[基于MTD的NANDFLASH设备驱动底层实现原理分析(一)](http://blog.csdn.net/eilianlau/article/details/6961406)

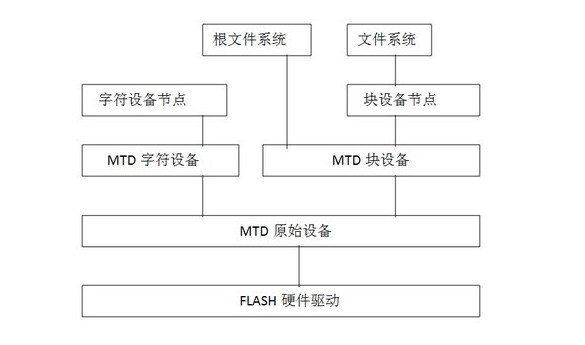
分类： [Linux驱动开发](http://blog.csdn.net/eilianlau/article/category/922573)2011-11-12 00:08 4039人阅读 [评论](http://blog.csdn.net/eilianlau/article/details/6961406#comments)(3) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/eilianlau/article/details/6961406#report)

[flash](http://www.csdn.net/tag/flash)[linux内核](http://www.csdn.net/tag/linux%e5%86%85%e6%a0%b8)[硬件驱动](http://www.csdn.net/tag/%e7%a1%ac%e4%bb%b6%e9%a9%b1%e5%8a%a8)[linux](http://www.csdn.net/tag/linux)[存储](http://www.csdn.net/tag/%e5%ad%98%e5%82%a8)[struct](http://www.csdn.net/tag/struct)

经过UBOOT初步的移植,Linux内核初步的移植,Linux内核总线设备模型的分析,等一系列痛苦的折腾,目的就是想更好的来分析下NANDFLASH的驱动。。大概一共历经了半个月的时间,慢慢的对NANDFLASH驱动程序有感觉了。。。

一、MTD体系结构：Linux内核提供MTD子系统来建立FLASH针对Linux的统一、抽象接口。MTD将文件系统与底层的FLASH存储器进行隔离。

      引入MTD后Linux系统中对FLASH的设备驱动分为4层



**设备节点：**用户在/dev目录下使用mknod命令建立MTD字符设备节点(主设备号为90)，或者MTD块设备节点(主设备号为31)，使用该设备节点即可访问MTD设备。  
  
**MTD设备层：**基于MTD原始设备层，系统将MTD设备可以定义为MTD字符(在/mtd/mtdchar.c中实现,设备号90)和MTD块设备(在/mtd/mtdblock.c中实现,设备号31)。  
  
**MTD原始设备层：**MTD原始设备层由两部分构成，一部分是MTD原始设备的通用代码，另一部分是各个特定Flash的数据，如分区。

                主要构成的文件有：

                     drivers/mtd/mtdcore.c支持mtd字符设备

                     driver/mtd/mtdpart.c支持mtd块设备

**Flash硬件驱动层：**Flash硬件驱动层负责对Flash硬件的读、写和擦除操作。MTD设备的Nor Flash芯片驱动位于drivers/mtd/chips/子目录下，Nand Flash芯片的驱动则位于drivers/mtd/nand/子目录下。

二、Linux内核中基于MTD的NANDFLASH驱动代码布局：

在Linux2.6.35内核中，MTD源代码放在driver/mtd目录中，该目录中包含chips、devices、maps、nand、onenand、lpdrr、tests和ubi八个子目录。

其中只有nand和onenand目录中的代码才与NAND驱动相关，不过nand目录中的代码比较通用，而onenand目录中的代码相对于nand中的代码而言则简化了很多，它是针对三星公司开发的另一类Flash芯片，即OneNAND Flash。

本文我们需要关注的代码是linux-2.6.35/drivers/mtd/nand目录中,在该目录中我们关心的文件如下：

1、  nand\_base.c：

定义了NAND驱动中对NAND芯片最基本的操作函数和操作流程，如擦除、读写page、读写oob等。当然这些函数都只是进行一些default的操作，若你的系统在对NAND操作时有一些特殊的动作，则需要在你自己的驱动代码中进行定义，然后Replace这些default的函数。

2、  nand\_bbt.c：

定义了NAND驱动中与坏块管理有关的函数和结构体。

3、  nand\_ids.c：

定义了两个全局类型的结构体：struct nand\_flash\_dev nand\_flash\_ids[ ]和struct nand\_manufacturers nand\_manuf\_ids[ ]。其中前者定义了一些NAND芯片的类型，后者定义了NAND芯片的几个厂商。NAND芯片的ID至少包含两项内容：厂商ID和厂商为自己的NAND芯片定义的芯片ID。当NAND驱动被加载的时候，它会去读取具体NAND芯片的ID，然后根据读取的内容到上述定义的nand\_manuf\_ids[ ]和nand\_flash\_ids[ ]两个结构体中去查找，以此判断该NAND芯片是那个厂商的产品，以及该NAND芯片的类型。若查找不到，则NAND驱动就会加载失败，因此在开发NAND驱动前必须事先将你的NAND芯片添加到这两个结构体中去(其实这两个结构体中已经定义了市场上绝大多数的NAND芯片，所以除非你的NAND芯片实在比较特殊，否则一般不需要额外添加)。值得一提的是，nand\_flash\_ids[ ]中有三项属性比较重要，即pagesize、chipsize和erasesize，驱动就是依据这三项属性来决定对NAND芯片进行擦除，读写等操作时的大小的。其中pagesize即NAND芯片的页大小，一般为256、512或2048；chipsize即NAND芯片的容量；erasesize即每次擦除操作的大小，通常就是NAND芯片的block大小。

4、  nand\_ecc.c：

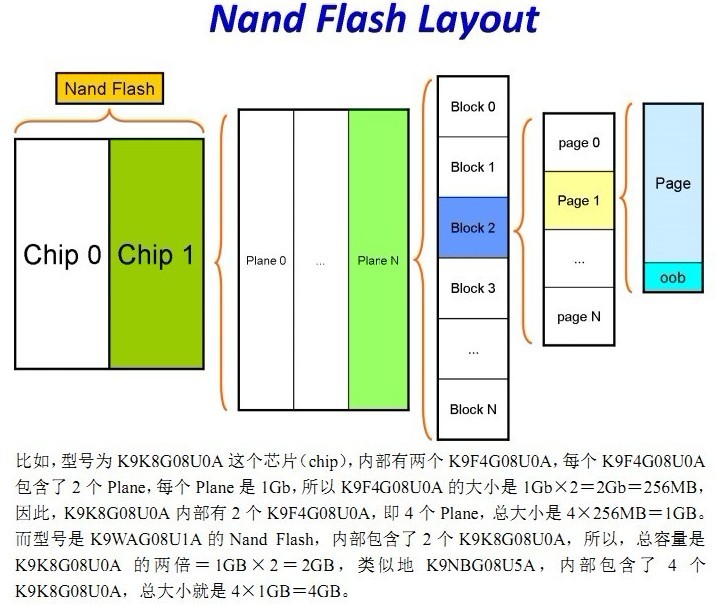
定义了NAND驱动中与softeware ECC有关的函数和结构体，若你的系统支持hardware ECC，且不需要software ECC，则该文件也不需理会。

上面这些内容我是Copy别人的我觉得写得太好了,因为一开始我真的很迷茫,在nand目录下有那么多的文件,到底哪个是值得我读的.我真的不值得,读了这个大神的博客后对NANDDLASH的驱动我不再是那么的迷茫。

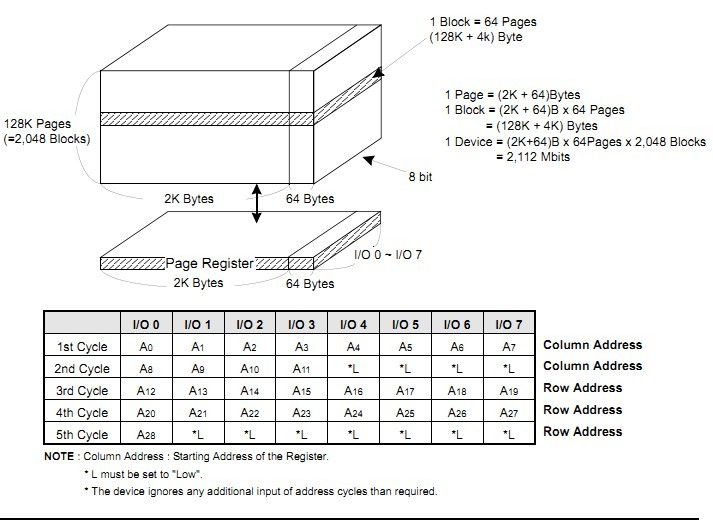
三、NANDFLASH的硬件特性

要想读懂后面Linux系统中对NANDFLASH硬件驱动代码,了解NANDFLASH的硬件特性这是再好不过的。

1、NANDFLASH的内部布局



2、Nand Flash的物理存储单元的阵列组织结构(以开发板上的K9F2G08为例)



        K9F2G08的大小是256M

    a)block:"Block是Nand Flash的擦除操作的基本/最小单位",一片NANDFLASH(chip)由很多块

(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称之为small block的Nand Flash。

    b)page:"page是读写操作的最小单位",每一个block里面包又含了许多page(页),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

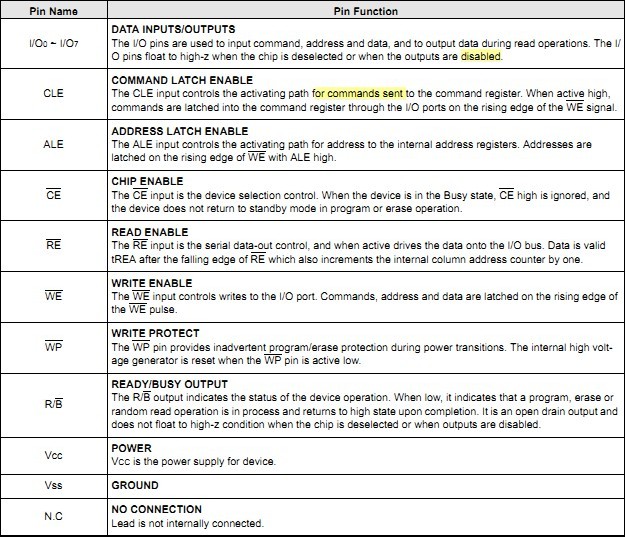
2KB的NandFlash，被称作 big block的 Nand Flash，对应的发读写命令地址，一共 5个周期(cycle)，

而老的 Nand Flash，页大小是 256B，512B，，这类的 Nand Flash被称作 small block的nandflash

地址周期只有4个。

    c)oob: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：   
   1、 标记是否是坏快   
   2、存储ECC数据   
   3、存储一些和文件系统相关的数据。如 jffs2 就会用到这些空间存储一些特定信息，  
   4、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。

3、K9F2G08的引脚定义



      IO7~IO0:用于输入地址/数据/命令，输出数据。

    CLE：命令锁存使能位,在发送命令之前要先将模式寄存器中设置CLE使能(高电平有效)。

    ALE：地址锁存使能位,在发送地址之前,要先将模式寄存器中设置ALE使能(高电平有效)。

    CE：(nFCE)芯片的片选信号,操作nandflash前应该拉低该位使之选中该芯片。

    RE：(nFRE)读使能,低电平有效,读之前使CE有效。

    WE：(nFWE)写使能,低电平有效,写之前必须使WE有效。

    WP：写保护低电平有效

    R/B：(R/nB)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。(其中就绪：高电平,忙：低电平)。

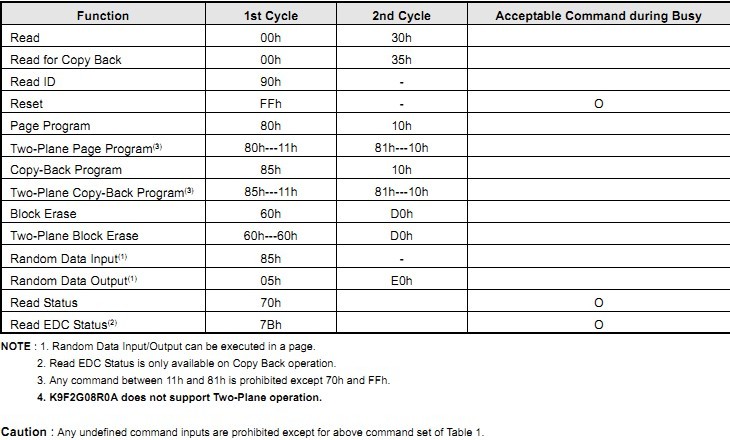
[基于MTD的NANDFLASH设备驱动底层实现原理分析(二)](http://blog.csdn.net/eilianlau/article/details/6962290)

分类： [Linux驱动开发](http://blog.csdn.net/eilianlau/article/category/922573)2011-11-12 11:14 1936人阅读 [评论](http://blog.csdn.net/eilianlau/article/details/6962290#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/eilianlau/article/details/6962290#report)

[flash](http://www.csdn.net/tag/flash)[linux](http://www.csdn.net/tag/linux)

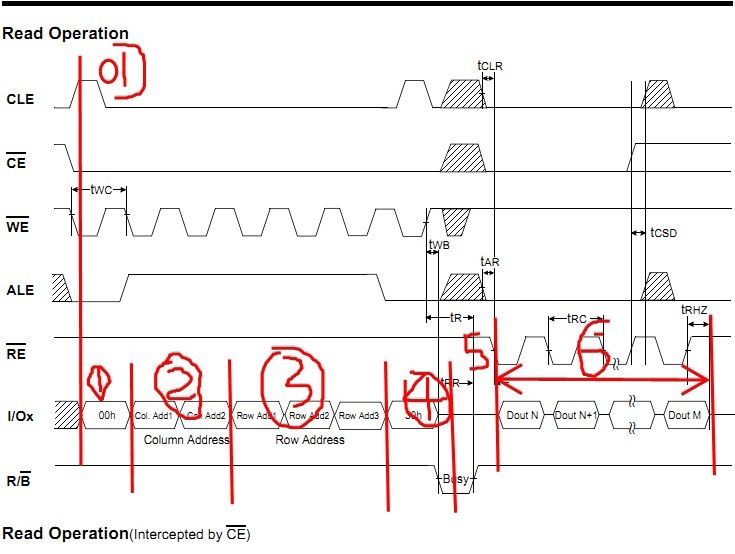
四、常见的NANDFLASH的操作

      1、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：



比如说要实现读一页的数据,就要发送Read命令,而且分两个周期发送,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，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,然后发送要写入的地址，再发送0x10h。

    2、读(Read)nandflash操作过程分析



    1)红色竖线穿过的第一行，是 CLE。前面介绍命令所存使能(CLE)的那个引脚将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，内部硬件逻辑才会直到收到的是命令，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。

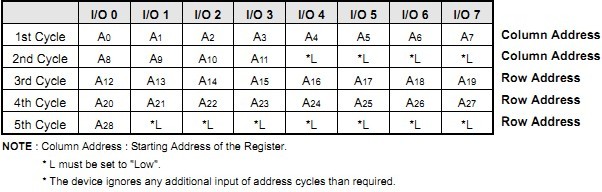
    7)第七行，R/B,高电平，表示 R（Ready）/就绪，因为到了后面的第 5阶段，硬件内部，在第四阶段，接受了外界的读取命令后，把该页的数据一点点送到页寄存器中，这段时间，属于系统在忙着干活，属于忙的阶段所以，R/B才变成低，表示 Busy忙的状态的。 其他的时序的就类似的理解。

3、计算我们要读取或者写入的行地址和例地址

     以mini2440开发板上的K9F2G08为例，此Nand Flash，一共有 2048 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 1000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



    从上图可以看出,该FLASH的地址周期一共有5个,2个列地址(Column)周期,3个行地址(Row)周期

    a)对应的列地址就是页内地址,该flash一个页的大小是2K即2048个字节,所以它的地址范围是0~2047,对应的上图的列地址A0-A10就是页内地址。你可能会发现多出了一个A11,从A0-A11，这样一共就有了12位,那它的地址范围就是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。

    b)对应地，A12～A28,称作页号，页的号码，可以定位到具体是哪一个页,该FLASH一共是64页一共需要6位即A12~A17,而其中A18～A28表示对应的块号,即属于哪个块。

这里有一个很重要的地方就是我们要传入的地址的每一位，就是对应着上表中的 A0 到 A28 ,实际上上表中的 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 1208，还没有超过 2047 呢，所以 A11肯定是 0

然后我们再来算上面我们要访问的地址：第 1000个块中的第 25 页中的 1208字节处的地址

第 1000个块中的第 25 页中的 1208字节处的地址它对应着的页内地址为：

页内地址 =1208Bytes

              =0x4B8

页      号 =块数\*每块多少页 + 块内的页号

              =1000\*64 + 25

              =0xFA19

也就是我们要访问0xFA19页内的0x4B8地址,再把这个地址转换成列和行地址

A0~A10是用来表示页内地址的所以把0x4B8拆分成两列

列地址1=0xB8,列地址2=0x04;

再把页号0xFA19拆分成3行

行地址1=0x19,行地址2=0xFA,行地址3=0x0;

    对应的看看linux2.6.35/driver/mtd/nand/nand\_base.c中地址的发送



上面的column即对应着页内地址,通常情况是0,如果不是0则通过传入进来的地址除于页地址就可得到相应的列地址了。而 page\_addr 即页号，就是通过要访问的地址，除于页大小，即可得到。

对于其他操作还正在研究中。。。。。。虽然说上面这些东东大部分都是来自别人的东西,但是我相信现在它已经变成我自己的东西了。。我不记得那个大帅的博客了,因为我是直接把它的博客给保存到本地了。。

[基于MTD的NANDFLASH设备驱动底层实现原理分析(三)](http://blog.csdn.net/eilianlau/article/details/6962693)

分类： [Linux驱动开发](http://blog.csdn.net/eilianlau/article/category/922573)2011-11-12 13:56 2260人阅读 [评论](http://blog.csdn.net/eilianlau/article/details/6962693#comments)(4) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/eilianlau/article/details/6962693#report)

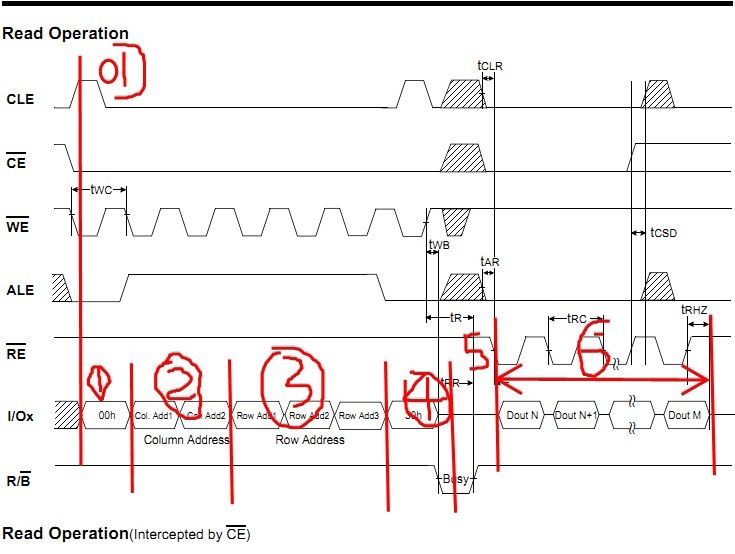
[command](http://www.csdn.net/tag/command)[cmd](http://www.csdn.net/tag/cmd)[struct](http://www.csdn.net/tag/struct)[delay](http://www.csdn.net/tag/delay)[flash](http://www.csdn.net/tag/flash)[硬件驱动](http://www.csdn.net/tag/%e7%a1%ac%e4%bb%b6%e9%a9%b1%e5%8a%a8)

非常尴尬的说：我突然发现在写这些关于NAND驱动的文章的时候，原来我一直是在改写别人的博客。。。。。其实这并不要紧的,我也觉得这不仅仅是一种比较好的学习方法了，为什么呢,因为当我在看他的博客的时候，我明白了一点,然后当我自己要写的时候。。对这个东东又进一步了解一点了。。呵呵Copy也分档次了

五、硬件时序到软件代码的演变过程对nand\_base.c部分代码的分析

该文件位于</linux2.6.35/dricer/mtd/nand/nand\_base.c>

还是把那个读NAND的硬件时序图给贴上,如下图：



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。   
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)  
{  
    /\*\*\*此处省略部分代码\*\*/

    。。。。。。。。。。。。。。

    while(1) {  
           /\*\*\*\*\*\*省略\*\*\*\*/

          .。。。。。。。。。。。。。。。

            if (likely(sndcmd)) {/\*#define NAND\_CMD\_READ0 0\*/

                /\*1)\*\*\*读取数据前肯定要先发送对应的读页命令\*\*\*\*\*\*/

                chip->cmdfunc(mtd, NAND\_CMD\_READ0, 0x00, page);  
                sndcmd = 0;  
            }  
  
            /\* Now read the page into the buffer \*/  
            if (unlikely(ops->mode == MTD\_OOB\_RAW))  
                ret = chip->ecc.read\_page\_raw(mtd, chip,  
                                  bufpoi, page);  
            else if (!aligned && NAND\_SUBPAGE\_READ(chip) && !oob)  
                ret = chip->ecc.read\_subpage(mtd, chip, col, bytes, bufpoi);  
            else

             /\*\*\*\*\*\*执行到这里read\_page函数读取对应的数据了\*\*\*\*\*\*/

                ret = chip->ecc.read\_page(mtd, chip, bufpoi,  
                              page);  
            if (ret < 0)  
                break;  
  
            /\* Transfer not aligned data \*/  
            if (!aligned) {  
                if (!NAND\_SUBPAGE\_READ(chip) && !oob)  
                    chip->pagebuf = realpage;  
                memcpy(buf, chip->buffers->databuf + col, bytes);  
            }  
  
            buf += bytes;  
          。。。。。。。。。。。。。。。。。。  
  
    if (mtd->ecc\_stats.failed - stats.failed)  
        return -EBADMSG;  
  
    return  mtd->ecc\_stats.corrected - stats.corrected ? -EUCLEAN : 0;  
}

上面这些代码都不需要我们去实现的,使用MTD层的自定义代码就行。。。

nand\_command\_lp的分析

static void nand\_command\_lp(struct mtd\_info \*mtd, unsigned int command,  
                int column, int page\_addr)  
{  
    register struct nand\_chip \*chip = mtd->priv;  
  
    /\* Emulate NAND\_CMD\_READOOB \*/  
    if (command == NAND\_CMD\_READOOB) {  
        column += mtd->writesize;  
        command = NAND\_CMD\_READ0;  
    }  
  
    /\* 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;

          /\* 发送两个column列地址，对应着上述步骤中的② \*/

            chip->cmd\_ctrl(mtd, column, ctrl);/\*发送列地址1\*/  
            ctrl &= ~NAND\_CTRL\_CHANGE;  
            chip->cmd\_ctrl(mtd, column >> 8, ctrl);/\*发送列地址2\*/  
        }  
        if (page\_addr != -1) {

          /\* 接下来是发送三个Row，行地址，对应着上述步骤中的② \*/

            chip->cmd\_ctrl(mtd, page\_addr, ctrl);/\*发送行地址1\*/  
            chip->cmd\_ctrl(mtd, page\_addr >> 8,/\*发送行地址2\*/  
                       NAND\_NCE | NAND\_ALE);  
            /\* One more address cycle for devices > 128MiB \*/  
            if (chip->chipsize > (128 << 20))  
                chip->cmd\_ctrl(mtd, page\_addr >> 16,/\*发送行地址3\*/  
                           NAND\_NCE | NAND\_ALE);  
        }  
    }  
    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) {  
。。。。。。。。。。。。。  
        return;  
   /\*\*\*复位\*\*/  
    case NAND\_CMD\_RESET:  
        if (chip->dev\_ready)  
            break;  
        udelay(chip->chip\_delay);  
        chip->cmd\_ctrl(mtd, NAND\_CMD\_STATUS,  
                   NAND\_NCE | NAND\_CLE | NAND\_CTRL\_CHANGE);  
        chip->cmd\_ctrl(mtd, NAND\_CMD\_NONE,  
                   NAND\_NCE | NAND\_CTRL\_CHANGE);  
        while (!(chip->read\_byte(mtd) & NAND\_STATUS\_READY)) ;  
        return;  
    /\*读忙信号\*/  
    case NAND\_CMD\_RNDOUT:  
        /\* No ready / busy check necessary \*/  
        chip->cmd\_ctrl(mtd, NAND\_CMD\_RNDOUTSTART,  
                   NAND\_NCE | NAND\_CLE | NAND\_CTRL\_CHANGE);  
        chip->cmd\_ctrl(mtd, NAND\_CMD\_NONE,  
                   NAND\_NCE | NAND\_CTRL\_CHANGE);  
        return;  
/\* 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 \*/   
    case NAND\_CMD\_READ0:  
        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:  
        /\*  
         \* If we don't have access to the busy pin, we apply the given  
         \* command delay  
         \*/  
        if (!chip->dev\_ready) {  
            udelay(chip->chip\_delay);  
            return;  
        }  
    }  
  
    /\* 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);  
}

还有一个步骤没有实现那就是步骤⑥了一点一点的把数据读出来

 nand\_read\_page\_hwecc分析

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);

         /\*\*这个最重要了这才是真正的从NAND的缓冲区中把数据给读出来\*\*\*\*/

        chip->read\_buf(mtd, p, eccsize);  
        chip->ecc.calculate(mtd, p, &ecc\_calc[i]);  
    }

  。。。。。。。。。。  
    return 0;  
}

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，那肯定是我们自己的事情了。。。接下来的工作是什么?MTD原始设备和硬件驱动层的交互了.这个才是我们要去真正实现的。。

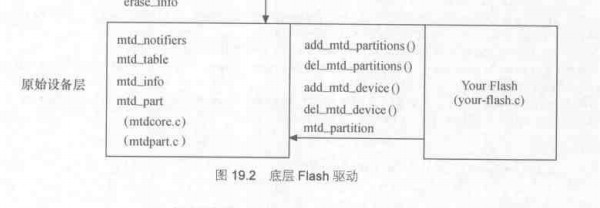
[基于MTD的NANDFLASH设备驱动底层实现原理分析(四)](http://blog.csdn.net/eilianlau/article/details/6962834)

分类： [Linux驱动开发](http://blog.csdn.net/eilianlau/article/category/922573)2011-11-12 15:06 2013人阅读 [评论](http://blog.csdn.net/eilianlau/article/details/6962834#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/eilianlau/article/details/6962834#report)

[struct](http://www.csdn.net/tag/struct)[linux内核](http://www.csdn.net/tag/linux%e5%86%85%e6%a0%b8)[structure](http://www.csdn.net/tag/structure)[硬件驱动](http://www.csdn.net/tag/%e7%a1%ac%e4%bb%b6%e9%a9%b1%e5%8a%a8)[locking](http://www.csdn.net/tag/locking)[user](http://www.csdn.net/tag/user)

进过前面3篇文章对NANDFLASH的一些硬件特性以及MTD的上层操作已经有了一个大体概念,这些东西的重要性就像你要吃饭那么你首先得学会拿筷子道理一样吧,应该一样的。

五、MTD原始设备层和硬件驱动层的桥梁：



熟悉这几个重要的结构体：linux/mtd/mtd.h

struct mtd\_info {  
    u\_char type;               /\*\*内存技术类型(包括MTD\_RAM,MTD\_ROM,MTD\_NANDFLASH等)\*\*/  
    uint32\_t flags;           /\*MTD设备属性标志\*/  
    uint64\_t size;     // Total size of the MTD  
  
    uint32\_t erasesize;//MTD设备的擦除单元大小，对于NandFlash来说就是Block的大小  
      
    uint32\_t writesize;//最小的可写单元的字节数  
  
    uint32\_t oobsize;   // Amount of OOB data per block (e.g. 16)  
    uint32\_t oobavail;  // Available OOB bytes per block  
  
    unsigned int erasesize\_shift;  
    unsigned int writesize\_shift;  
    /\* Masks based on erasesize\_shift and writesize\_shift \*/  
    unsigned int erasesize\_mask;  
    unsigned int writesize\_mask;  
  
    // Kernel-only stuff starts here.  
    const char \*name;  
    int index;  
  
    /\* ecc layout structure pointer - read only ! \*/  
    struct nand\_ecclayout \*ecclayout;  
  
    /\* Data for variable erase regions. If numeraseregions is zero,  
     \* it means that the whole device has erasesize as given above.

     \*一般为1吧

     \*/  
    int numeraseregions;

    //擦除区域的指针

    struct mtd\_erase\_region\_info \*eraseregions;  
    //擦除函数将一个erase\_info结构放入擦除队列中  
    int (\*erase) (struct mtd\_info \*mtd, struct erase\_info \*instr);  
  
    /\* This stuff for eXecute-In-Place \*/  
    /\* phys is optional and may be set to NULL \*/  
    int (\*point) (struct mtd\_info \*mtd, loff\_t from, size\_t len,  
            size\_t \*retlen, void \*\*virt, resource\_size\_t \*phys);  
  
    /\* We probably shouldn't allow XIP if the unpoint isn't a NULL \*/  
    void (\*unpoint) (struct mtd\_info \*mtd, loff\_t from, size\_t len);  
  
    /\* Allow NOMMU mmap() to directly map the device (if not NULL)  
     \* - return the address to which the offset maps  
     \* - return -ENOSYS to indicate refusal to do the mapping  
     \*/  
    unsigned long (\*get\_unmapped\_area) (struct mtd\_info \*mtd,  
                        unsigned long len,  
                        unsigned long offset,  
                        unsigned long flags);  
  
    /\* Backing device capabilities for this device  
     \* - provides mmap capabilities  
     \*/  
    struct backing\_dev\_info \*backing\_dev\_info;  
  
  //read和write分别用于MTD设备的读和写  
    int (\*read) (struct mtd\_info \*mtd, loff\_t from, size\_t len, size\_t \*retlen, u\_char \*buf);  
    int (\*write) (struct mtd\_info \*mtd, loff\_t to, size\_t len, size\_t \*retlen, const u\_char \*buf);  
  
    int (\*panic\_write) (struct mtd\_info \*mtd, loff\_t to, size\_t len, size\_t \*retlen, const u\_char \*buf);  
    //读写MTD设备的OOB区域的数据  
    int (\*read\_oob) (struct mtd\_info \*mtd, loff\_t from,  
             struct mtd\_oob\_ops \*ops);  
    int (\*write\_oob) (struct mtd\_info \*mtd, loff\_t to,  
             struct mtd\_oob\_ops \*ops);  
     //访问一些受保护的寄存器    
    int (\*get\_fact\_prot\_info) (struct mtd\_info \*mtd, struct otp\_info \*buf, size\_t len);  
    int (\*read\_fact\_prot\_reg) (struct mtd\_info \*mtd, loff\_t from, size\_t len, size\_t \*retlen, u\_char \*buf);  
    int (\*get\_user\_prot\_info) (struct mtd\_info \*mtd, struct otp\_info \*buf, size\_t len);  
    int (\*read\_user\_prot\_reg) (struct mtd\_info \*mtd, loff\_t from, size\_t len, size\_t \*retlen, u\_char \*buf);  
    int (\*write\_user\_prot\_reg) (struct mtd\_info \*mtd, loff\_t from, size\_t len, size\_t \*retlen, u\_char \*buf);  
    int (\*lock\_user\_prot\_reg) (struct mtd\_info \*mtd, loff\_t from, size\_t len);  
  
    /\* kvec-based read/write methods.  
       NB: The 'count' parameter is the number of \_vectors\_, each of  
       which contains an (ofs, len) tuple.  
    \*/  
    int (\*writev) (struct mtd\_info \*mtd, const struct kvec \*vecs, unsigned long count, loff\_t to, size\_t \*retlen);  
  
    /\* Sync \*//\*同步\*/  
    void (\*sync) (struct mtd\_info \*mtd);  
  
    /\* Chip-supported device locking \*/  
    int (\*lock) (struct mtd\_info \*mtd, loff\_t ofs, uint64\_t len);  
    int (\*unlock) (struct mtd\_info \*mtd, loff\_t ofs, uint64\_t len);  
  
    /\* Power Management functions \*/  
    int (\*suspend) (struct mtd\_info \*mtd);  
    void (\*resume) (struct mtd\_info \*mtd);  
  
    /\* Bad block management functions \*/  
    int (\*block\_isbad) (struct mtd\_info \*mtd, loff\_t ofs);  
    int (\*block\_markbad) (struct mtd\_info \*mtd, loff\_t ofs);  
  
    struct notifier\_block reboot\_notifier;  /\* default mode before reboot \*/  
  
    /\* ECC status information \*/  
    struct mtd\_ecc\_stats ecc\_stats;  
    /\* Subpage shift (NAND) \*/  
    int subpage\_sft;  
   //私有数据 指向map\_info结构  
    void \*priv;  
  
    struct module \*owner;  
    struct device dev;  
    int usecount;

   //设备驱动回调函数

    int (\*get\_device) (struct mtd\_info \*mtd);  
    void (\*put\_device) (struct mtd\_info \*mtd);  
};

上面的read()、write()、read\_oob()、等都是MTD设备驱动要实现的主要函数,不过这些函数都是透明的不需要我们自己去实现,因为Linux在MTD的下层实现了针对NORFLASH和NANDFLASH的通用mtd\_info成员函数。

感觉没什么可写的了,因为这些都不是我要关注的东西,但是又不能不知道有这么回事

这些结构体还是得了解了解

driver/mtd/mtdpart.c

/\* Our partition node structure \*/  
struct mtd\_part {  
    struct mtd\_info mtd;    //分区信息  
    struct mtd\_info \*master; //该分区的主分区  
    uint64\_t offset;              //该分区的偏移量  
    struct list\_head list;  
};

mtd\_partition会在MTD原始设备调用add\_mtd\_partitions()的时候传递分区参数/linux/mtd/partition.h

struct mtd\_partition {  
    char \*name;            /\* identifier string \*/  
    uint64\_t size;            /\* partition size \*/  
    uint64\_t offset;        /\* offset within the master MTD space \*/  
    uint32\_t mask\_flags;        /\* master MTD flags to mask out for this partition \*/  
    struct nand\_ecclayout \*ecclayout;    /\* out of band layout for this partition (NAND only)\*/  
};

一个MTD原始设备可以通过mtd\_part分割成数个MTD原始设备注册进mtd\_table，mtd\_table中的每个MTD原始设备都可以被注册成一个MTD设备，有两个函数可以完成这个工作，即add\_mtd\_device函数和add\_mtd\_partitions函数。

其中add\_mtd\_device函数是把整个NAND FLASH注册进MTD Core，而add\_mtd\_partitions函数则是把NAND FLASH的各个分区分别注册进MTD Core。

int add\_mtd\_device(struct mtd\_info \*mtd)

int del\_mtd\_device(struct mtd\_info \*mtd)

int add\_mtd\_partitions(struct mtd\_info \*master,structmtd\_partitions \*parts,int nbparts)

int del\_mtd\_partitions(struct mtd\_info \*master)

当MTD原始设备调用add\_mtd\_partitions()的时候它会对每一个新建分区建立一个struct mtd\_part 结构将其加入mtd\_partitions中并调用add\_mtd\_device()将此分区做为MTD设备注册进MTD Core,成功的时返回0

重点关注一下add\_mtd\_partitions(struct mtd\_info \*master,struct mtd\_partitions\*parts,int nbparts)其中master就是这个MTD原始设备，parts即NAND的分区信息，nbparts指有几个分区。那么parts和nbparts怎么来的呢,其实上面有一句话已经说了。。举个例子在我们移植Linux内核到我们的开发板的时候我们会对NANDFLASH进行分区 这eilian240\_default\_nand\_part就是起到上面这两个参数的作用如下：

static struct mtd\_partition eilian240\_default\_nand\_part[] = {  
    [0] = {  
        .name    = "bootloader",/\*uboot存放的地址对应dev/mtdblock0\*/  
        .size    = 0x00040000,  /\*大小256KB=((D)(0x00040000))/1024\*/  
        .offset    = 0,  
    },  
    [1] = {  
        .name    = "param",  
        .offset = 0x00040000,/\*\*\*如果UBOOT比较大就放在这个区域可以将前面的覆盖掉\*\*//\*\*0x00040000是偏移量\*\*/  
        .size    = 0x00020000,  /\*\*128KB\*\*/  
    },  
    [2] = {  
        .name    = "Linux Kernel",/\*\*\*\*\*\*\*\*用于存放Linux内核镜像dev/mtdblock3\*/  
        .offset = 0x00060000,  
        .size    = 0x00500000,        /\*5M\*/  
    },  
    [3] = {  
        .name    = "rootfs",  
        .offset = 0x00560000,  
        .size    = 1024 \* 1024 \* 1024, //1G因为该移植同时支持大容量的NAND  
    },  
    [4] = {  
        .name    = "nand",  
        .offset = 0x00000000,  
        .size    = 1024 \* 1024 \* 1024, //  
    }  
};  
/\*arch/arm/plat-samsung/include/plat/nand.h \*\*/

[基于MTD的NANDFLASH设备驱动底层实现原理分析(五)](http://blog.csdn.net/eilianlau/article/details/6963225)

分类： [Linux驱动开发](http://blog.csdn.net/eilianlau/article/category/922573)2011-11-12 17:39 1791人阅读 [评论](http://blog.csdn.net/eilianlau/article/details/6963225#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/eilianlau/article/details/6963225#report)

[struct](http://www.csdn.net/tag/struct)[linux内核](http://www.csdn.net/tag/linux%e5%86%85%e6%a0%b8)[数据结构](http://www.csdn.net/tag/%e6%95%b0%e6%8d%ae%e7%bb%93%e6%9e%84)[command](http://www.csdn.net/tag/command)[cmd](http://www.csdn.net/tag/cmd)[存储](http://www.csdn.net/tag/%e5%ad%98%e5%82%a8)

Linux内核在MTD的下层实现了通用的NAND驱动(/driver/mtd/nand/nand\_base.c)因此芯片级的驱动实现不再需要我们关心mtd中的那些成员函数了主题转移到nand\_chip数据结构中

先了解了解nand\_chip结构体

struct nand\_chip {  
    void  \_\_iomem    \*IO\_ADDR\_R;    //读8位I/O线的地址  
    void  \_\_iomem    \*IO\_ADDR\_W;   //写8位I/O线的地址  
    uint8\_t        (\*read\_byte)(struct mtd\_info \*mtd);//从芯片读一个字节  
    u16        (\*read\_word)(struct mtd\_info \*mtd);//从芯片读一个字  
    void        (\*write\_buf)(struct mtd\_info \*mtd, const uint8\_t \*buf, int len);//将缓冲区的数据写入芯片  
    void        (\*read\_buf)(struct mtd\_info \*mtd, uint8\_t \*buf, int len);//将芯片中的数据独到缓冲区中

    int        (\*verify\_buf)(struct mtd\_info \*mtd, const uint8\_t \*buf, int len); //验证芯片和写入缓冲区中的数据

   int         (\*block\_bad)(struct mtd\_info \*mtd, loff\_t ofs, int getchip);//检查是否坏块  
    int        (\*block\_markbad)(struct mtd\_info \*mtd, loff\_t ofs);//标记坏块

    void        (\*select\_chip)(struct mtd\_info \*mtd, int chip);          //实现选中芯片  
    void        (\*cmd\_ctrl)(struct mtd\_info \*mtd, int dat,                 
                    unsigned int ctrl);//控制ALE/CLE/nCE,也用于写命令和地址  
    int        (\*dev\_ready)(struct mtd\_info \*mtd);//设备就绪  
    void        (\*cmdfunc)(struct mtd\_info \*mtd, unsigned command, int column, int page\_addr); //实现命令发送

    int        (\*waitfunc)(struct mtd\_info \*mtd, struct nand\_chip \*this);  
    void        (\*erase\_cmd)(struct mtd\_info \*mtd, int page);//擦除命令的处理  
    int        (\*scan\_bbt)(struct mtd\_info \*mtd);//扫描坏块  
    int        (\*errstat)(struct mtd\_info \*mtd, struct nand\_chip \*this, int state, int status, int page);

    int        (\*write\_page)(struct mtd\_info \*mtd, struct nand\_chip \*chip,  
                      const uint8\_t \*buf, int page, int cached, int raw);//写一页

    int        chip\_delay;//有板决定的延迟时间  
   unsigned int    options;//与具体的NAND芯片相关的一些选项，如NAND\_NO\_AUTOINCR，NAND\_BUSWIDTH\_16等，至于这些选项具体表示什么含义，可以参考<linux/mtd/nand.h>，那里有较为详细的说明；  
  
    int        page\_shift;//用位表示的NAND芯片的page大小，如某片NAND芯片的一个page有512个字节，那么page\_shift就是9；  
    int        phys\_erase\_shift;//用位表示的NAND芯片的每次可擦除的大小，如某片NAND芯片每次可擦除16K字节(通常就是一个block的大小)，那么phys\_erase\_shift就是14；  
    int        bbt\_erase\_shift;//用位表示的bad block table的大小，通常一个bbt占用一个block，所以bbt\_erase\_shift通常与phys\_erase\_shift相等；  
    int        chip\_shift;用位表示的NAND芯片的容量；  
    int        numchips;表示系统中有多少片NAND芯片；  
    uint64\_t    chipsize;//NAND芯片的大小；  
    int        pagemask;//计算page number时的掩码，总是等于chipsize/page大小－ 1；  
    int        pagebuf;用来保存当前读取的NAND芯片的page number，这样一来，下次读取的数据若还是属于同一个page，就不必再从NAND芯片读取了，而是从data\_buf中直接得到；  
    int        subpagesize;  
    uint8\_t        cellinfo;  
    int        badblockpos;  
    int        badblockbits;  
  
    flstate\_t    state;  
  
    uint8\_t        \*oob\_poi;  
    struct nand\_hw\_control  \*controller;  
    struct nand\_ecclayout    \*ecclayout;  
  
    struct nand\_ecc\_ctrl ecc;  
    struct nand\_buffers \*buffers;  
    struct nand\_hw\_control hwcontrol;  
  
    struct mtd\_oob\_ops ops;  
  
    uint8\_t        \*bbt;  
    struct nand\_bbt\_descr    \*bbt\_td;  
    struct nand\_bbt\_descr    \*bbt\_md;  
  
    struct nand\_bbt\_descr    \*badblock\_pattern;  
  
    void        \*priv;

};

这上面有一个与ECC相关的结构体 struct nand\_ecc\_ctrl

struct nand\_ecc\_ctrl {  
    nand\_ecc\_modes\_t    mode;  
    int            steps;  
    int            size;  
    int            bytes;  
    int            total;  
    int            prepad;  
    int            postpad;  
    struct nand\_ecclayout    \*layout;  
    void            (\*hwctl)(struct mtd\_info \*mtd, int mode);/\*\*控制硬件ECC\*/  
    int            (\*calculate)(struct mtd\_info \*mtd,  
                         const uint8\_t \*dat,  
                         uint8\_t \*ecc\_code);/\*\*根据data计算ecc值\*\*/  
    int            (\*correct)(struct mtd\_info \*mtd, uint8\_t \*dat,  
                       uint8\_t \*read\_ecc,  
                       uint8\_t \*calc\_ecc);  
    int            (\*read\_page\_raw)(struct mtd\_info \*mtd,  
                         struct nand\_chip \*chip,  
                         uint8\_t \*buf, int page);/\*\*向NANDFLASH芯片读一个页的原始数据\*/  
    void            (\*write\_page\_raw)(struct mtd\_info \*mtd,  
                          struct nand\_chip \*chip,  
                          const uint8\_t \*buf);  
    int            (\*read\_page)(struct mtd\_info \*mtd,  
                         struct nand\_chip \*chip,  
                         uint8\_t \*buf, int page);/\*\*读一个页但包含ecc校验\*/  
    int            (\*read\_subpage)(struct mtd\_info \*mtd,  
                         struct nand\_chip \*chip,  
                         uint32\_t offs, uint32\_t len,  
                         uint8\_t \*buf);  
    void            (\*write\_page)(struct mtd\_info \*mtd,  
                          struct nand\_chip \*chip,  
                          const uint8\_t \*buf);  
    int            (\*read\_oob)(struct mtd\_info \*mtd,  
                        struct nand\_chip \*chip,  
                        int page,  
                        int sndcmd);/\*\*读OOB但不包含MAIN部分\*\*/  
    int            (\*write\_oob)(struct mtd\_info \*mtd,  
                         struct nand\_chip \*chip,  
                         int page);  
};

hwctl：这个函数用来控制硬件产生ecc，其实它主要的工作就是控制NAND controller向NAND芯片发出NAND\_ECC\_READ、NAND\_ECC\_WRITE和NAND\_ECC\_READSYN等命令，与struct nand\_chip结构体中的cmdfunc类似，只不过发起的命令是ECC相关的罢了；

calculate：根据data计算ecc值；

correct：根据ecc值，判断读写数据时是否有错误发生，若有错，则立即试着纠正，纠正失败则返回错误；

read\_page\_raw和write\_page\_raw：从NAND芯片中读取一个page的原始数据和向NAND芯片写入一个page的原始数据，所谓的原始数据，即不对读写的数据做ecc处理，该读写什么值就读写什么值。另外，这两个函数会读写整个page中的所有内容，即不但会读写一个page中MAIN部分，还会读写OOB部分。

read\_page和write\_page：与read\_page\_raw和write\_page\_raw类似，但不同的是，read\_page和write\_page在读写过程中会加入ecc的计算，校验，和纠正等处理。

read\_oob和write\_oob：读写oob中的内容，不包括MAIN部分。

其实，以上提到的这几个read\_xxx和write\_xxx函数，最终都会调用struct nand\_chip中的read\_buf和write\_buf这两个函数，所以如果没有特殊需求的话，我认为不必自己实现，使用MTD提供的default的函数即可。

下面这个结构体用来ECC在oob中布局的一个结构体。

struct nand\_ecclayout {  
    \_\_u32 eccbytes;//ecc字节数,对于512字节/page的NANDflash,eccbytes=3,如果需要额外用到oob中的数据，那么也可以大于3.  
    \_\_u32 eccpos[64];ECC数据在oob中的位置，这里之所以是个64字节的数组，是因为对于2K/页的NAND来说，它的oob有64个字节。而对于512字节/page的NAND来说，可以而且只可以定义它的前16个字节。  
    \_\_u32 oobavail;OOB中可用的字节数,不需要对该成员赋值,MTD会根据其他三个成员计算出来  
    struct nand\_oobfree oobfree[MTD\_MAX\_OOBFREE\_ENTRIES];//显示定义空闲的OOB字节  
};

大家都知道OOB但是OOB里面究竟存了些什么,OOB主要用来存储两种信息：坏块信息和ECC数据,对于小页的NANDFLASH一般坏块占据一个字节(并且是在第6个字节),ECC占3个字节,上面这个结构体就是起到了这个作用告诉那些与操作ECC无关的函数,在OOB区域里那部分是来存储ECC的(不可他用),那些字节是空闲的。

[基于MTD的NANDFLASH设备驱动底层实现原理分析(六)](http://blog.csdn.net/eilianlau/article/details/6965749)

分类： [Linux驱动开发](http://blog.csdn.net/eilianlau/article/category/922573)2011-11-13 13:42 2067人阅读 [评论](http://blog.csdn.net/eilianlau/article/details/6965749#comments)(4) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/eilianlau/article/details/6965749#report)

[struct](http://www.csdn.net/tag/struct)[c](http://www.csdn.net/tag/c)[module](http://www.csdn.net/tag/module)[null](http://www.csdn.net/tag/null)[平台](http://www.csdn.net/tag/%e5%b9%b3%e5%8f%b0)[flash](http://www.csdn.net/tag/flash)

实在写不下去了,仔细的想了一想还是把mtd/nand/s3c2410.c好好的分析分析

在Linux中NANDFLASH设备驱动是被注册成平台驱动的。我还是从函数的入口出发一步一个脚印的分析。突然间发现这些代码真的很经典

由于这一次CPU是S3C2440所以分析过程中会把其他的CPU去掉

七、mtd/nand/s3c2410.c函数的解析

1、函数中出现的几个结构体

struct s3c24xx\_nand\_mtd {  
    struct mtd\_info            mtd;/\*用来表述MTD原始设备的结构\*/  
    struct nand\_chip        chip;/\*该结构体表示一个NAND芯片\*/

   //该结构定义在arch/arm/plat-s3c/include/plat/nand.h中,该结构体包含了一个或多个的NAND芯片的信息

    struct s3c2410\_nand\_set        \*set;  
    struct s3c24xx\_nand\_info    \*info;  
    int                scan\_res;  
};  
/\*\*CPU选型的枚举\*\*/  
enum s3c\_cpu\_type {  
    TYPE\_S3C2410,  
    TYPE\_S3C2440,  
};  
/\*\*NANDFLASH控制器状态的结构\*\*/  
struct s3c24xx\_nand\_info {  
    /\* mtd info \*/

   //include/linux/mtd/nand.h硬件控制器结构

    struct nand\_hw\_control        controller;  
    struct s3c24xx\_nand\_mtd        \*mtds;

   //NANDFLASH平台驱动结构

    struct s3c2410\_platform\_nand    \*platform;  
  
    /\* device info \*/  
    struct device            \*device;  
    struct resource            \*area;  
    struct clk            \*clk;  
    void \_\_iomem            \*regs;  
    void \_\_iomem            \*sel\_reg;  
    int                sel\_bit;  
    int                mtd\_count;  
    unsigned long            save\_sel;  
    unsigned long            clk\_rate;  
  
    enum s3c\_cpu\_type        cpu\_type;  
  
#ifdef CONFIG\_CPU\_FREQ  
    struct notifier\_block    freq\_transition;  
#endif  
};

2、函数入口和出口

static struct platform\_driver s3c24xx\_nand\_driver = {  
    .probe        = s3c24xx\_nand\_probe,  
    .remove        = s3c24xx\_nand\_remove,  
    .suspend    = s3c24xx\_nand\_suspend,  
    .resume        = s3c24xx\_nand\_resume,  
    .id\_table    = s3c24xx\_driver\_ids,  
    .driver        = {  
        .name    = "s3c24xx-nand",  
        .owner    = THIS\_MODULE,  
    },  
};  
  
static int \_\_init s3c2410\_nand\_init(void)  
{  
    printk("S3C24XX NAND Driver, (c) 2004 Simtec Electronics\n");

    //这些已经没什么好解释的了。注册平台驱动  
    return platform\_driver\_register(&s3c24xx\_nand\_driver);  
}  
  
static void \_\_exit s3c2410\_nand\_exit(void)  
{

    //当我们卸载内核模块的时候会呗调用 注销平台驱动

    platform\_driver\_unregister(&s3c24xx\_nand\_driver);  
}  
  
module\_init(s3c2410\_nand\_init);  
module\_exit(s3c2410\_nand\_exit);  
  
MODULE\_LICENSE("GPL");  
MODULE\_AUTHOR("Ben Dooks <ben@simtec.co.uk>");  
MODULE\_DESCRIPTION("S3C24XX MTD NAND driver");  
3、NANDFLASH中的s3c24xx\_nand\_probe函数。

<随便提一句>可能我们都懂当我们向系统注册一个平台驱动的时候,会去遍历系统上的平台设备,找到能够与之相对应的平台设备。当找到可用的设备后该函数才会被调用<称之探测函数>对于外设来说驱动的本身应该是从这个探测函数真正开始的,那么它究竟有什么用呢?它到底做了什么事情,这可能是我们最关心的。在MTD/NAND的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yuan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。如果说把这个函数理解透了那么整个驱动起码是完成三分之二的工作了。在s3c2410.c中s3c24xx\_nand\_probe函数特别的长,所以一段一段的分析.

static int s3c24xx\_nand\_probe(struct platform\_device \*pdev)  
{

   /\*\*从总线设备中取出平台设备的数据,我一直在思索最后这个结构会从哪里传进来呢?\*\*如果你移植过Linux到你的开发板那么你一定会在你的板层文件中构建这样一个结构体请看下面步骤a)中的结构体\*/

    struct s3c2410\_platform\_nand \*plat = to\_nand\_plat(pdev);

    enum s3c\_cpu\_type cpu\_type;  
    struct s3c2410\_nand\_info \*info;  
    struct s3c2410\_nand\_mtd \*nmtd;  
    struct s3c2410\_nand\_set \*sets;  
    struct resource \*res;  
    int err = 0;  
    int size;  
    int nr\_sets;  
    int setno;  
    /\*获取CPU类型\*/  
    cpu\_type = platform\_get\_device\_id(pdev)->driver\_data;  
  
    pr\_debug("s3c2410\_nand\_probe(%p)\n", pdev);  
    /\*为s3c2410\_nand\_info分配内存\*/  
    info = kmalloc(sizeof(\*info), GFP\_KERNEL);  
    if (info == NULL) {  
        dev\_err(&pdev->dev, "no memory for flash info\n");  
        err = -ENOMEM;  
        goto exit\_error;  
    }  
    /\*\*初始化s3c2410\_nand\_info\*\*/  
    memset(info, 0, sizeof(\*info));  
    /\*\*将nand设备的数据信息传递到系统平台设备中去\*\*/  
    platform\_set\_drvdata(pdev, info);  
    /\*\*初始化自旋锁和等待队列\*\*/  
    spin\_lock\_init(&info->controller.lock);  
    init\_waitqueue\_head(&info->controller.wq);  
  
    /\* get the clock source and enable it \*/  
  
    info->clk = clk\_get(&pdev->dev, "nand");  
    if (IS\_ERR(info->clk)) {  
        dev\_err(&pdev->dev, "failed to get clock\n");  
        err = -ENOENT;  
        goto exit\_error;  
    }  
  
    clk\_enable(info->clk);//使能时钟  
  
    /\* allocate and map the resource \*/  
       
    /\* currently we assume we have the one resource \*/  
    res  = pdev->resource;  
    size = res->end - res->start + 1;  
    /\*\*为资源申请IO内存\*\*/  
    info->area = request\_mem\_region(res->start, size, pdev->name);  
  
    if (info->area == NULL) {  
        dev\_err(&pdev->dev, "cannot reserve register region\n");  
        err = -ENOENT;  
        goto exit\_error;  
    }  
    /\*\*\*\*\*这都是给info这个结构的成员变量赋值啊\*没啥可说的,\*/  
    info->device     = &pdev->dev;  
    info->platform   = plat;  
    info->regs       = ioremap(res->start, size);  
    info->cpu\_type   = cpu\_type;  
  
    if (info->regs == NULL) {  
        dev\_err(&pdev->dev, "cannot reserve register region\n");  
        err = -EIO;  
        goto exit\_error;  
    }  
  
    dev\_dbg(&pdev->dev, "mapped registers at %p\n", info->regs);  
  
    /\* initialise the hardware \*//\*\*解析c\*/  
    /\*\*初始化硬件\*\*/  
    err = s3c2410\_nand\_inithw(info);  
    if (err != 0)  
        goto exit\_error;  
     /\*struct s3c2410\_platform\_nand 中包含了一个struct s3c2410\_nand\_set \*sets  
    的成员变量 专门用来描述芯片信息的\*\*/  
    sets = (plat != NULL) ? plat->sets : NULL;  
    nr\_sets = (plat != NULL) ? plat->nr\_sets : 1;   
  
    info->mtd\_count = nr\_sets;  
  
    /\* allocate our information \*/  
  
    size = nr\_sets \* sizeof(\*info->mtds);  
    info->mtds = kmalloc(size, GFP\_KERNEL);  
    if (info->mtds == NULL) {  
        dev\_err(&pdev->dev, "failed to allocate mtd storage\n");  
        err = -ENOMEM;  
        goto exit\_error;  
    }  
  
    memset(info->mtds, 0, size);  
  
    /\* initialise all possible chips \*/  
  
    nmtd = info->mtds;  
  
    for (setno = 0; setno < nr\_sets; setno++, nmtd++) {  
        pr\_debug("initialising set %d (%p, info %p)\n", setno, nmtd, info);  
       /\*\*nand\_chip结构的初始化\*\*见下一遍分析d)\*\*/  
        s3c2410\_nand\_init\_chip(info, nmtd, sets);  
        /\*\*读取芯片的ID\*\*/  
        nmtd->scan\_res = nand\_scan\_ident(&nmtd->mtd,  
                         (sets) ? sets->nr\_chips : 1);  
  
        if (nmtd->scan\_res == 0) {  
            s3c2410\_nand\_update\_chip(info, nmtd);  
            nand\_scan\_tail(&nmtd->mtd);  
            s3c2410\_nand\_add\_partition(info, nmtd, sets);  
        }  
  
        if (sets != NULL)  
            sets++;  
    }  
  
    err = s3c2410\_nand\_cpufreq\_register(info);  
    if (err < 0) {  
        dev\_err(&pdev->dev, "failed to init cpufreq support\n");  
        goto exit\_error;  
    }  
  
    if (allow\_clk\_stop(info)) {  
        dev\_info(&pdev->dev, "clock idle support enabled\n");  
        clk\_disable(info->clk);  
    }  
  
    pr\_debug("initialised ok\n");  
    return 0;  
  
 exit\_error:  
    s3c24xx\_nand\_remove(pdev);  
  
    if (err == 0)  
        err = -EINVAL;  
    return err;  
}

我真不知道该怎么去分析这个程序。

    a)分析该函数的第一行的代码我就迷茫了呵呵.看看下面这个结构体.你是否会发现什么  
static struct s3c2410\_platform\_nand eilian240\_nand\_info = {  
    .tacls        = 20,  
    .twrph0        = 60,  
    .twrph1        = 20,  
    .nr\_sets    = ARRAY\_SIZE(eilian240\_nand\_sets),  
    .sets        = eilian240\_nand\_sets,  
    .ignore\_unset\_ecc = 1,  
};发现了吧这些芯片硬件信息是在我们移植的时候才传进去的<所以这就和目标板有关系了>

  那么这些芯片信息又是怎样和系统里面的总线设备关联起来的呢?看下面这三句

   info = kmalloc(sizeof(\*info), GFP\_KERNEL); 为info结构分配内存

   memset(info, 0, sizeof(\*info));              //初始化内存  
   platform\_set\_drvdata(pdev, info); //别小看这一句,功能可强大了,这些芯片信息就是通过它传递到总线设备里面去的,看下源代码你就明白了

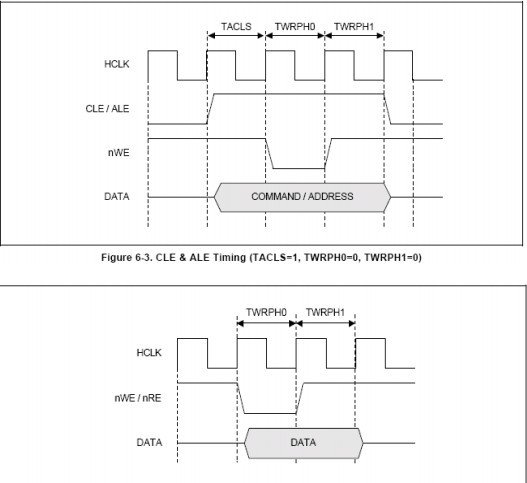
    b)关于时钟使能和平台设备驱动程序,想设备申请资源这是必须要做的事情,我解释不了,表达能力很有限哦。

    c)硬件的初始化函数s3c2410\_nand\_inithw(info);其实就是在设置一些寄存器和时钟的频率,看看它是怎么实现的

static int s3c2410\_nand\_inithw(struct s3c2410\_nand\_info \*info)  
{  
    int ret;  
  
    ret = s3c2410\_nand\_setrate(info);  
    if (ret < 0)  
        return ret;  
  
     switch (info->cpu\_type) {  
     case TYPE\_S3C2410:  
    default:  
        break;  
  
     case TYPE\_S3C2440:  
     case TYPE\_S3C2412:  
        /\* enable the controller and de-assert nFCE \*/  
        writel(S3C2440\_NFCONT\_ENABLE, info->regs + S3C2440\_NFCONT);  
    }  
  
    return 0;  
}

static int s3c2410\_nand\_setrate(struct s3c2410\_nand\_info \*info)  
{  
    struct s3c2410\_platform\_nand \*plat = info->platform;  
    int tacls\_max = (info->cpu\_type == TYPE\_S3C2412) ? 8 : 4;  
    int tacls, twrph0, twrph1;  
    /\*\*获取时钟频率\*\*/  
    unsigned long clkrate = clk\_get\_rate(info->clk);//这个时钟我们在之前已经获取了吧  
    unsigned long uninitialized\_var(set), cfg, uninitialized\_var(mask);//这其实都是一些宏,看看源代码就知道了    unsigned long flags;  
  
    /\* calculate the timing information for the controller \*/  
  
    info->clk\_rate = clkrate;  
    clkrate /= 1000;    /\* turn clock into kHz for ease of use \*/  
  
    if (plat != NULL) {  
        tacls = s3c\_nand\_calc\_rate(plat->tacls, clkrate, tacls\_max);  
        twrph0 = s3c\_nand\_calc\_rate(plat->twrph0, clkrate, 8);  
        twrph1 = s3c\_nand\_calc\_rate(plat->twrph1, clkrate, 8);  
    } else {  
        /\* default timings \*/  
        tacls = tacls\_max;  
        twrph0 = 8;  
        twrph1 = 8;  
    }  
  
    if (tacls < 0 || twrph0 < 0 || twrph1 < 0) {  
        dev\_err(info->device, "cannot get suitable timings\n");  
        return -EINVAL;  
    }  
  
    dev\_info(info->device, "Tacls=%d, %dns Twrph0=%d %dns, Twrph1=%d %dns\n",  
           tacls, to\_ns(tacls, clkrate), twrph0, to\_ns(twrph0, clkrate), twrph1, to\_ns(twrph1, clkrate));  
  
    switch (info->cpu\_type) {  
。。。。。。。  
    case TYPE\_S3C2440:  
    case TYPE\_S3C2412:  
        mask = (S3C2440\_NFCONF\_TACLS(tacls\_max - 1) |  
            S3C2440\_NFCONF\_TWRPH0(7) |  
            S3C2440\_NFCONF\_TWRPH1(7));  
  
        set = S3C2440\_NFCONF\_TACLS(tacls - 1);  
        set |= S3C2440\_NFCONF\_TWRPH0(twrph0 - 1);  
        set |= S3C2440\_NFCONF\_TWRPH1(twrph1 - 1);  
        break;  
  
    default:  
        BUG();  
    }  
  
    local\_irq\_save(flags);  
  
    cfg = readl(info->regs + S3C2410\_NFCONF);  
    cfg &= ~mask;  
    cfg |= set;  
    writel(cfg, info->regs + S3C2410\_NFCONF);  
  
    local\_irq\_restore(flags);  
  
    dev\_dbg(info->device, "NF\_CONF is 0x%lx\n", cfg);  
  
    return 0;  
}

在分析这几段代码的时候我为Nandflash存储的时序纠结了好久,看看下面这个图,S3C2440数据手册上的

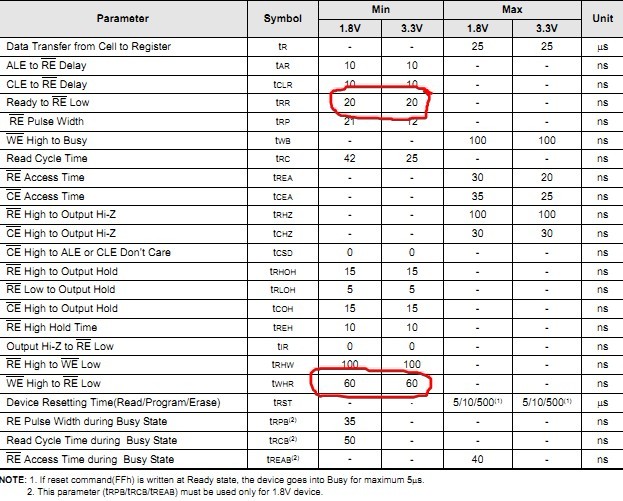


TACLS：当CLE/ALE使能后再过多长的时间nWe才有效/\* time for active CLE/ALE to nWE/nOE \*/

TWRPH0：nWE/nRE的有效时间的长度                       /\* active time for nWE/nOE \*/

TWRPH1：nWE/nRE无效开始到CLE/ALE也无效的时间的长度 /\* time for release CLE/ALE from nWE/nOE inactive \*/

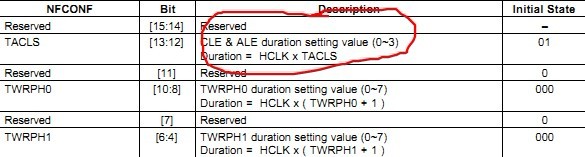
不知道是不是这样的....如果分析错了有大神看到请指出。在k9f2g08上有这样一个表



所以有a中的：

    .tacls            = 20,  
    .twrph0        = 60,  
    .twrph1        = 20,    不过到这个地方我还是有一点点的迷惑不知道是我的数据手册错了还是什么,随便找了一个时序看了看时间不对啊。。。。。

 int tacls\_max = (info->cpu\_type == TYPE\_S3C2412) ? 8 : 4;//这一句先看看下面这个表2<对下面分析还要用到的>



红色标注的部分不是明明是0---3么 怎么这里设置的4呢?别急继续往下看。。

tacls ,twrph0,twrph1它是怎么进行转换的呢

static int s3c\_nand\_calc\_rate(int wanted, unsigned long clk, int max)  
{  
    int result;  
    /\*\*[(wanted \* clk)+NS\_IN\_KHZ)-1]/NS\_IN\_KHZ \*/  
    result = DIV\_ROUND\_UP((wanted \* clk), NS\_IN\_KHZ);  
   /\*\* #define NS\_IN\_KHZ 1000000      //原本单位是HZ现在转换成KHZ\*\*/  
    pr\_debug("result %d from %ld, %d\n", result, clk, wanted);  
  
    if (result > max) {  
        printk("%d ns is too big for current clock rate %ld\n", wanted, clk);  
        return -1;  
    }  
  
    if (result < 1)  
        result = 1;  
  
    return result;  
}

现在再往上看那段if(plat != NULL){。。。。。。。。}或许明白了

再返回 int s3c2410\_nand\_setrate(struct s3c2410\_nand\_info \*info)中下面这段代码

switch (info->cpu\_type) {  
。。。。。。。  
    case TYPE\_S3C2440:  
    case TYPE\_S3C2412:

        /\*\*好了现在来了个tacls\_max - 1或许就明白了上面为何是4而不是3了。这些代码都是在给NANDFLASH配置寄存器的相应的位赋值,不过还没有写进去。后面有写入的操作的。\*/

        mask = (S3C2440\_NFCONF\_TACLS(tacls\_max - 1) |  
            S3C2440\_NFCONF\_TWRPH0(7) |  
            S3C2440\_NFCONF\_TWRPH1(7));  
  
        set = S3C2440\_NFCONF\_TACLS(tacls - 1);  
        set |= S3C2440\_NFCONF\_TWRPH0(twrph0 - 1);  
        set |= S3C2440\_NFCONF\_TWRPH1(twrph1 - 1);  
        break;

        。。。。。。。。。。。。。

      /\*\*将这些数据写进寄存器中去\*/

    cfg = readl(info->regs + S3C2410\_NFCONF);  
    cfg &= ~mask;  
    cfg |= set;  
    writel(cfg, info->regs + S3C2410\_NFCONF);

再返回static int s3c2410\_nand\_inithw(struct s3c2410\_nand\_info \*info)有这么一断代码

。。。。。。。。

   case TYPE\_S3C2440:  
     case TYPE\_S3C2412:  
        /\* enable the controller and de-assert nFCE \*/  
      /\*\*\*\*\*使能NANDFLASH控制寄存器\*\*\*\*\*\*/  
        writel(S3C2440\_NFCONT\_ENABLE, info->regs + S3C2440\_NFCONT);  
    }

/\*\*\*\*\*\*\*\*\*\*\*到这里为止这个硬件的初始化工作就结束了\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

  上面基本的硬件初始化完后应该是要初始化nand\_chip实例了,并运行nand\_scan()扫描NAND设备了,最后添加板文件中的分区表。nand\_chip是NANDFLASH驱动的核心结构上一遍文章已经分析过了。还是放到下一遍文章吧

# [基于MTD的NANDFLASH设备驱动底层实现原理分析(七)](http://blog.csdn.net/eilianlau/article/details/6966284)

分类： [Linux驱动开发](http://blog.csdn.net/eilianlau/article/category/922573)2011-11-13 18:21 1700人阅读 [评论](http://blog.csdn.net/eilianlau/article/details/6966284#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/eilianlau/article/details/6966284#report)

[struct](http://www.csdn.net/tag/struct)[c](http://www.csdn.net/tag/c)[table](http://www.csdn.net/tag/table)[layout](http://www.csdn.net/tag/layout)[delay](http://www.csdn.net/tag/delay)[module](http://www.csdn.net/tag/module)

上接：

### [基于MTD的NANDFLASH设备驱动底层实现原理分析(六)](http://blog.csdn.net/eilianlau/article/details/6965749)

初始化基本的硬件配置后probe函数就会开始与NAND芯片进行交互了，它要做的事情主要包括这几个方面：读取NAND芯片的ID，然后查表得到这片NAND芯片的如厂商，page size，erase size以及chip size等信息，接着，根据struct nand\_chip中options的值的不同，或者在NAND芯片中的特定位置查找bad block table，或者scan整个NAND芯片，并在内存中建立bad block table。这些都由nand\_scan()完成。

nand\_scan函数主要有两个两个函数组成，即nand\_scan\_ident函数和nand\_scan\_tail函数。其中nand\_scan\_ident函数会读取NAND芯片的ID，而nand\_scan\_tail函数则会查找或者建立bbt (bad block table)。

最后调用add\_mtd\_partitions()添加板层文件platform中定义的分区表。

      d)nand\_chip的初始化,关于nand\_chip上第5编文章中已介绍

 /\*\*初始化nand\_chip结构\*\*/  
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;//&nmtd->chip=&(nmtd->chip)  
    void \_\_iomem \*regs = info->regs;//用于保存地址的。。  
     /\*\*下面这一段是在给nand\_chip中的函数指针赋值\*\*/  
    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;//延迟时间  
    chip->priv       = nmtd; //这个是很重要的将struct s3c2410\_nand\_mtd赋值给nand\_chip的私有数据成员  
    chip->options       = 0;//在第5篇文章中有介绍这个地方好像错了,或许后面会有赋值,因为没有0定义这个宏  
    chip->controller   = &info->controller;//指向struct nand\_hw\_control  的指针  
 。。。。。。。。。。。。。。。。  
    case TYPE\_S3C2440:  
        chip->IO\_ADDR\_W = regs + S3C2440\_NFDATA;//2440NAND数据寄存器  
        info->sel\_reg   = regs + S3C2440\_NFCONT;//2440NAND控制寄存器  
        info->sel\_bit    = S3C2440\_NFCONT\_nFCE;//1<<1，此刻芯片片选信号为disable,默认就是disable的  
        chip->cmd\_ctrl  = s3c2440\_nand\_hwcontrol;//控制ALE/CLE/nCE,也用于写命令和地址  
        chip->dev\_ready = s3c2440\_nand\_devready;//设备就绪  
        chip->read\_buf  = s3c2440\_nand\_read\_buf;//将芯片中的数据读到缓冲区中  
        chip->write\_buf    = s3c2440\_nand\_write\_buf;//将缓冲区中的数据写入芯片  
        break;

        。。。。。。。。。。。。。。。。  
      }  
  
    chip->IO\_ADDR\_R = chip->IO\_ADDR\_W;  
  
    nmtd->info       = info;  
    nmtd->mtd.priv       = chip; //把指向struct nand\_chip结构体的指针赋给struct mtd\_info的priv成员变量，因为MTD Core中很多函数之间的调用都只传递struct mtd\_info，它需要通过priv成员变量得到struct nand\_chip。  
    nmtd->mtd.owner    = THIS\_MODULE;  
    nmtd->set       = set;  
    /\*\*如果采用硬件ECC\*\*/  
    if (hardware\_ecc) {  
        chip->ecc.calculate = s3c2410\_nand\_calculate\_ecc;  
        chip->ecc.correct   = s3c2410\_nand\_correct\_data;  
        chip->ecc.mode        = NAND\_ECC\_HW;  
        switch (info->cpu\_type) {  
     。。。。。。。。。。。。。。。。  
        case TYPE\_S3C2440:  
              chip->ecc.hwctl     = s3c2440\_nand\_enable\_hwecc;  
              chip->ecc.calculate = s3c2440\_nand\_calculate\_ecc;  
            break;  
        }  
    } else {

       //使用软件校验

        chip->ecc.mode        = NAND\_ECC\_SOFT;  
    }  
      /\*\*ECC\*/  
    if (set->ecc\_layout != NULL)  
        chip->ecc.layout = set->ecc\_layout;  
     /\*\*禁止ECC\*/  
    if (set->disable\_ecc)  
        chip->ecc.mode    = NAND\_ECC\_NONE;  
  
    switch (chip->ecc.mode) {  
    。。。。。。。。。。。  
    }  
  
    /\* If you use u-boot BBT creation code, specifying this flag will  
     \* let the kernel fish out the BBT from the NAND, and also skip the  
     \* full NAND scan that can take 1/2s or so. Little things... \*/  
    if (set->flash\_bbt)//当flashbbt=1的时候系统在启动的时候将跳过对bbt的扫描  
        chip->options |= NAND\_USE\_FLASH\_BBT | NAND\_SKIP\_BBTSCAN;  
}

接下来是读取芯片的ID调用int nand\_scan\_ident(struct mtd\_info \*mtd, int maxchips)函数该函数是通用的,定义在nand\_base.c中暂时还没研究过

/\*\*读取芯片的ID\*\*/  
        nmtd->scan\_res = nand\_scan\_ident(&nmtd->mtd,  
                         (sets) ? sets->nr\_chips : 1);  
        
        if (nmtd->scan\_res == 0) {  /\*\*如果读取成功则返回0\*\*/  
            s3c2410\_nand\_update\_chip(info, nmtd);//更新,下面看看它的原型  
            nand\_scan\_tail(&nmtd->mtd);//查找或者建立bbt(bad block table)  
            s3c2410\_nand\_add\_partition(info, nmtd, sets);//添加分区,与板层文件相关

           //这个函数的实现那肯定就是调用add\_mtd\_device(&mtd->mtd)。。。

        }

    e)s3c2410\_nand\_update\_chip(struct s3c2410\_nand\_info \*info,  
                     struct s3c2410\_nand\_mtd \*nmtd)分析

static void s3c2410\_nand\_update\_chip(struct s3c2410\_nand\_info \*info,  
                     struct s3c2410\_nand\_mtd \*nmtd)  
{  
    struct nand\_chip \*chip = &nmtd->chip;  
  
    dev\_dbg(info->device, "chip %p => page shift %d\n",  
        chip, chip->page\_shift);  
  
    if (chip->ecc.mode != NAND\_ECC\_HW)//如果不是硬件校验则直接返回  
        return;  
  
        /\* change the behaviour depending on wether we are using  
         \* the large or small page nand device \*/  
  
    if (chip->page\_shift > 10) { //page\_shift用位来表示页的大小 大于2KB的大页  
        chip->ecc.size        = 256;  
        chip->ecc.bytes        = 3;  
    } else {      小页  
        chip->ecc.size        = 512;  
        chip->ecc.bytes        = 3;  
        chip->ecc.layout    = &nand\_hw\_eccoob;  
    }

     对于这些关于ECC\_LAYOUT的暂时还是不怎么懂的。。。。。。

}

看看上面用到的nand\_hw\_eccoob它是一个结构体用来管理OOB中的ECC和坏块的(第5篇中有详细的说明)

static struct nand\_ecclayout nand\_hw\_eccoob = {  
    .eccbytes = 3,  
    .eccpos = {0, 1, 2},//ECC在OOB中的位置  
    .oobfree = {{8, 8}}//空闲的OOB字节区域  
};

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

在上面初始化nand\_chip的时候,其中nand\_chip里面有一个这样的(struct nand\_ecc\_ctrl)结构体.里面有许多的成员函数,在初始化的过程中都赋上了值同时nand\_chip中也有许多成员函数赋上了值.。至于它们是怎么实现的,看看返回去看看他们的实现。难度不大。。。。。