 1
读数据得到 device code 0xda	val=NFDATA	md.b 0x4E000010 1
退出读 ID 的状态	NFCMMD=0xff	mw.b 0x4E000008 0xff

u-boot 中有些命令可以读内存，“md.l 0x4E000004 1”是指从地址“0x4E000004 - NFCONT 地址”读“1”次。不写次数时，会读 16 次，即是读诸如这种间格的地址“0x4E000010 -> 0x4E000011 -> .. ->0x4E00001F”.

“md” ：显示内存。需要一个地址。[一个可选的数字]。

```
OpenJTAG> help md
md [.b, .w, .l] address [# of objects]
    - memory display
```

md 显示内存：b以字节，w以2字，l以4字。

因为寄存器是 4 字节 长度，故这里以“md.l”。从 u-boot 读寄存器 NFCONT（0x4E000004）的结果：

```
OpenJTAG> md.l 0x4E000004 1
4e000004: 00000003    ....
```

读出是“0x3”。查 2440 手册：

CONTROL REGISTER

Register	Address	R/W	Description	Reset Value
NFCONT	0x4E000004	R/W	NAND flash control register	0x0384

Reg_Nce ： NAND 的片选域。此位为“0”时，使能芯片选择。

Reg_nCE	[1]	NAND Flash Memory nFCE signal control 0: Force nFCE to low (Enable chip select) 1: Force nFCE to high (Disable chip select) **Note:** During boot time, it is controlled automatically. This value is only valid while MODE bit is 1	1

上面通过“md.l 0x4E000004 1”读出的结果是“0x3”，进制即“10”，即“Reg_nCE”位此时为“1”，是没选中的。要操作 NAND，则要把“NFCONT”寄存器的“Reg_nCE”位设置为“0”。即对寄存器“NFCONT”写入“0x1”也就是把“Reg_nCE”位置为“0”了，片选选中。

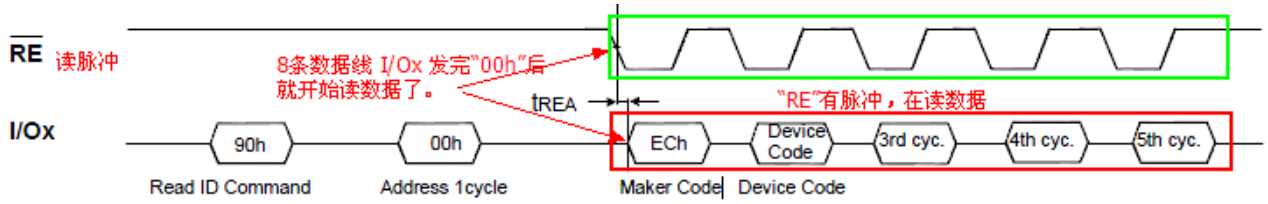
“mw”：写内存。

```
OpenJTAG> help mw
mw [.b, .w, .l] address value [count]
    - write memory
```

```
OpenJTAG> md.l 0x4E000004 1
4e000004: 00000003    ....
OpenJTAG> mw.l 0x4E000004  1
OpenJTAG> mw.b 0x4E000008 0x90
OpenJTAG> mw.b 0x4E00000C 0x00
OpenJTAG> md.b 0x4E000010 1
4e000010: ec     .
OpenJTAG> md.b 0x4E000010 1
4e000010: da     .
OpenJTAG>
```

选中NAND片选
发90h命令
发00h地址
读1个数据
再读1个数据

以上先“mw.l 0x4E000004 1”选中了 NAND .写数据时是“ec”和“da”.看 NAND 芯片手册的时序图：



读的第一个数据是“EC”.

Device	Device Code (2nd Cycle)	3rd Cycle	4th Cycle	5th Cycle
K9F2G08R0A	AAh	00h	15h	44h
K9F2G08U0A	DAh	10h	95h	44h

读的第二个数据是“Device Code”是“da”.

2. 读内容：读 NAND 中 0 地址 的数据

使用 UBOOT 命令:

nand dump 0

Page 00000000 dump:

17 00 00 ea 14 f0 9f e5 14 f0 9f e5 14 f0 9f e5

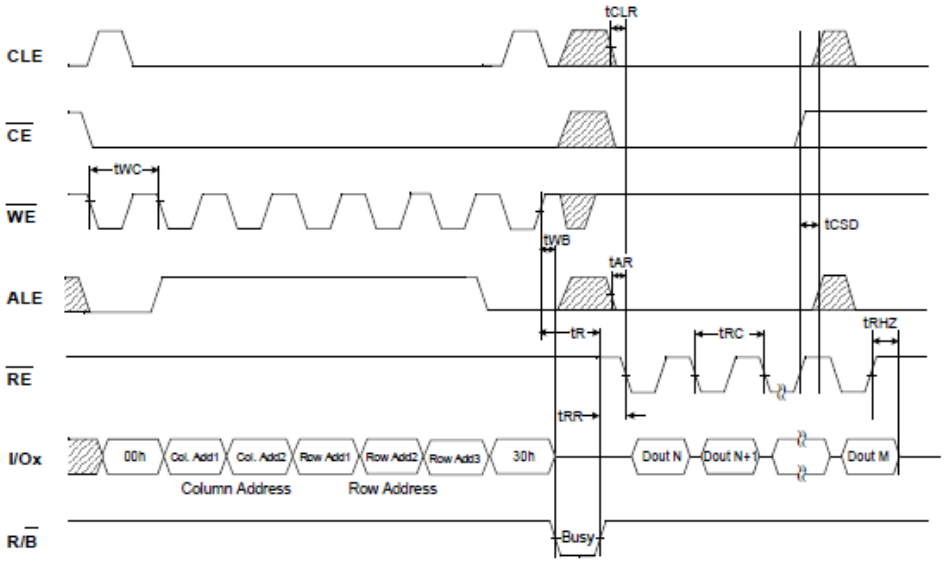
```
OpenJTAG> nand dump 0
Page 00000000 dump:
        17 00 00 ea 14 f0 9f e5  14 f0 9f e5 14 f0 9f e5
        14 f0 9f e5 14 f0 9f e5  14 f0 9f e5 14 f0 9f e5
        60 01 f8 33 c0 01 f8 33  20 02 f8 33 80 02 f8 33
        e0 02 f8 33 00 04 f8 33  20 04 f8 33 ef be ad de
        00 00 f8 33 00 00 f8 33  ac 06 fb 33 04 79 fb 33
        de c0 ad 0b de c0 ad 0b  00 00 00 00 de c0 ad 0b
        de c0 ad 0b 00 00 0f e1  1f 00 c0 e3 d3 00 80 e3
        00 f0 29 e1 53 04 a0 e3  00 10 a0 e3 00 10 80 e5
        00 10 e0 e3 d4 03 9f e5  00 10 80 e5 d0 13 9f e5
        d0 03 9f e5 00 10 80 e5  a0 00 4f e2 64 10 1f e5
```

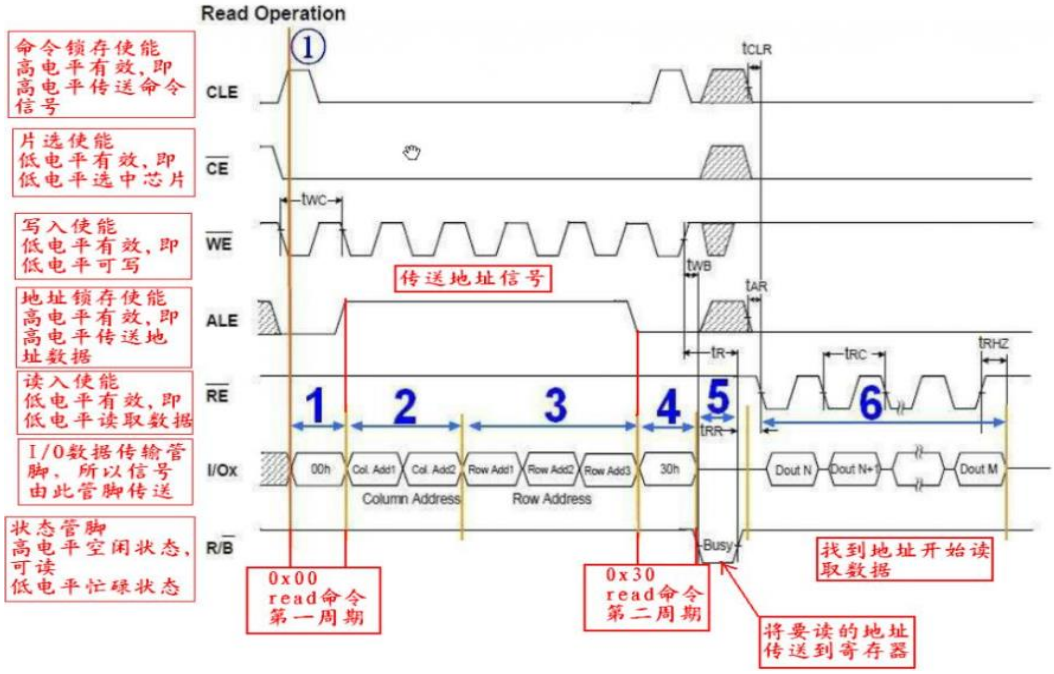
上面是 UBOOT 中提供的读 0 地址内容的命令。

Nand dump 实现了上面发命令，发地址等操作。若想通过操作寄存器，如何读出 nand dump 读到的内容。

读的时序图：Read Operation

Read	00h	30h	读 命令，第1个周期发出 00h 命令，第1个周期发 30h





I/Ox 从“列”轴上看到“CLE”高电平（CLE 为高电平时，数据线发“命令”）发出“00h”命令，之后“CLE”成为“低电平”，而“ALE”变成高电平（ALE 为高电平时，数据线发送地址），所以"I/Ox"接着发出地址“Col. Add1、Col. Add2、Row Add1、Row Add2、Row Add3”5 个地址“0”.

这里需要发 5 个地址，因为这个 NAND 是 256 M。要多少个地址位。

若“地址线”只有一条，则只能表示“1”或“0”，即表示 2 个地址。

若“地址线”有两条，则能表示 2 次方 “00”“01”“10”“11”4 个地址。

当 容量为 256M 时，“地址线”要多少条？

2^n =256M

= $2^8 + 2^{20}$

= 2^{28}

M 是 2 的 20 次方，256M 至少需要 2 的 28 条，至少需要 28 位的数据表示这个地址值。28 除以 8 差不多为 4 个周期，为了兼容更大容量的 FLASH.

	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	*L	*L	*L	*L	*L	*L	*L	Row Address

NOTE : Column Address : Starting Address of the Register.
* L must be set to "Low".
* The device ignores any additional input of address cycles than required.

对 NAND 规定了要发出 5 个周期 的地址。
发完 5 个周期的地址命令后，再发“30h”命令（确定）命令。接着就可以读了。

	S3C2440	u-boot
选中	NFCONT的bit1设为0	md.l 0x4E000004 1; mw.l 0x4E000004 1
发出命令0x00	NFCMMD=0x00	mw.b 0x4E000008 0x00
发出地址0x00	NFADDR=0x00	mw.b 0x4E00000C 0x00
发出地址0x00	NFADDR=0x00	mw.b 0x4E00000C 0x00
发出地址0x00	NFADDR=0x00	mw.b 0x4E00000C 0x00
发出地址0x00	NFADDR=0x00	mw.b 0x4E00000C 0x00
发出地址0x00	NFADDR=0x00	mw.b 0x4E00000C 0x00
发出命令0x30	NFCMMD=0x30	mw.b 0x4E000008 0x30
读数据得到0x17	val=NFDATA	md.b 0x4E000010 1
读数据得到0x00	val=NFDATA	md.b 0x4E000010 1
读数据得到0x00	val=NFDATA	md.b 0x4E000010 1
读数据得到0xea	val=NFDATA	md.b 0x4E000010 1
退出读状态	NFCMMD=0xff	mw.b 0x4E000008 0xff

```
OpenJTAG>  mw.l 0x4E000004  1      片选NAND
OpenJTAG> mw.b 0x4E000008 0x00     发读命令
OpenJTAG> mw.b 0x4E00000C 0x00
OpenJTAG> mw.b 0x4E00000C 0x00
OpenJTAG> mw.b 0x4E00000C 0x00     发5个周期地址命令
OpenJTAG> mw.b 0x4E00000C 0x00
OpenJTAG> mw.b 0x4E00000C 0x00
OpenJTAG> mw.b 0x4E000008 0x30     发确定命令
OpenJTAG> md.b 0x4E000010 1        读数据
4e000010: 17     .
OpenJTAG> md.b 0x4E000010 1
4e000010: 00     .
OpenJTAG> md.b 0x4E000010 1
4e000010: 00     .
OpenJTAG> md.b 0x4E000010 1
4e000010: ea     .
OpenJTAG> md.b 0x4E000010 1
4e000010: 14     .
OpenJTAG> md.b 0x4E000010 1
4e000010: f0     .
OpenJTAG>
```

读到的结果和“nand dump”结果一样。

NAND 驱动编写：

一，入口函数框架：

```
//入口函数
static int s3c_nand_init(void)
{
    //1,分配 nand_chip 结构
    s3c_nand = kzalloc(sizeof(struct nand_chip), GFP_KERNEL);

    //2，设置

    //3,硬件相关的操作

    //4,使用:nand_scan (联系 nand_chip 结构与 mtd_info 结构)
        //4.1.2,从nand_scan原型知要mtd_info结构.分配 mtd_info 结构
    s3c_mtd = kzalloc(sizeof(struct mtd_info), GFP_KERNEL);
        //4.2,联系 nand_chip 结构与 mtd_info 结构
    s3c_mtd->owner = THIS_MODULE;
    s3c_mtd->priv = s3c_nand;    //mtd_info结构的私有数据等于nand_chip结构体
    nand_scan(s3c_mtd, 1);       //参2最多芯片个数我们为1个。nand_scan 扫描、识别、构造mtd_info结构体(读写擦除函数)。


    //5,增加mtd分区:add_mtd_partitions
    return 0;
} ? end s3c_nand_init ?
```

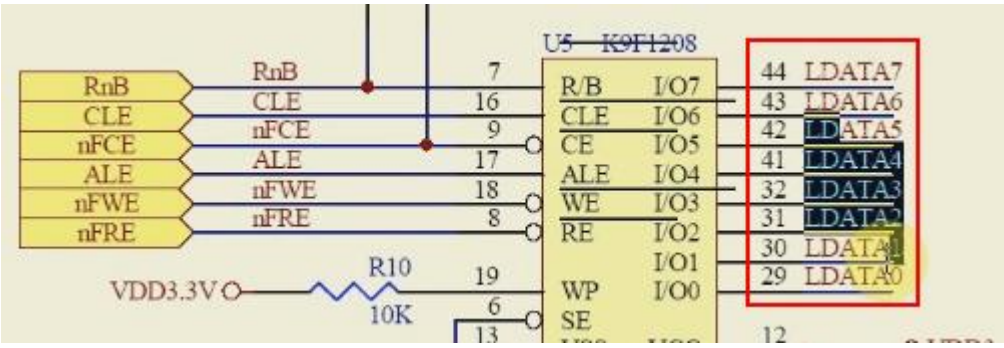
设置“nand_chip”结构就是给“nand_scan()”来使用，这里先不知道如何设置，就先看如何使用，然后再设置。

一，设置 nand_chip 结构：

过程主要是分析“nand_scan”中使用 nand_chip 的情况来分析要如何设置它。

跟踪 nand_scan 函数：

```
nand_scan  // drivers/mtd/nand/nand_base.c 根据nand_chip的底层操作函数识别NAND FLASH，构造mtd_info
    nand_scan_ident
        busw = chip->options & NAND_BUSWIDTH_16; //总线宽度即FLASH为8位还是16位
```



我们实验里的 NAND FLASH 是 8 位（8 条数据线）

nand_set_defaults 下面都是相关函数为空时给一个默认的函数。nandflash 协议这一层

```
if (!chip->select_chip)
chip->select_chip = nand_select_chip; // 默认值不适用

if (chip->cmdfunc == NULL) cmdfunc名字上知是发命令的函数。若为空就给一个默认的nand_command/
chip->cmdfunc = nand_command;
chip->cmd_ctrl(mtd, command, ctrl);
if (!chip->read_byte)
chip->read_byte = nand_read_byte;
readb(chip->IO_ADDR_R);
if (chip->waitfunc == NULL)
chip->waitfunc = nand_wait;
chip->dev_ready
```

假设上面的默认函数我们都可以使用， 接着如下：

```
        nand_get_flash_type：硬件相关代码这一层
           chip->select_chip(mtd, 0); 选中芯片，看看默认的函数中设置的是什么，是否适合我们的平台。
           chip->cmdfunc(mtd, NAND_CMD_READID, 0x00, -1); 即发命令又发0地址
           *maf_id = chip->read_byte(mtd); 读厂家ID
           dev_id = chip->read_byte(mtd);  读设备ID
    nand_scan_tail
                       mtd->erase = nand_erase;
                       mtd->read = nand_read;
                       mtd->write = nand_write;
```

上面是分析其他代码中使用“nand_scan()”时涉及“nand_chip”结构和“mtd_info”结构方面相关要设置的部分。因为我们自己写代码时，要设置这两个结构，但一开始并不知道如何去设置，这样去分析别人的代码里关于“使用”时，nand_scan()中会如何使得这两个结构，从而得出应该如何去设置这两个结构。

从对 nand_scan 的分析，其他有一个“nand_set_default”的默认设置的函数（相当于知道要去做什么才定义这些

默认操作函数：选中，发命令\地址，读数据等<nand flash 协议>）。这个默认设置完成后，就开始去实现（要知道如何具体实现<硬件相关代码>）

假设上面的默认函数我们都可以使用， 接着如下：

1，片选芯片：

chip->select_chip(mtd, 0); 选中芯片，看看默认的函数中设置的是什么，是否适合我们的平台。
则看默认的选择芯片的函数：chip->select_chip = nand_select_chip;在其的实现中，如下，

```
static void nand_select_chip(struct mtd_info *mtd, int chipnr)
{
struct nand_chip *chip = mtd->priv;

switch (chipnr) {
case -1:
chip->cmd_ctrl(mtd, NAND_CMD_NONE, 0 | NAND_CTRL_CHANGE);
break;
case 0: //为0时为第一个芯片，里面什么也没做
break;

default:
BUG();
}
}
```

则 chip->select_chip(mtd, 0); 当形参 2 为“0”时，是"case:0"部分，是什么也没有做。所以这个默认的选择芯片的默认操作并不适合我们。我们要自己写一个“select_chip()”函数。
选中芯片时，要选中 2440 NAND 控制器中的“NFCONT”寄存器“Reg_nCE”bit1 域设置为“0”，片选上 NAND 芯片。

```
s3c_nand->select_chip = s3c2440_select_chip;    //不适合:选择芯片
//2.1，选择芯片(片选)
static void s3c2440_select_chip(struct mtd_info *mtd, int chipnr)
{
if (chipnr == -1)
{
/* 取消选中: NFCONT[1]设为1 */
s3c_nand_regs->nfcont |= (1<<1);
}
else
{
/* 选中: NFCONT[1]设为0 */
s3c_nand_regs->nfcont &= ~(1<<1);
}
}
```

2，发命令/地址、数据：

```
chip->cmdfunc(mtd, NAND_CMD_READID, 0x00, -1);
默认函数为：chip->cmdfunc = nand_command;
void nand_command(struct mtd_info *mtd, unsigned int command, int column, int page_addr)
“column”：页内地址（页内哪一个地址）
```

“page_addr”：页地址

即 NAND FLASH 取地址时，是先看是哪一页，再看这一页里是哪个地址。
-->chip->cmd_ctrl(mtd, readcmd, ctrl)；发命令。参 2 为命令值或地址值，参 3 控制了是发命令还是发地址。
不知道这里是如何实现的“cmd_ctrl”，
则参考“nand_chip->cmd_ctrl = at91_nand_cmd_ctrl;”

```
void at91_nand_cmd_ctrl(struct mtd_info *mtd, int cmd, unsigned int ctrl)
-->if (ctrl & NAND_CLE) //#define NAND_CLE 0x02 。‘与’上ctrl后为真。
writeb(cmd, host->io_base + (1 << host->board->cle)); CLE为高电平时发命令
else
writeb(cmd, host->io_base + (1 << host->board->ale));
```

ALE 为高电平时发地址
从上面知道，默认函数 chip->cmdfunc = nand_command;也是需要构造 nand_chip 结构里的“cmdfunc”函数。只不过从“at91_nand.c”中知道了如何具体实现这个函数而已。

Register	Address	R/W	Description	Reset Value
NFCMMD	0x4E000008	R/W	NAND flash command set register	0x00

NFCMMD	Bit	Description	Initial State
Reserved	[15:8]	Reserved	0x00
NFCMMD	[7:0]	NAND flash memory command value	0x00

ADDRESS REGISTER

Register	Address	R/W	Description	Reset Value
NFADDR	0x4E00000C	R/W	NAND flash address set register	0x0000XX00

```
s3c_nand->cmd_ctrl    = s3c2440_cmd_ctrl;       //默认函数也需要定义:发命令、地址、数据
//2.2,发命令/地址或数据
static void s3c2440_cmd_ctrl(struct mtd_info *mtd, int dat, unsigned int ctrl)
{
if (ctrl & NAND_CLE)
{
/* 发命令: NFCMMD=dat 2440NAND控制器中NFCMMD中发命令*/
s3c_nand_regs->nfcmd = dat;
}
else
{
/* 发地址: NFADDR=dat 2440NAND控制器中NFADDR发地址*/
```

```
s3c_nand_regs->nfaddr = dat;
}
}
```

3，读数据：厂家 ID 和设备 ID

默认函数为：chip->read_byte = busw ? nand_read_byte16 : nand_read_byte;
我们的总线数据位宽是 8 位（8 条数据线），所以看“nand_read_byte”函数。

```
static uint8_t nand_read_byte(struct mtd_info *mtd)
{
struct nand_chip *chip = mtd->priv;
return readb(chip->IO_ADDR_R);
}
```

从默认函数的操作中发现还是要提供：nand_chip->IO_ADDR_R.所以我们得自己写“读数据”的函数。

```
s3c_nand->IO_ADDR_R   = &s3c_nand_regs->nfdata; //需要提供NFDATA寄存器的虚拟地址.:读数据
```

4，写数据：

默认函数为：

```
chip->write_buf = busw ? nand_write_buf16 : nand_write_buf;
```

我们的总线数据位宽是 8 位（8 条数据线），所以看“nand_write_buf”函数。

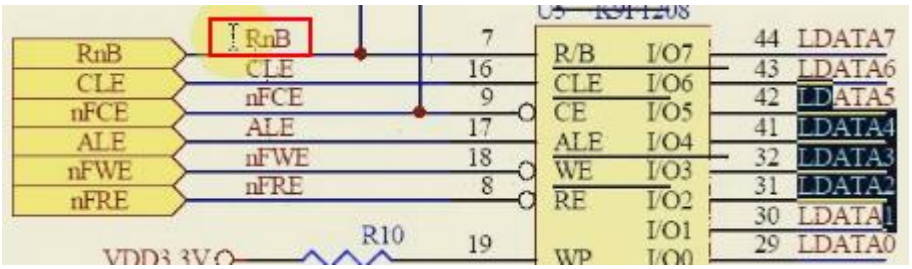
```
void nand_write_buf(struct mtd_info *mtd, const uint8_t *buf, int len)
{
int i;
struct nand_chip *chip = mtd->priv;

for (i = 0; i < len; i++)
writeb(buf[i], chip->IO_ADDR_W);
}
```

是写到“IO_ADDR_W”中，也需要自己提供。

```
s3c_nand->IO_ADDR_W   = &s3c_nand_regs->nfdata; //写数据
```

5，判断状态：



从原理图上看，判断状态是读“RnB”引脚。

```
chip->waitfunc = nand_wait;
```

```
-->chip->dev_ready(mtd)
```

这个“dev_ready()”函数也得自己提供。2440 中有一个 NFSTAT 寄存器：

NFCON STATUS REGISTER

Register	Address	R/W	Description	Reset Value
NFSTAT	0x4E000020	R/W	NAND flash operation status register	0xXX00

RnB (Read-only)	[0]	The status of RnB input pin. 0: NAND Flash memory busy 1: NAND Flash memory ready to operate	1

这一位域为“只读”，要判断状态，只需要返回这个“RnB”位（即 NFSTAT 的 bit[0]）的值即可。

```
s3c_nand->dev_ready   = s3c2440_dev_ready;      //判断状态
//2.3,判断状态
static int s3c2440_dev_ready(struct mtd_info *mtd)
{
return (s3c_nand_regs->nfstat & (1<<0));
}
```

nand_chip 结构中读写数据和判断状态都要用到 2440NAND 控制器的寄存器

Register Name	Address (B. Endian)	Address (L. Endian)	Acc. Unit	Read/ Write	Function
NAND Flash					
NFCONF	0x4E000000	←	W	R/W	NAND flash configuration
NFCONT	0x4E000004				NAND flash control
NFCMD	0x4E000008				NAND flash command
NFADDR	0x4E00000C				NAND flash address
NFDATA	0x4E000010				NAND flash data
NFMECC0	0x4E000014				NAND flash main area ECC0/1
NFMECC1	0x4E000018				NAND flash main area ECC2/3
NFSECC	0x4E00001C				NAND flash spare area ECC
NFSTAT	0x4E000020				NAND flash operation status
NFESTAT0	0x4E000024				NAND flash ECC status for I/O[7:0]
NFESTAT1	0x4E000028				NAND flash ECC status for I/O[15:8]
NFMECC0	0x4E00002C			R	NAND flash main area ECC0 status
NFMECC1	0x4E000030				NAND flash main area ECC1 status
NFSECC	0x4E000034				NAND flash spare area ECC status
NFSBLK	0x4E000038			R/W	NAND flash start block address
NFEBLK	0x4E00003C				NAND flash end block address

一会在结构体中要相关 4 字节，所以要查看看上面寄存器间的间隔。上面都是间隔 4 字节 。

二，硬件相关的设置：

要看时序图：硬件设置主要就是下面这些信号间的时间参数的设置。

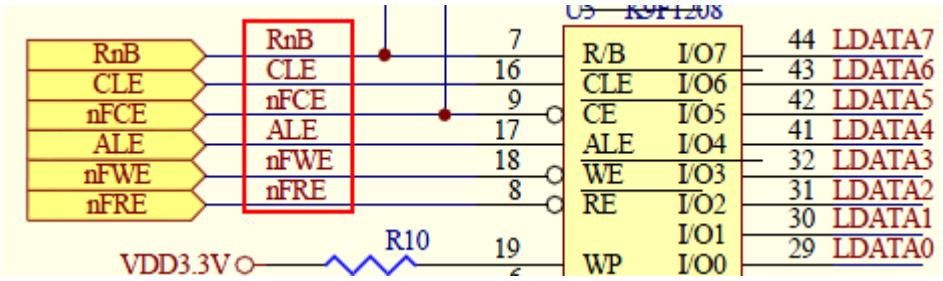
A,NFCONF 寄存器设置：

HCLK=100MHz

＊ TACLS： 发出 CLE/ALE 之后多长时间才发出 nWE 信号，从 NAND 手册可知 CLE/ALE 与 nWE 可以同时发出,所以 TACLS=0

＊ TWRPH0：nWE 的脉冲宽度，HCLK x （ TWRPH0 + 1 ），从 NAND 手册可知它要>=12ns，所以 TWRPH0>=1

＊ TWRPH1：nWE 变为高电平后多长时间 CLE/ALE 才能变为低电平，从 NAND 手册可知它要>=5ns，所以 TWRPH1>=0



发上面这些信号之间需要维持一个稳定的时钟。不然 NAND FLASH 可以反应不过来。

从 2440 手册上看，就是要设置下面 NFCONF 寄存器中的三个位域。

NAND FLASH CONFIGURATION REGISTER

Register	Address	R/W	Description	Reset Value
NFCONF	0x4E000000	R/W	NAND flash configuration register	0x0000100X

TACLS	[13:12]	CLE & ALE duration setting value (0~3) Duration = HCLK x TACLS	01

TWRPH0	[10:8]	TWRPH0 duration setting value (0~7) Duration = HCLK x (TWRPH0 + 1)	000

TWRPH1	[6:4]	TWRPH1 duration setting value (0~7) Duration = HCLK x (TWRPH1 + 1)	000

内核的输入信息说明“HCLK”为 100MHz。则一个周期为 1/100M = 10ns .

NAND FLASH MEMORY TIMING

TACLS

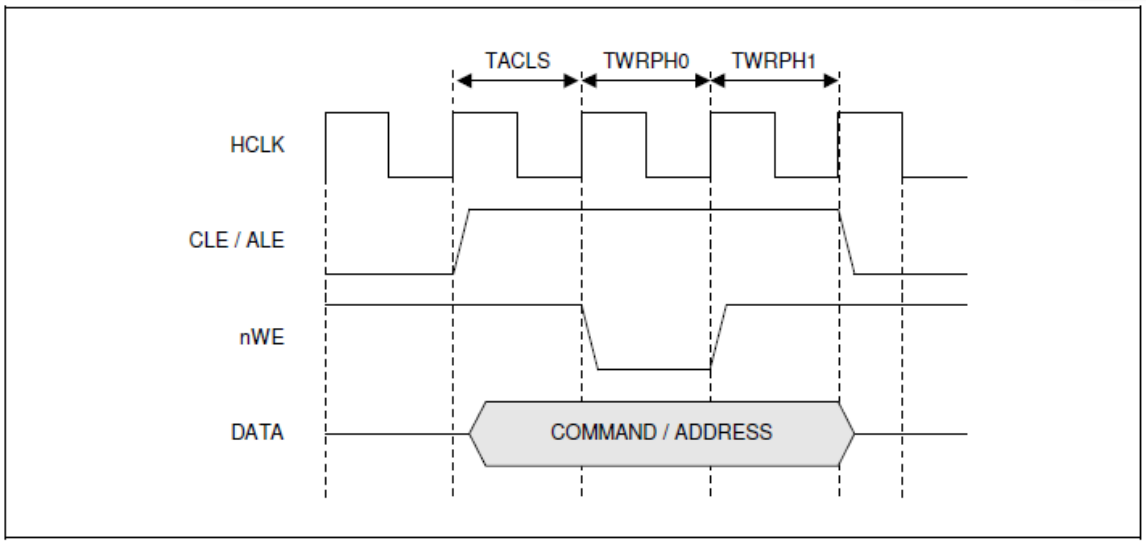


Figure 6-3. CLE & ALE Timing (TACLS=1, TWRPH0=0, TWRPH1=0)

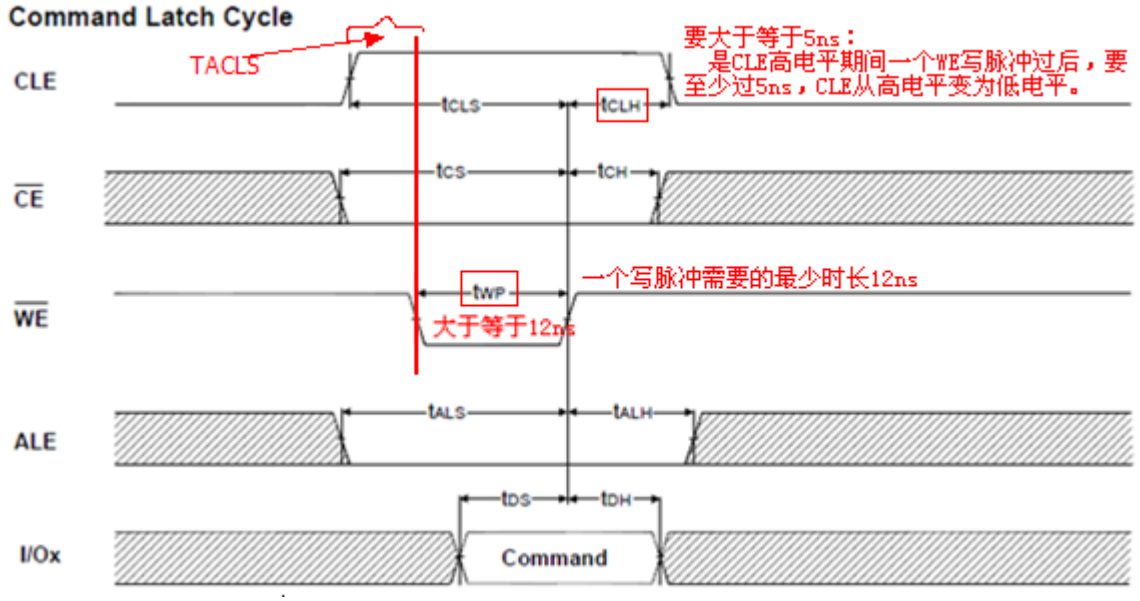
从上图知道：2440NAND 控制器

“TWRPH0”是“nWE”(写信号)的脉冲宽度。

发出“CLE/ALE”后，要过“TACLS”时间才能开始发一个“nWE”写信号。

“nWE”写信号变成高电平之后，还要过“TWRPH1”时间，“CLE/ALE”就变成低电平。

再看 NAND 芯片的时序图：找个简单的（意思一样）



NAND 芯片手册

1，″tWP″:“WE”写脉冲要维持的时间“twp”：

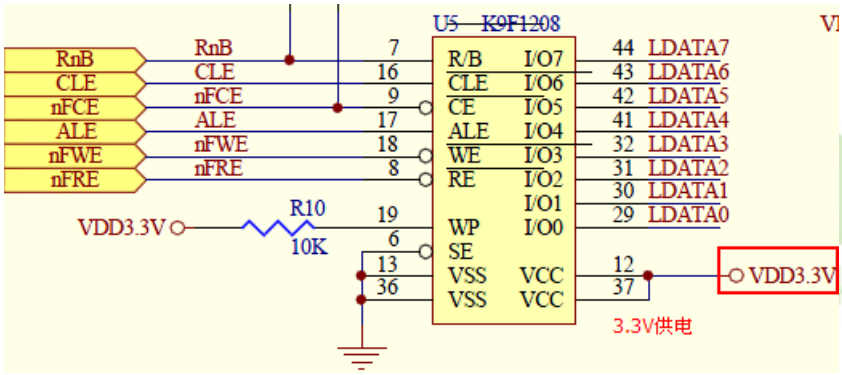
对应 2440 上的“TWRPH0”

AC Timing Characteristics for Command / Address / Data Input

Parameter	Symbol	Min		Max		Unit
		1.8V	3.3V	1.8V	3.3V	
CLE Setup Time	tCLS(1)	21	12	-	-	ns
CLE Hold Time	tCLH	5	5	-	-	ns
$\overline{CE}$ Setup Time	tCS(1)	25	20	-	-	ns
$\overline{CE}$ Hold Time	tCH	5	5	-	-	ns
$\overline{WE}$ Pulse Width	tWP	21	12	-	-	ns
ALE Setup Time	tALS(1)	21	12	-	-	ns
ALE Hold Time	tALH	5	5	-	-	ns
Data Setup Time	tDS(1)	20	12	-	-	ns
Data Hold Time	tDH	5	5	-	-	ns
Write Cycle Time	tWC	42	25	-	-	ns
$\overline{WE}$ High Hold Time	tWH	15	10	-	-	ns
Address to Data Loading Time	tADL(2)	100	100	-	-	ns

NOTES : 1. The transition of the corresponding control pins must occur only once while $\overline{WE}$ is held low
2. tADL is the time from the WE rising edge of final address cycle to the WE rising edge of first data cycle

$\overline{WE}$ Pulse Width	tWP	21	12 3.3V	-	-	ns



查看 NAND 芯片手册上规定的 WE 写脉冲时间长度。当 3.3V 时最小值是“12”纳秒。

TACLS	[13:12]	CLE & ALE duration setting value (0~3) Duration = HCLK x TACLS	01

根据公式得到：

('?'+1) * HCLK ≥ 12ns 即 ('?'+1) * 10ns ≥ 12ns => ? ≥ 1.

2,"tCLH":

当一个“WE”写脉冲结束后，要过“tCLH”时长，发命令 CLE 高电平期间会变成低电平。对应 2440 上“TWRPH1”.

CLE Hold Time	tCLH	5	5	-	-	ns

最小值要 5ns。

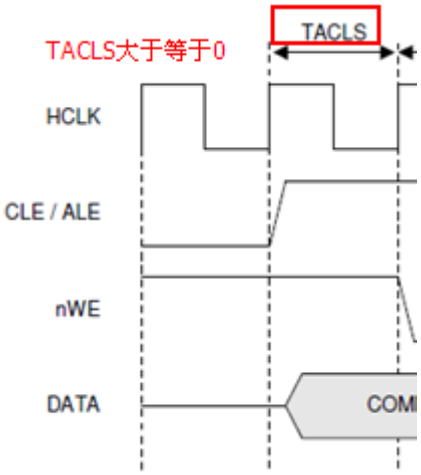
TWRPH1	[6:4]	TWRPH1 duration setting value (0~7) Duration = HCLK x (TWRPH1 + 1)	000

根据公式得到：

('?'+1) * HCLK ≥ 5ns 即 ('?'+1) * 10ns ≥ 5ns 则 ? ≥ 0.

3,“CLE/ALE”高电平多久后，开始“WE”写脉冲。

对应于 2440 上的"TACLS":通过 NAND 芯片手册规则看 TACLS≥0.



CLE Setup Time	tCLS(1)	21	12	-	-	ns
ALE Setup Time	tALS(1)	21	12	-	-	ns
WE Pulse Width	tWP	21	12	-	-	ns

从 NAND 芯片手册上规定的时间来看，ALE/CLE 的 setup 时间都是“大于等于 12ns”和 WE（脉冲时间为 12ns）可以同时发出。

即：

TACLS	[13:12]	CLE & ALE duration setting value (0~3) Duration = HCLK x TACLS	01

TACLS = HCLK x (≥0)

TACLS 取值大于等于 0.

4,"TWRPH1":

一个"WE"写脉冲过后，要过 TWRPH1 时钟，CLE/ALE 从高电平期变成低电平。

TWRPH0	[10:8]	TWRPH0 duration setting value (0~7) Duration = HCLK x (TWRPH0 + 1)	000

从 2440 手册上看有“加 1”。即 ('?'+1) * HCLK ≥ 12ns

/* NFCONF 寄存器相关设置: HCLK=100MHz (从单板内核启动信息看到)

* TACLS([13:12]): 发出 CLE/ALE 之后多长时间才发出 nWE 信号, 从 NAND 手册可知 CLE/ALE 与 nWE 可以同时发出,所以 TACLS=0

* 2440 上公式:TACLS->Duration = HCLK x TACLS

* TWRPH0([10:8]): nWE 的脉冲宽度, HCLK x (TWRPH0 + 1), 从 NAND 手册可知它要>=12ns, 所以 TWRPH0>=1

* 2440 上公式:TWRPH0->Duration = HCLK x (TWRPH0 + 1)

```
 * TWRPH1([6:4]): nWE 变为高电平后多长时间 CLE/ALE 才能变为低电平，从 NAND 手册可知它要>=5ns，所以 TWRPH1>=0
 *              2440 上公式:TWRPH1->Duration = HCLK x ( TWRPH1 + 1 )
 */
#define TACLS    0  //因为TACLS=0
#define TWRPH0   1  //因为TWRPH0 >= 1
#define TWRPH1   0  //因为TWRPH1 >= 0
s3c_nand_regs->nfconf = (TACLS<<12) | (TWRPH0<<8) | (TWRPH1<<4);
```

B,NFCONT 寄存器设置：

Reserved [14:15] ： Reserved
Lock-tight [13]：先不用管。
Soft Lock [12] ： 软件锁，都不用管。
Reserved [11] ： Reserved
EnbRnBINT [9] ： 中断不需要。
RnB_TransMode [8] ： 上升沿触发，或下降沿触发，中断中需要。这个也不用管。
SpareECCLock [6] ： ECC 校验先不管。
Reg_nCE [1] ： 使能信号。NAND Flash Memory nFCE signal control
0: Force nFCE to low (Enable chip select)
1: Force nFCE to high (Disable chip select)
MODE [0] ： NAND flash controller operating mode ＮＡＮＤ控制器要使能才能用。
0: NAND flash controller disable (Don’t work)
1: NAND flash controller enable
代码：

```
/* NFCONT:
 * BIT1-设为 1，取消片选
 * BIT0-设为 1，使能 NAND FLASH 控制器
 */
s3c_nand_regs->nfcont = (1<<1) | (1<<0);
```

以上便完成了ＮＡＮＤ的硬件操作，因为２４４０有ＮＡＮＤ控制器，所以只是设置了“ＮＦＣＯＮＦ”和“ＮＦＣＯＮＴ”两个寄存器中的相关位域。

Ｃ，为了省电。内核启动时会将一些用不到的模块都关掉。要使用某模块就得使能ＣＬＫＣＯＮ相关位域。

CLOCK CONTROL REGISTER (CLKCON)

Register	Address	R/W	Description	Reset Value
CLKCON	0x4C00000C	R/W	Clock generator control register	0xFFFFF0

CLKCON	Bit	Description	Initial State
AC97	[20]	Control PCLK into AC97 block. 0 = Disable, 1 = Enable	1
Camera	[19]	Control HCLK into Camera block. 0 = Disable, 1 = Enable	1
SPI	[18]	Control PCLK into SPI block. 0 = Disable, 1 = Enable	1
IIS	[17]	Control PCLK into IIS block. 0 = Disable, 1 = Enable	1
IIC	[16]	Control PCLK into IIC block. 0 = Disable, 1 = Enable	1
ADC(&Touch Screen)	[15]	Control PCLK into ADC block. 0 = Disable, 1 = Enable	1
RTC	[14]	Control PCLK into RTC control block. Even if this bit is cleared to 0, RTC timer is alive. 0 = Disable, 1 = Enable	1
GPIO	[13]	Control PCLK into GPIO block. 0 = Disable, 1 = Enable	1
UART2	[12]	Control PCLK into UART2 block. 0 = Disable, 1 = Enable	1
UART1	[11]	Control PCLK into UART1 block. 0 = Disable, 1 = Enable	1
UART0	[10]	Control PCLK into UART0 block. 0 = Disable, 1 = Enable	1
SDI	[9]	Control PCLK into SDI interface block. 0 = Disable, 1 = Enable	1
PWMTIMER	[8]	Control PCLK into PWMTIMER block. 0 = Disable, 1 = Enable	1
USB device	[7]	Control PCLK into USB device block. 0 = Disable, 1 = Enable	1
USB host	[6]	Control HCLK into USB host block. 0 = Disable, 1 = Enable	1
LCDC	[5]	Control HCLK into LCDC block. 0 = Disable, 1 = Enable	1
NAND Flash Controller	[4]	Control HCLK into NAND Flash Controller block. 0 = Disable, 1 = Enable	1
SLEEP	[3]	Control SLEEP mode of S3C2440A. 0 = Disable, 1 = Transition to SLEEP mode	0
IDLE BIT	[2]	Enter IDLE mode. This bit is not cleared automatically. 0 = Disable, 1 = Transition to IDLE mode	0
Reserved	[1:0]	Reserved	0

这里要发　使能　ＮＡＮＤ模块，就要将“ＣＬＫＣＯＮ”的bit[4]设置为“1”，可以先 ioremap 此寄存器后再设置些位，也可以用“clk_get()”函数。
可以理解这是一个总开关。使能了此 NAND 模块后，才能作设置。
/* 使能 NAND FLASH 控制器的时钟 */

```
clk = clk_get(NULL, "nand");
clk_enable(clk);          /* CLKCON'bit[4] 设置为1*/
```

三，实验：

```
book@book-desktop:/work/drivers_and_test/14th_nand/2th$ make
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M]  /work/drivers_and_test/14th_nand/2th/s3c_nand.o
  Building modules, stage 2.
  MODPOST 1 modules
  LD [M]  /work/drivers_and_test/14th_nand/2th/s3c_nand.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
book@book-desktop:/work/drivers_and_test/14th_nand/2th$ cp s3c_nand.ko /work/nfs_root/first_fs
book@book-desktop:/work/drivers_and_test/14th_nand/2th$
```

现在只是做了识别，分区还没有添加。所以此时做实验可以不卸载原来的 NAND 驱动。

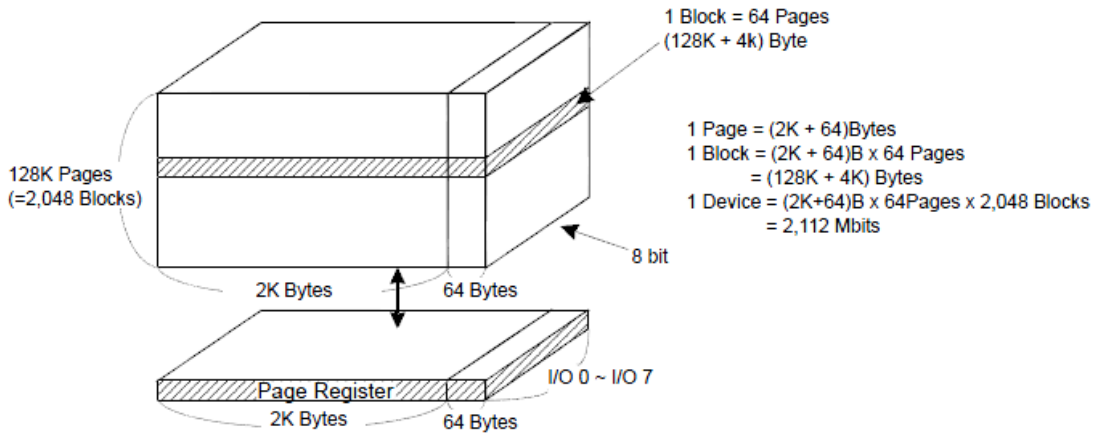
```
# insmod s3c_nand.ko
NAND device: Manufacturer ID: 0xec, Chip ID: 0xda (Samsung NAND 256MiB 3,3V 8-bit)
NAND_ECC_NONE selected by board driver. This is not recommended !!
Scanning device for bad blocks
Bad eraseblock 256 at 0x02000000
Bad eraseblock 257 at 0x02020000
Bad eraseblock 319 at 0x027e0000
Bad eraseblock 606 at 0x04bc0000
Bad eraseblock 608 at 0x04c00000
#
```

上面的警告是说不使用 ECC 是不推荐的。

最后只需要“add_mtd_partitions（）”就能通知“字符设备”或“块设备”。“块设备”里面会最终分配一个 gendisk 结构，add_disk()注册这个 gendisk 等。

四，关于 ECC:

此 NAND 的结构：



	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	*L	*L	*L	*L	*L	*L	*L	Row Address

是一页一页（一个扇区一个扇区）的结构，一页是 2KB，除了这 2KB 外还有 64B 的空间，这个 64B 区叫“OOB”(out of bank 叫作在 BANK 之外的东西）。这个 OOB 区不参与“统一编址”，就是说假若某一页 A 的地址是“2-2047”，那么 2048 这个地址不是在 OOB 区，而是在页 B 上（2048-4095）。

引入 OOB，是因为 NAND 有个缺点：位反转。如读一页数据时，里面很可能有某一位发生了位反转。本来值为 0，读出来为“1”。写的时候也有可能发生“位反转”。
这样引入了“ECC”校验。

解决位反转：烧写时

1，写一页数据。
2，用这一页数据生成 ECC 码（校验码）。
3，把 ECC 写入 OOB 里。

读的时候：

1，读整页的数据。
2，读 OOB 里的 ECC 码。

3，通过读出来的一页数据算校验码。
4，比较从 OOB 里读出来的 ECC 码和通过读到的一页数据算出来的 ECC 码是否相同。
不同则是发生了位反转。ECC 码是特定设置的，可以通过它知道是哪一位发生了反转。

ECC 校验码，可以用硬件生成，也可以用软件生成。

```
s3c_nand->ecc.mode     = NAND_ECC_SOFT;           //用软件生成ECC校验
```

```
# rmmod s3c_nand
# insmod s3c_nand.ko
NAND device: Manufacturer ID: 0xec, Chip ID: 0xda (Samsung NAND 256MiB 3.3V 8-bit)
Scanning device for bad blocks
Bad eraseblock 256 at 0x02000000
Bad eraseblock 257 at 0x02020000
Bad eraseblock 319 at 0x027e0000
Bad eraseblock 606 at 0x04bc0000
Bad eraseblock 608 at 0x04c00000
#
```

这时便没有警告了。上面有扫描坏块，NAND 上扫描坏块是正常的。

五，添加分区：

若只是把 NAND 分成一个分区，则只要用“add_mtd_device(mtd_info 结构)”就可以。
若要构造分区，则用“add_mtd_partitions（）”。

```
int add_mtd_partitions(struct mtd_info *master,const struct mtd_partition *parts,int
nbparts) {}
```

参 1，mtd_info 结构体。
参 2，mtd_partition 结构指针。相当于一个结构数组。最终是数组，组元是结构体。
参 3，就是参 2 这个 mtd_partition 结构数组的组元有多少项。
可以看到内核中原来分区信息：

```
Creating 4 MTD partitions on "NAND 256MiB 3.3V 8-bit":
0x00000000-0x00040000 : "bootloader"
0x00040000-0x00060000 : "params"
0x00060000-0x00260000 : "kernel"
0x00260000-0x10000000 : "root"
```

可以在内核中搜索上面的分区信息：

```
linux-2.6.22.6\arch\arm\plat-s3c24xx\Common-smdk.c
```

```
/* NAND parititon from 2.4.18-swl5 */
static struct mtd_partition smdk_default_nand_part[] = {
     [0] = {
          .name    = "bootloader",
          .size    = 0x00040000,
          .offset = 0,
     },
     [1] = {
          .name    = "params",
          .offset = MTDPART_OFS_APPEND,
          .size    = 0x00020000,
     },
     [2] = {
          .name    = "kernel",
          .offset = MTDPART_OFS_APPEND,
          .size    = 0x00200000,
     },
     [3] = {
          .name    = "root",
          .offset = MTDPART_OFS_APPEND,
          .size    = MTDPART_SIZ_FULL,
     }
};
```

直接将此结构数组改个自己的名字即可。

//要分区时，要构造这个 mtd_partition 结构数组。里面有 4 项。

```
static struct mtd_partition s3c_nand_part[] = {
[0] = {
        .name   = "bootloader", //分区名字
        .size   = 0x00040000,   //分区大小
.offset          = 0,            //分区偏移值
},
[1] = {
        .name   = "params",
        .offset = MTDPART_OFS_APPEND,   //APPEND紧跟上一个分区
        .size   = 0x00020000,
},
[2] = {
        .name   = "kernel",
        .offset = MTDPART_OFS_APPEND,
        .size   = 0x00200000,
},
[3] = {
        .name   = "root",
        .offset = MTDPART_OFS_APPEND,
        .size   = MTDPART_SIZ_FULL,     //剩下的所有空间大小
}
};
```

del_mtd_partitions; //清除分区结构数组.

测试 4th:

```
book@book-desktop:/work/drivers_and_test/14th_nand/4th$ make && cp s3c_nand.ko /work/nfs_root/first_fs
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M]  /work/drivers_and_test/14th_nand/4th/s3c_nand.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /work/drivers_and_test/14th_nand/4th/s3c_nand.mod.o
  LD [M]  /work/drivers_and_test/14th_nand/4th/s3c_nand.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
book@book-desktop:/work/drivers_and_test/14th_nand/4th$ cd /work/system/linux-2.6.22.6/
```

1. make menuconfig 去掉内核自带的 NAND FLASH 驱动

```
-> Device Drivers
  -> Memory Technology Device (MTD) support  内存技术设备
    -> NAND Device Support
   < >   NAND Flash support for S3C2410/S3C2440 SoC 先去掉原来的驱动
```

2. make uImage

使用新内核启动，并且使用 NFS 作为根文件系统 。因为已把 NAND 驱动去掉了。之前在 NAND 上的根文件系统就用不了。

```
book@book-desktop:/work/system/linux-2.6.22.6$ cp arch/arm/boot/uImage /work/nfs_root/uImage_nonand
book@book-desktop:/work/system/linux-2.6.22.6$
```

```
Environment size: 443/131068 bytes
OpenJTAG> set bootargs console=ttySAC0 root=/dev/nfs nfsroot=192.168.1.5:/work/nfs_root/first_fs ip=192.168.1.17:192.168.1.5:192.168
.1.1:255.255.255.0::eth0:off
OpenJTAG>
```

off指不自动配置

设置 UBOOT 启动参数。

```
OpenJTAG> nfs 30000000 192.168.1.5:/work/nfs_root/uImage_nonand:
```

从 nfs 上下载内核。

```
OpenJTAG> bootm 30000000
```

启动内核

接下来的启动信息中也看不到 NAND 信息了，因为 make menuconfigJ 时去掉了 NAND 的驱动。

3. insmod s3c_nand.ko

```
# ls /dev/mtd*
ls: /dev/mtd*: No such file or directory
# insmod s3c_nand.ko
NAND device: Manufacturer ID: 0xec, Chip ID: 0xda (Samsung NAND 256MiB 3.3V 8-bit)
Scanning device for bad blocks
Bad eraseblock 256 at 0x02000000
Bad eraseblock 257 at 0x02020000
Bad eraseblock 319 at 0x027e0000
Bad eraseblock 606 at 0x04bc0000
Bad eraseblock 608 at 0x04c00000
Creating 4 MTD partitions on "NAND 256MiB 3.3V 8-bit":
0x00000000-0x00040000 : "bootloader"
0x00040000-0x00060000 : "params"
0x00060000-0x00260000 : "kernel"
0x00260000-0x10000000 : "root"
# ls /dev/mtd*
/dev/mtd0       /dev/mtd1ro     /dev/mtd3       /dev/mtdblock1
/dev/mtd0ro     /dev/mtd2       /dev/mtd3ro     /dev/mtdblock2
/dev/mtd1       /dev/mtd2ro     /dev/mtdblock0  /dev/mtdblock3
# ls /dev/mtd* -l
crw-rw----    1 0        0          90,   0 Jan  1 00:00 /dev/mtd0
crw-rw----    1 0        0          90,   1 Jan  1 00:00 /dev/mtd0ro
crw-rw----    1 0        0          90,   2 Jan  1 00:00 /dev/mtd1
crw-rw----    1 0        0          90,   3 Jan  1 00:00 /dev/mtd1ro
crw-rw----    1 0        0          90,   4 Jan  1 00:00 /dev/mtd2
crw-rw----    1 0        0          90,   5 Jan  1 00:00 /dev/mtd2ro
crw-rw----    1 0        0          90,   6 Jan  1 00:00 /dev/mtd3
crw-rw----    1 0        0          90,   7 Jan  1 00:00 /dev/mtd3ro
brw-rw----    1 0        0          31,   0 Jan  1 00:00 /dev/mtdblock0
brw-rw----    1 0        0          31,   1 Jan  1 00:00 /dev/mtdblock1
brw-rw----    1 0        0          31,   2 Jan  1 00:00 /dev/mtdblock2
brw-rw----    1 0        0          31,   3 Jan  1 00:00 /dev/mtdblock3
#
```

4 个分区，创建了 4 对（只读的和读写的）字符设备节点。和 4 个块设备。

```
brw-rw----    1 0        0          31,   0 Jan  1 00:00 /dev/mtdblock0
brw-rw----    1 0        0          31,   1 Jan  1 00:00 /dev/mtdblock1
brw-rw----    1 0        0          31,   2 Jan  1 00:00 /dev/mtdblock2
brw-rw----    1 0        0          31,   3 Jan  1 00:00 /dev/mtdblock3
# mount /dev/mtdblock3 /mnt
UDF-fs: No VRS found
yaffs: dev is 32505859 name is "mtdblock3"
yaffs: passed flags ""
yaffs: Attempting MTD mount on 31.3, "mtdblock3"
yaffs: auto selecting yaffs2
block 238 is bad
block 239 is bad
block 301 is bad
block 588 is bad
block 590 is bad

yaffs: dev is 32505859 name is "mtdblock3"
yaffs: passed flags ""
yaffs: Attempting MTD mount on 31.3, "mtdblock3"
block 238 is bad
block 239 is bad
block 301 is bad
block 588 is bad
block 590 is bad
#
```

挂接 mtdblock3 .

```
# cd /mnt/
# ls
bin         lib         mnt         root        tmp
dev         linuxrc     opt         sbin        usr
etc         lost+found  proc        sys
#
```

可以看到之前 NAND 上的内容还在。

若之前 NAND 没有内容。

4. 格式化

(参考下面编译工具：mtd-utils-05.07.23.tar.bz2) 只要擦除即格式化了。

```
Creating 4 MTD partitions on "NAND 256MiB 3.3V 8-bit":
0x00000000-0x00040000 : "bootloader"
0x00040000-0x00060000 : "params"
0x00060000-0x00260000 : "kernel"
0x00260000-0x10000000 : "root"
```

从上面看“root” 为 /dev/mtd3,这里先格式化它。内核就先不要格了。

flash_eraseall /dev/mtd3 // yaffs 擦除后本身就是 yaffs 文件系统，

```
# flash_eraseall  /dev/mtd3
Erasing 128 Kibyte @ 1d80000 -- 11 % complete.
Skipping bad block at 0x01da0000

Skipping bad block at 0x01dc0000
Erasing 128 Kibyte @ 2560000 -- 14 % complete.
Skipping bad block at 0x02580000
Erasing 128 Kibyte @ 4940000 -- 28 % complete.
Skipping bad block at 0x04960000
Erasing 128 Kibyte @ 4980000 -- 28 % complete.
Skipping bad block at 0x049a0000
Erasing 128 Kibyte @ fd80000 -- 99 % complete.
```

5. 挂接

mount -t yaffs /dev/mtdblock3 /mnt

```
# mount -t yaffs /dev/mtdblock3 /mnt
yaffs: dev is 32505859 name is "mtdblock3"
yaffs: passed flags ""
yaffs: Attempting MTD mount on 31.3, "mtdblock3"
yaffs: auto selecting yaffs2
block 238 is bad
block 239 is bad
block 301 is bad
block 588 is bad
block 590 is bad
# cd /mnt/
# ls
lost+found
#
```

6. 在/mnt 目录下建文件

编译工具：在 PC 上交叉编译

1. tar xjf mtd-utils-05.07.23.tar.bz2
2. cd mtd-utils-05.07.23/util

修改 Makefile:

#CROSS=arm-linux-

改为

CROSS=arm-linux-

3. make
4. cp flash_erase flash_eraseall /work/nfs_root/first_fs/bin/

flash_erase 是擦除一个扇区。

flash_eraseall 擦除整个分区