# Linux驱动\_MTD

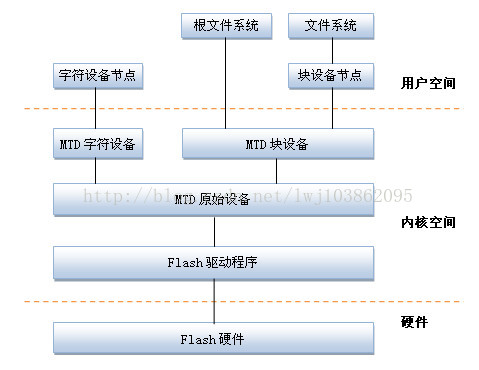
## 1. 基本知识

MTD 设备是象闪存芯片、小型闪存卡、记忆棒等之类的设备，它们在嵌入式设备中的使用正在不断增长。

MTD 驱动程序是在Linux下专门为嵌入式环境开发的新的一类驱动程序。相对于常规块设备驱动程序，使用 MTD 驱动程序的主要优点在于MTD驱动程序是专门为基于闪存的设备所设计的，所以它们通常有更好的支持、更好的管理和基于扇区的擦除和读写操作的更好的接口。

注意：SD/MMC卡、CF（Compact Flash）卡、USB Flash等并不是MTD设备，因为这些设备中已经有一个内置的Flash Translation Layer，这个layer处理erase、wear leveling事情了（这个TL应该是固件中支持的）。所以这些设备直接当做普通的Block Device使用。

MTD层为NOR FLASH和NAND FLASH设备提供统一接口。MTD将文件系统与底层FLASH存储器进行了隔离。



如上图所示，MTD设备通常可分为四层，从上到下依次是：设备节点、MTD设备层、MTD原始设备层、硬件驱动层。

**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Nand Flash芯片的驱动则drivers/mtd/nand/子目录下,Nor Flash芯片驱动位于drivers/mtd/chips/子目录下。

**MTD原始设备层：**用于描述MTD原始设备的[**数据结构**](http://lib.csdn.net/base/datastructure)是mtd\_info，它定义了大量的关于MTD的数据和操作函数。其中**mtdcore.c**:  MTD原始设备接口相关实现，**mtdpart.c** :  MTD分区接口相关实现。

**MTD设备层：**基于MTD原始设备，linux系统可以定义出MTD的**块设备（主设备号31）**和**字符设备（设备号90）**。其中**mtdchar.c** :  MTD字符设备接口相关实现，**mtdblock.c** : MTD块设备接口相关实现。

**设备节点：**通过mknod在/dev子目录下建立**MTD块设备节点（主设备号为31）**和**MTD字符设备节点（主设备号为90）**。通过访问此设备节点即可访问MTD字符设备和块设备 。

## 2. nandflash驱动分析

分析驱动的套路都是差不多的，先找到nandflash对应的platform device和platform driver。

### 2.1 platform device

首先定义了nandflash的platform\_device结构体，指定了操作的nandflash的寄存器地址。

static struct resource s3c\_nand\_resource**[]** **=** **{**

**[**0**]** **=** **{**

**.**start **=** S3C2410\_PA\_NAND**,**

**.**end **=** S3C2410\_PA\_NAND **+** S3C24XX\_SZ\_NAND **-** 1**,**

**.**flags **=** IORESOURCE\_MEM**,**

**}**

**};**

struct platform\_device s3c\_device\_nand **=** **{**

**.**name **=** "s3c2410-nand"**,**

**.**id **=** **-**1**,**

**.**num\_resources **=** ARRAY\_SIZE**(**s3c\_nand\_resource**),**

**.**resource **=** s3c\_nand\_resource**,**

**};**

在s3c2440\_map\_io函数中，将s3c\_device\_nand的名称修改为s3c2440-nand。

s3c\_device\_nand**.**name **=** "s3c2440-nand"**;**

当然，仅有这些数据是完全不够的，还需要设置一些私有数据。

s3c\_device\_nand**.**dev**.**platform\_data **=** **&**smdk\_nand\_info**;**

static struct s3c2410\_platform\_nand smdk\_nand\_info **=** **{**

**.**tacls **=** 20**,**

**.**twrph0 **=** 60**,**

**.**twrph1 **=** 20**,**

**.**nr\_sets **=** ARRAY\_SIZE**(**smdk\_nand\_sets**),**

**.**sets **=** smdk\_nand\_sets**,**

**};**

其中：

TACLS：表示CLT/ALE的建立时间(setup time)。

TWRPH0：表示CLE/ALE的持续时间。

TWRPH1：表示CLE/ALE的维持时间(hold time)。

smdk\_nand\_sets为nandflash分区表。

static struct s3c2410\_nand\_set smdk\_nand\_sets**[]** **=** **{**

**[**0**]** **=** **{**

**.**name **=** "NAND"**,**

**.**nr\_chips **=** 1**,**

**.**nr\_partitions **=** ARRAY\_SIZE**(**smdk\_default\_nand\_part**),**

**.**partitions **=** smdk\_default\_nand\_part**,**

**},**

**};**

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**,**

**}**

**};**

最后调用platform\_device\_register注册到系统中。

### 2.2 platform driver

同样的，会注册s3c2440-nand对应的platform driver。

static struct platform\_driver s3c2440\_nand\_driver **=** **{**

**.**probe **=** s3c2440\_nand\_probe**,**

**.**remove **=** s3c2410\_nand\_remove**,**

**.**suspend **=** s3c24xx\_nand\_suspend**,**

**.**resume **=** s3c24xx\_nand\_resume**,**

**.**driver **=** **{**

**.**name **=** "s3c2440-nand"**,**

**.**owner **=** THIS\_MODULE**,**

**},**

**};**

### 2.3 s3c2440\_nand\_probe

static int s3c2440\_nand\_probe**(**struct platform\_device **\***dev**)**

**{**

**return** s3c24xx\_nand\_probe**(**dev**,** TYPE\_S3C2440**);**

**}**

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

enum s3c\_cpu\_type cpu\_type**)**

**{**

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

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**;**

//分配内存，用于存储s3c2410\_nand\_info

info **=** kmalloc**(sizeof(\***info**),** GFP\_KERNEL**);**

memzero**(**info**,** **sizeof(\***info**));**

//设置platform\_device的私有参数

//可以使用platform\_get\_drvdata得到私有参数

platform\_set\_drvdata**(**pdev**,** info**);**

//初始化自旋锁

spin\_lock\_init**(&**info**->**controller**.**lock**);**

//初始化等待队列

init\_waitqueue\_head**(&**info**->**controller**.**wq**);**

//打开nandflash时钟

info**->**clk **=** clk\_get**(&**pdev**->**dev**,** "nand"**);**

clk\_enable**(**info**->**clk**);**

//获取nandflash操作寄存器，并标记这块空间

res **=** pdev**->**resource**;**

size **=** res**->**end **-** res**->**start **+** 1**;**

info**->**area **=** request\_mem\_region**(**res**->**start**,** size**,** pdev**->**name**);**

//将device结构体赋给info

//device结构体用于描述设备相关的信息设备之间的层次关系，以及设备与总线、驱动的关系。

info**->**device **=** **&**pdev**->**dev**;**

//platform为platform device的platform\_data，也就是smdk\_nand\_info

info**->**platform **=** plat**;**

//将nandflash的寄存器地址ioremap

info**->**regs **=** ioremap**(**res**->**start**,** size**);**

//cpu\_type设置为TYPE\_S3C2440

info**->**cpu\_type **=** cpu\_type**;**

//设置TACLS/TWRPH0/TWRPH1寄存器并使能controller

err **=** s3c2410\_nand\_inithw**(**info**,** pdev**);**

//获取分区信息

sets **=** **(**plat **!=** **NULL)** **?** plat**->**sets **:** **NULL;**

nr\_sets **=** **(**plat **!=** **NULL)** **?** plat**->**nr\_sets **:** 1**;**

//分区的个数

info**->**mtd\_count **=** nr\_sets**;**

//分配分区个数\*s3c2410\_nand\_mtd结构体空间

size **=** nr\_sets **\*** **sizeof(\***info**->**mtds**);**

info**->**mtds **=** kmalloc**(**size**,** GFP\_KERNEL**);**

memzero**(**info**->**mtds**,** size**);**

//初始化每个mtd分区,并将mtd分区添加到系统中

nmtd **=** info**->**mtds**;**

**for** **(**setno **=** 0**;** setno **<** nr\_sets**;** setno**++,** nmtd**++)** **{**

s3c2410\_nand\_init\_chip**(**info**,** nmtd**,** sets**);**

nmtd**->**scan\_res **=** nand\_scan**(&**nmtd**->**mtd**,** **(**sets**)** **?** sets**->**nr\_chips **:** 1**);**

**if** **(**nmtd**->**scan\_res **==** 0**)** **{**

s3c2410\_nand\_add\_partition**(**info**,** nmtd**,** sets**);**

**}**

**if** **(**sets **!=** **NULL)**

sets**++;**

**}**

**}**

#### 2.3.1 s3c2410\_nand\_init\_chip

s3c2410\_nand\_init\_chip用于初始化s3c2410\_nand\_mtd结构体，将nandflash的操作函数和相关寄存器以及分区表信息保存在该结构体中。

//s3c2410\_nand\_info:2410 nandflash相关的信息

//s3c2410\_nand\_mtd:mtd分区结构体

//s3c2410\_nand\_set:分区状态结构体

static void s3c2410\_nand\_init\_chip**(**struct s3c2410\_nand\_info **\***info**,**

struct s3c2410\_nand\_mtd **\***nmtd**,**

struct s3c2410\_nand\_set **\***set**)**

**{**

//nand\_chip存储了nandflash操作的相关函数

struct nand\_chip **\***chip **=** **&**nmtd**->**chip**;**

void \_\_iomem **\***regs **=** info**->**regs**;**

//往IO\_ADDR\_W写数据

chip**->**write\_buf **=** s3c2410\_nand\_write\_buf**;**

//读IO\_ADDR\_R的数据

chip**->**read\_buf **=** s3c2410\_nand\_read\_buf**;**

//操作sel\_reg寄存器的sel\_bit选中寄存器

chip**->**select\_chip **=** s3c2410\_nand\_select\_chip**;**

//nandflash关键操作后的延时时间

chip**->**chip\_delay **=** 50**;**

//设置chip的私有变量为nmtd

chip**->**priv **=** nmtd**;**

chip**->**options **=** 0**;**

chip**->**controller **=** **&**info**->**controller**;**

**switch** **(**info**->**cpu\_type**)** **{**

**case** TYPE\_S3C2440**:**

//设置IO\_ADDR\_W地址,IO\_ADDR\_W用于data数据的写入

chip**->**IO\_ADDR\_W **=** regs **+** S3C2440\_NFDATA**;**

//设置芯片选中操作寄存器和地址

info**->**sel\_reg **=** regs **+** S3C2440\_NFCONT**;**

info**->**sel\_bit **=** S3C2440\_NFCONT\_nFCE**;**

//cmd\_ctrl用于操作data线上传输的是命令还是数据

chip**->**cmd\_ctrl **=** s3c2440\_nand\_hwcontrol**;**

//dev\_ready用于检查操作是否完成

chip**->**dev\_ready **=** s3c2440\_nand\_devready**;**

**break;**

**}**

//IO\_ADDR\_R用于data数据的读取，这里读写是公用寄存器

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

//nmtd保存了nandinfo的信息，nandflash相关的操作函数合集chip和分区表信息

nmtd**->**info **=** info**;**

nmtd**->**mtd**.**priv **=** chip**;**

nmtd**->**mtd**.**owner **=** THIS\_MODULE**;**

nmtd**->**set **=** set**;**

//这里使用的是软件ecc

chip**->**ecc**.**mode **=** NAND\_ECC\_SOFT**;**

**}**

#### 2.3.2 nand\_scan

nand\_scan用于读取nandflash的id，并获取nandflash的相关参数信息以及设置正确的操作函数。

int nand\_scan**(**struct mtd\_info **\***mtd**,** int maxchips**)**

**{**

int ret**;**

/\* Many callers got this wrong, so check for it for a while... \*/

**if** **(!**mtd**->**owner **&&** caller\_is\_module**())** **{**

printk**(**KERN\_CRIT "nand\_scan() called with NULL mtd->owner!\n"**);**

BUG**();**

**}**

ret **=** nand\_scan\_ident**(**mtd**,** maxchips**);**

**if** **(!**ret**)**

ret **=** nand\_scan\_tail**(**mtd**);**

**return** ret**;**

**}**

nand\_scan\_ident代码如下：

int nand\_scan\_ident**(**struct mtd\_info **\***mtd**,** int maxchips**)**

**{**

int i**,** busw**,** nand\_maf\_id**;**

struct nand\_chip **\***chip **=** mtd**->**priv**;**

struct nand\_flash\_dev **\***type**;**

//options=0，data线是8条，所以总线宽度为8

busw **=** chip**->**options **&** NAND\_BUSWIDTH\_16**;**

//设置nandflash默认的一些操作函数，但是并不一定是最优的

nand\_set\_defaults**(**chip**,** busw**);**

//读取id，获取nandflash的关键参数并修改一些操作函数，存储在mtd和chip当中

type **=** nand\_get\_flash\_type**(**mtd**,** chip**,** busw**,** **&**nand\_maf\_id**);**

/\* Check for a chip array \*/

**for** **(**i **=** 1**;** i **<** maxchips**;** i**++)** **{**

chip**->**select\_chip**(**mtd**,** i**);**

/\* Send the command for reading device ID \*/

chip**->**cmdfunc**(**mtd**,** NAND\_CMD\_READID**,** 0x00**,** **-**1**);**

/\* Read manufacturer and device IDs \*/

**if** **(**nand\_maf\_id **!=** chip**->**read\_byte**(**mtd**)** **||**

type**->**id **!=** chip**->**read\_byte**(**mtd**))**

**break;**

**}**

**if** **(**i **>** 1**)**

printk**(**KERN\_INFO "%d NAND chips detected\n"**,** i**);**

/\* Store the number of chips and calc total size for mtd \*/

chip**->**numchips **=** i**;**

mtd**->**size **=** i **\*** chip**->**chipsize**;**

**return** 0**;**

**}**

nand\_get\_flash\_typed代码如下：

static struct nand\_flash\_dev **\***nand\_get\_flash\_type**(**struct mtd\_info **\***mtd**,**

struct nand\_chip **\***chip**,**

int busw**,** int **\***maf\_id**)**

**{**

struct nand\_flash\_dev **\***type **=** **NULL;**

int i**,** dev\_id**,** maf\_idx**;**

//选中芯片

chip**->**select\_chip**(**mtd**,** 0**);**

//发送NAND\_CMD\_READID指令，获取nandflash id号

chip**->**cmdfunc**(**mtd**,** NAND\_CMD\_READID**,** 0x00**,** **-**1**);**

//读取Read manufacturer和device IDs

**\***maf\_id **=** chip**->**read\_byte**(**mtd**);**

dev\_id **=** chip**->**read\_byte**(**mtd**);**

//从nand\_flash\_ids表中找到对应的id

//{"NAND 256MiB 3,3V 8-bit", 0xDA, 0, 256, 0, LP\_OPTIONS},

**for** **(**i **=** 0**;** nand\_flash\_ids**[**i**].**name **!=** **NULL;** i**++)** **{**

**if** **(**dev\_id **==** nand\_flash\_ids**[**i**].**id**)** **{**

type **=** **&**nand\_flash\_ids**[**i**];**

**break;**

**}**

**}**

//256左移20位256\*1024\*1024 256MB

chip**->**chipsize **=** type**->**chipsize **<<** 20**;**

/\* Newer devices have all the information in additional id bytes \*/

//新的设备pagesize信息包含在additional id里面

**if** **(!**type**->**pagesize**)** **{**

int extid**;**

//继续读一个字节

chip**->**cellinfo **=** chip**->**read\_byte**(**mtd**);**

//继续读一个字节

extid **=** chip**->**read\_byte**(**mtd**);**

//计算pagesize

mtd**->**writesize **=** 1024 **<<** **(**extid **&** 0x3**);**

extid **>>=** 2**;**

//计算oob size

mtd**->**oobsize **=** **(**8 **<<** **(**extid **&** 0x01**))** **\*** **(**mtd**->**writesize **>>** 9**);**

extid **>>=** 2**;**

//计算blocksize Blocksize is multiples of 64KiB

mtd**->**erasesize **=** **(**64 **\*** 1024**)** **<<** **(**extid **&** 0x03**);**

extid **>>=** 2**;**

//计算bus宽度

busw **=** **(**extid **&** 0x01**)** **?** NAND\_BUSWIDTH\_16 **:** 0**;**

**}**

//制造商识别

**for** **(**maf\_idx **=** 0**;** nand\_manuf\_ids**[**maf\_idx**].**id **!=** 0x0**;** maf\_idx**++)** **{**

**if** **(**nand\_manuf\_ids**[**maf\_idx**].**id **==** **\***maf\_id**)**

**break;**

**}**

//给定一个address，计算其所在的page，因为使用除法费时间，所以需要右移

chip**->**page\_shift **=** ffs**(**mtd**->**writesize**)** **-** 1**;**

//整个chipsize所拥有的page数-1

chip**->**pagemask **=** **(**chip**->**chipsize **>>** chip**->**page\_shift**)** **-** 1**;**

//擦除是按block位单位，这里用于计算block number

chip**->**bbt\_erase\_shift **=** chip**->**phys\_erase\_shift **=**

ffs**(**mtd**->**erasesize**)** **-** 1**;**

//这里用于计算当前处于的page number

chip**->**chip\_shift **=** ffs**(**chip**->**chipsize**)** **-** 1**;**

//oob中存储坏块标志的位置

chip**->**badblockpos **=** mtd**->**writesize **>** 512 **?**

NAND\_LARGE\_BADBLOCK\_POS **:** NAND\_SMALL\_BADBLOCK\_POS**;**

/\* Get chip options, preserve non chip based options \*/

chip**->**options **&=** **~**NAND\_CHIPOPTIONS\_MSK**;**

chip**->**options **|=** type**->**options **&** NAND\_CHIPOPTIONS\_MSK**;**

/\*

\* Set chip as a default. Board drivers can override it, if necessary

\*/

chip**->**options **|=** NAND\_NO\_AUTOINCR**;**

/\* Check if chip is a not a samsung device. Do not clear the

\* options for chips which are not having an extended id.

\*/

**if** **(\***maf\_id **!=** NAND\_MFR\_SAMSUNG **&&** **!**type**->**pagesize**)**

chip**->**options **&=** **~**NAND\_SAMSUNG\_LP\_OPTIONS**;**

/\* Check for AND chips with 4 page planes \*/

**if** **(**chip**->**options **&** NAND\_4PAGE\_ARRAY**)**

chip**->**erase\_cmd **=** multi\_erase\_cmd**;**

**else**

chip**->**erase\_cmd **=** single\_erase\_cmd**;**

//nandflash命令操作函数 512bytes和2048bytes page的操作函数是不一样的

**if** **(**mtd**->**writesize **>** 512 **&&** chip**->**cmdfunc **==** nand\_command**)**

chip**->**cmdfunc **=** nand\_command\_lp**;**

**return** type**;**

**}**

nand\_scan\_tail会进一步设置nandflash和mtd的操作函数。

//nandflash相关参数的第二步设置

int nand\_scan\_tail**(**struct mtd\_info **\***mtd**)**

**{**

int i**;**

struct nand\_chip **\***chip **=** mtd**->**priv**;**

//申请一块buffer用于存储data+oob等数据

**if** **(!(**chip**->**options **&** NAND\_OWN\_BUFFERS**))**

chip**->**buffers **=** kmalloc**(sizeof(\***chip**->**buffers**),** GFP\_KERNEL**);**

//oob的存储区域在databuf后的2048个字节处

chip**->**oob\_poi **=** chip**->**buffers**->**databuf **+** mtd**->**writesize**;**

//oob结构体，记录了oob中eccbytes的长度和位置

**if** **(!**chip**->**ecc**.**layout**)** **{**

**switch** **(**mtd**->**oobsize**)** **{**

**case** 64**:**

chip**->**ecc**.**layout **=** **&**nand\_oob\_64**;**

**break;**

**}**

**}**

//nandflash的页写函数,代码分析可以见uboot

//nand\_write\_page分raw模式和ecc模式

**if** **(!**chip**->**write\_page**)**

chip**->**write\_page **=** nand\_write\_page**;**

//raw模式为直接读写data和oob

**if** **(!**chip**->**ecc**.**read\_page\_raw**)**

chip**->**ecc**.**read\_page\_raw **=** nand\_read\_page\_raw**;**

**if** **(!**chip**->**ecc**.**write\_page\_raw**)**

chip**->**ecc**.**write\_page\_raw **=** nand\_write\_page\_raw**;**

**switch** **(**chip**->**ecc**.**mode**)** **{**

//soft ecc模式下的读写函数，

//写的过程中，会计算ecc，最后面通过raw方式写

//读的过程中，先raw方式读出数据，并进行ecc纠错

**case** NAND\_ECC\_SOFT**:**

chip**->**ecc**.**calculate **=** nand\_calculate\_ecc**;**

chip**->**ecc**.**correct **=** nand\_correct\_data**;**

chip**->**ecc**.**read\_page **=** nand\_read\_page\_swecc**;**

chip**->**ecc**.**write\_page **=** nand\_write\_page\_swecc**;**

chip**->**ecc**.**read\_oob **=** nand\_read\_oob\_std**;**

chip**->**ecc**.**write\_oob **=** nand\_write\_oob\_std**;**

chip**->**ecc**.**size **=** 256**;**

chip**->**ecc**.**bytes **=** 3**;**

**break;**

**}**

//oob区域能被fs使用的个数oobavail=64-(24+1)=39

//除了ECC校验码之外可用的OOB字节数

chip**->**ecc**.**layout**->**oobavail **=** 0**;**

**for** **(**i **=** 0**;** chip**->**ecc**.**layout**->**oobfree**[**i**].**length**;** i**++)**

chip**->**ecc**.**layout**->**oobavail **+=**

chip**->**ecc**.**layout**->**oobfree**[**i**].**length**;**

mtd**->**oobavail **=** chip**->**ecc**.**layout**->**oobavail**;**

//ecc计算所需要的step 256个字节计算一次

chip**->**ecc**.**steps **=** mtd**->**writesize **/** chip**->**ecc**.**size**;**

**if(**chip**->**ecc**.**steps **\*** chip**->**ecc**.**size **!=** mtd**->**writesize**)** **{**

printk**(**KERN\_WARNING "Invalid ecc parameters\n"**);**

BUG**();**

**}**

chip**->**ecc**.**total **=** chip**->**ecc**.**steps **\*** chip**->**ecc**.**bytes**;**

/\*

\* Allow subpage writes up to ecc.steps. Not possible for MLC

\* FLASH.

\*/

**if** **(!(**chip**->**options **&** NAND\_NO\_SUBPAGE\_WRITE**)** **&&**

**!(**chip**->**cellinfo **&** NAND\_CI\_CELLTYPE\_MSK**))** **{**

**switch(**chip**->**ecc**.**steps**)** **{**

**case** 2**:**

mtd**->**subpage\_sft **=** 1**;**

**break;**

**case** 4**:**

**case** 8**:**

mtd**->**subpage\_sft **=** 2**;**

**break;**

**}**

**}**

chip**->**subpagesize **=** mtd**->**writesize **>>** mtd**->**subpage\_sft**;**

/\* Initialize state \*/

chip**->**state **=** FL\_READY**;**

/\* De-select the device \*/

chip**->**select\_chip**(**mtd**,** **-**1**);**

/\* Invalidate the pagebuffer reference \*/

chip**->**pagebuf **=** **-**1**;**

//设置mtd操作函数，最终会调用nandchip中的操作函数

mtd**->**type **=** MTD\_NANDFLASH**;**

mtd**->**flags **=** MTD\_CAP\_NANDFLASH**;**

mtd**->**erase **=** nand\_erase**;**

mtd**->**point **=** **NULL;**

mtd**->**unpoint **=** **NULL;**

mtd**->**read **=** nand\_read**;**

mtd**->**write **=** nand\_write**;**

mtd**->**read\_oob **=** nand\_read\_oob**;**

mtd**->**write\_oob **=** nand\_write\_oob**;**

mtd**->**sync **=** nand\_sync**;**

mtd**->**lock **=** **NULL;**

mtd**->**unlock **=** **NULL;**

mtd**->**suspend **=** nand\_suspend**;**

mtd**->**resume **=** nand\_resume**;**

mtd**->**block\_isbad **=** nand\_block\_isbad**;**

mtd**->**block\_markbad **=** nand\_block\_markbad**;**

/\* propagate ecc.layout to mtd\_info \*/

mtd**->**ecclayout **=** chip**->**ecc**.**layout**;**

/\* Check, if we should skip the bad block table scan \*/

**if** **(**chip**->**options **&** NAND\_SKIP\_BBTSCAN**)**

**return** 0**;**

//坏块表的建立

//BBT可以存放在内存中，也可以存放在flash中，

//nand\_scan\_bbt会跟据bbt\_td的值判断BBT是存放在内存中还是在flash中，

//bbt\_td为NULL，那么BBT就会存放在内存中。

**return** chip**->**scan\_bbt**(**mtd**);**

**}**

#### 2.3.3 s3c2410\_nand\_add\_partition

如果MTD设备只有一个分区，那么使用下面两个函数注册和注销MTD设备。

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

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

如果MTD设备存在其他分区，那么使用下面两个函数注册和注销MTD设备。

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

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

s3c2410\_nand\_add\_partition函数定义如下：

static int s3c2410\_nand\_add\_partition**(**struct s3c2410\_nand\_info **\***info**,**

struct s3c2410\_nand\_mtd **\***mtd**,**

struct s3c2410\_nand\_set **\***set**)**

**{**

**if** **(**set **==** **NULL)**

**return** add\_mtd\_device**(&**mtd**->**mtd**);**

**if** **(**set**->**nr\_partitions **>** 0 **&&** set**->**partitions **!=** **NULL)** **{**

**return** add\_mtd\_partitions**(&**mtd**->**mtd**,** set**->**partitions**,** set**->**nr\_partitions**);**

**}**

**return** add\_mtd\_device**(&**mtd**->**mtd**);**

**}**

重要的数据结构：   
mtd\_info表示mtd原始设备, 所有mtd\_info结构体被存放在mtd\_info数组mtd\_table中。  
mtd\_part  表示MTD分区，其中包含了mtd\_info，每一个分区都是被看成一个MTD 原始设备。

struct mtd\_info **\***mtd\_table**[**MAX\_MTD\_DEVICES**];**

下面来分析一下add\_mtd\_partitions函数：  
此函数将创建nbparts个mtd\_part，其中每个mtd\_part包含了一个mtd\_info，mtd\_info的数据内容来自master和parts。程序将mtd\_info加入mtd\_table，并将mtd\_part中的list\_head加入list。

当所有分区建立完成后，一个分区对应一个mtd\_part，每个mtd\_part内部有一个实体mtd\_info。但struct mtd\_info \*master不会被建立分区，它仅仅是在FALSH驱动中的一个用来作参考用的结构。

这里贴出了部分关键代码。

slave**->**mtd**.**type **=** master**->**type**;**  //复制了主分区的信息

slave**->**mtd**.**size **=** parts**[**i**].**size**;**   //改了size

slave**->**mtd**.**oobblock **=** master**->**oobblock**;** //复制了主分区的信息

slave**->**mtd**.**name **=** parts**[**i**].**name**;** //改了名字

slave**->**mtd**.**bank\_size **=** master**->**bank\_size**;** //复制了主分区的信息

slave**->**mtd**.**read **=** part\_read**;**  //这个是自己的，下面分析看看

**if(**parts**[**i**].**mtdp**)**

**{**

/\* store the object pointer (caller may or may not register it \*/

**\***parts**[**i**].**mtdp **=** **&**slave**->**mtd**;**

slave**->**registered **=** 0**;**//你也可以不加进mtd\_table中

**}**

**else**

**{**

   /\* register our partition \*/

   add\_mtd\_device**(&**slave**->**mtd**);**//最后是这个，加进mtd\_table中的是而不是那个包含它的结构mtd\_part

   slave**->**registered **=** 1**;**

**}**

再来看看part\_read函数，其实还是调用主分区的读函数，就是加上了偏移量而已。

static int part\_read **(**struct mtd\_info **\***mtd**,** loff\_t from**,** size\_t len**,**

                     size\_t **\***retlen**,** u\_char **\***buf**)**

**{**

struct mtd\_part **\***part **=** PART**(**mtd**);**

    if **(**from **>=** mtd**->**size**)**

     len **=** 0**;**

    else **if** **(**from **+** len **>** mtd**->**size**)**

        len **=** mtd**->**size **-** from**;**

    if **(**part**->**master**->**read\_ecc **==** **NULL)**

        return part**->**master**->**read **(**part**->**master**,** from **+** part**->**offset**,**len**,** retlen**,** buf**);**

    else

        return part**->**master**->**read\_ecc **(**part**->**master**,** from **+** part**->**offset**,**len**,** retlen**,** buf**,** **NULL,** **&**mtd**->**oobinfo**);**

**}**

add\_mtd\_partitions 的最后调用add\_mtd\_device(&slave->mtd);把每个分区加进了mtd\_table，接下来分析一下add\_mtd\_device。

add\_mtd\_device将mtd\_info添加到mtd\_table中，并生成对应的mtd字符设备和块设备。如何生成字符设备和块设备，请看后面的分析。

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

**{**

int i**;**

BUG\_ON**(**mtd**->**writesize **==** 0**);**

mutex\_lock**(&**mtd\_table\_mutex**);**

**for** **(**i**=**0**;** i **<** MAX\_MTD\_DEVICES**;** i**++)**

**if** **(!**mtd\_table**[**i**])** **{**

struct list\_head **\*this;**

//将mtd添加到mtd\_table中

mtd\_table**[**i**]** **=** mtd**;**

mtd**->**index **=** i**;**

mtd**->**usecount **=** 0**;**

DEBUG**(**0**,** "mtd: Giving out device %d to %s\n"**,**i**,** mtd**->**name**);**

//创建mtd字符设备和块设备

list\_for\_each**(this,** **&**mtd\_notifiers**)** **{**

struct mtd\_notifier **\*not** **=** list\_entry**(this,** struct mtd\_notifier**,** list**);**

**not->**add**(**mtd**);**

**}**

mutex\_unlock**(&**mtd\_table\_mutex**);**

\_\_module\_get**(**THIS\_MODULE**);**

**return** 0**;**

**}**

mutex\_unlock**(&**mtd\_table\_mutex**);**

**return** 1**;**

**}**

### 2.4 总结

NAND芯片级初始化 ，主要有以下几个步骤:   
(1)分配nand\_chip内存，根据目标板及NAND控制器初始化nand\_chip中成员函数(若未初始化则使用nand\_base.c中的默认函数)，将mtd\_info中的priv指向nand\_chip(或板相关私有结构)，设置ecc模式及处理函数 。  
(2)以mtd\_info为参数调用nand\_scan()探测NAND FLash。   
(3)nand\_scan()会读取nand芯片ID，并根据mtd->priv即nand\_chip中成员初始化mtd\_info。  
(4)若有分区，则以mtd\_info和mtd\_partition为参数调用add\_mtd\_partitions()添加分区信息。

MTD对NAND芯片的读写，主要分三部分:   
(1)struct mtd\_info中的读写函数，如read，write\_oob等，这是MTD原始设备层与FLASH硬件层之间的接口；   
(2)struct nand\_ecc\_ctrl中的读写函数，如read\_page\_raw，write\_page等，主要用来做一些与ecc有关的操作；   
(3)struct nand\_chip中的读写函数，如read\_buf，cmdfunc等，与具体的NAND controller相关，就是这部分函数与硬件交互，通常需要我们自己来实现。   
**tip**: nand\_chip中的读写函数虽然与具体的NAND controller相关，但是MTD也为我们提供了默认的读写函数，如果NAND controller比较通用(使用PIO模式)，那么对NAND芯片的读写与MTD提供的这些函数一致，就不必自己实现这些函数。

## 3.MTD子系统框架



### 3.1 MTD设备层

设备层是实现了文件系统与Flash之间的桥梁，其基于MTD原始层提供了两种上层访问Flash的方式：MTD的字符设备和块设备，字符设备通过向内核注册字符设备的file\_operations结构实现了对MTD设备的读写和控制，提供了对闪存的原始字符访问，关联的设备是/dev/mtd\*，而MTD块设备则是定义了一个描述MTD块设备mtdblock\_tr的结构，关联的设备是/dev/mtdblock\*，下面主要看看其实现的原理。

#### 3.1.1 MTD字符设备

mtd字符设备的初始化很简单，创建字符设备，创建/sys/mtd，注册notifier。

#define MTD\_CHAR\_MAJOR 90

static const struct file\_operations mtd\_fops **=** **{**

**.**owner **=** THIS\_MODULE**,**

**.**llseek **=** mtd\_lseek**,**

**.**read **=** mtd\_read**,**

**.**write **=** mtd\_write**,**

**.**ioctl **=** mtd\_ioctl**,**

**.**open **=** mtd\_open**,**

**.**release **=** mtd\_close**,**

**};**

static int \_\_init init\_mtdchar**(**void**)**

**{**

//创建字符设备

register\_chrdev**(**MTD\_CHAR\_MAJOR**,** "mtd"**,** **&**mtd\_fops**);**

//创建/sys/mtd

mtd\_class **=** class\_create**(**THIS\_MODULE**,** "mtd"**);**

//为每个设备注册注册notifier

register\_mtd\_user**(&**notifier**);**

**return** 0**;**

**}**

这里来分析下register\_mtd\_user函数的实现。

static struct mtd\_notifier notifier **=** **{**

**.**add **=** mtd\_notify\_add**,**

**.**remove **=** mtd\_notify\_remove**,**

**};**

void register\_mtd\_user **(**struct mtd\_notifier **\*new)**

**{**

int i**;**

mutex\_lock**(&**mtd\_table\_mutex**);**

//往mtd\_notifiers链表添加notifier

list\_add**(&new->**list**,** **&**mtd\_notifiers**);**

\_\_module\_get**(**THIS\_MODULE**);**

//遍历mtd\_table，mtd\_table中保存了flash分区信息

//假如mtd\_table[i]存在，调用notifier中的mtd\_notify\_add函数

**for** **(**i**=**0**;** i**<** MAX\_MTD\_DEVICES**;** i**++)**

**if** **(**mtd\_table**[**i**])**

**new->**add**(**mtd\_table**[**i**]);**

mutex\_unlock**(&**mtd\_table\_mutex**);**

**}**

register\_mtd\_user往mtd\_notifiers添加了notifier，notifier用于flash分区字符设备的创建。其中/dev/mtdxro只能用来读。

static void mtd\_notify\_add**(**struct mtd\_info**\*** mtd**)**

**{**

**if** **(!**mtd**)**

**return;**

//创建设备/dev/mtdx

class\_device\_create**(**mtd\_class**,** **NULL,** MKDEV**(**MTD\_CHAR\_MAJOR**,** mtd**->**index**\***2**),NULL,** "mtd%d"**,** mtd**->**index**);**

//创建设备/dev/mtdxro

class\_device\_create**(**mtd\_class**,** **NULL,**

MKDEV**(**MTD\_CHAR\_MAJOR**,** mtd**->**index**\***2**+**1**),**

**NULL,** "mtd%dro"**,** mtd**->**index**);**

**}**

下面来分析这些mtd字符设备的操作函数。

##### 3.1.1.1 mtd\_open

mtd\_open打开对应的mtd字符设备，根据子设备号获取对应mtd\_info结构体。

static int mtd\_open**(**struct inode **\***inode**,** struct file **\***file**)**

**{**

int minor **=** iminor**(**inode**);**

int devnum **=** minor **>>** 1**;**

struct mtd\_info **\***mtd**;**

struct mtd\_file\_info **\***mfi**;**

// /dev/mtdxro设备不能被写打开

**if** **((**file**->**f\_mode **&** 2**)** **&&** **(**minor **&** 1**))**

**return** **-**EACCES**;**

// 获取devnum对应的mtd\_info结构体

mtd **=** get\_mtd\_device**(NULL,** devnum**);**

/\* You can't open it RW if it's not a writeable device \*/

**if** **((**file**->**f\_mode **&** 2**)** **&&** **!(**mtd**->**flags **&** MTD\_WRITEABLE**))** **{**

put\_mtd\_device**(**mtd**);**

**return** **-**EACCES**;**

**}**

//分配一个mtd\_file\_info结构体

mfi **=** kzalloc**(sizeof(\***mfi**),** GFP\_KERNEL**);**

mfi**->**mtd **=** mtd**;**

//设置file的私有数据

file**->**private\_data **=** mfi**;**

**return** 0**;**

**}** /\* mtd\_open \*/

##### 3.1.1.2 mtd\_read

static ssize\_t mtd\_read**(**struct file **\***file**,** char \_\_user **\***buf**,** size\_t count**,**loff\_t **\***ppos**)**

**{**

struct mtd\_file\_info **\***mfi **=** file**->**private\_data**;**

struct mtd\_info **\***mtd **=** mfi**->**mtd**;**

size\_t retlen**=**0**;**

size\_t total\_retlen**=**0**;**

int ret**=**0**;**

int len**;**

char **\***kbuf**;**

//检查读取的范围是否超出

**if** **(\***ppos **+** count **>** mtd**->**size**)**

count **=** mtd**->**size **-** **\***ppos**;**

//分配缓存

**if** **(**count **>** MAX\_KMALLOC\_SIZE**)**

kbuf**=**kmalloc**(**MAX\_KMALLOC\_SIZE**,** GFP\_KERNEL**);**

**else**

kbuf**=**kmalloc**(**count**,** GFP\_KERNEL**);**

**while** **(**count**)** **{**

**if** **(**count **>** MAX\_KMALLOC\_SIZE**)**

len **=** MAX\_KMALLOC\_SIZE**;**

**else**

len **=** count**;**

//mfi->mode可以由ioctl设置

**switch** **(**mfi**->**mode**)** **{**

**case** MTD\_MODE\_OTP\_FACTORY**:**

ret **=** mtd**->**read\_fact\_prot\_reg**(**mtd**,** **\***ppos**,** len**,** **&**retlen**,** kbuf**);**

**break;**

**case** MTD\_MODE\_OTP\_USER**:**

ret **=** mtd**->**read\_user\_prot\_reg**(**mtd**,** **\***ppos**,** len**,** **&**retlen**,** kbuf**);**

**break;**

**case** MTD\_MODE\_RAW**:**

**{**

//读取data和oob

struct mtd\_oob\_ops ops**;**

ops**.**mode **=** MTD\_OOB\_RAW**;**

ops**.**datbuf **=** kbuf**;**

ops**.**oobbuf **=** **NULL;**

ops**.**len **=** len**;**

ret **=** mtd**->**read\_oob**(**mtd**,** **\***ppos**,** **&**ops**);**

retlen **=** ops**.**retlen**;**

**break;**

**}**

**default:**

//读取数据

ret **=** mtd**->**read**(**mtd**,** **\***ppos**,** len**,** **&**retlen**,** kbuf**);**

**}**

**if** **(!**ret **||** **(**ret **==** **-**EUCLEAN**)** **||** **(**ret **==** **-**EBADMSG**))** **{**

**\***ppos **+=** retlen**;**

//返回给用户层

**if** **(**copy\_to\_user**(**buf**,** kbuf**,** retlen**))** **{**

kfree**(**kbuf**);**

**return** **-**EFAULT**;**

**}**

**else**

total\_retlen **+=** retlen**;**

count **-=** retlen**;**

buf **+=** retlen**;**

**if** **(**retlen **==** 0**)**

count **=** 0**;**

**}**

**else** **{**

kfree**(**kbuf**);**

**return** ret**;**

**}**

**}**

kfree**(**kbuf**);**

**return** total\_retlen**;**

**}** /\* mtd\_read \*/

mtd\_read最终调用的是mtd->read函数，也就是part\_read。

##### 3.1.1.3 mtd\_write

mtd\_read和mtd\_write差不多，这里就不分析了。

##### 3.1.1.4 mtd\_ioctl

mtd\_ioctl提供了一些指令，这里挑一些出来分析。

获取分区info，包含mtd分区类型，擦除大小，写入大小等。

**case** MEMGETINFO**:**

info**.**type **=** mtd**->**type**;**

info**.**flags **=** mtd**->**flags**;**

info**.**size **=** mtd**->**size**;**

info**.**erasesize **=** mtd**->**erasesize**;**

info**.**writesize **=** mtd**->**writesize**;**

info**.**oobsize **=** mtd**->**oobsize**;**

/\* The below fields are obsolete \*/

info**.**ecctype **=** **-**1**;**

info**.**eccsize **=** 0**;**

**if** **(**copy\_to\_user**(**argp**,** **&**info**,** **sizeof(**struct mtd\_info\_user**)))**

**return** **-**EFAULT**;**

**break;**

nandwrite -o -s 0x20000 /dev/mtd4 rootfs.yaffs2

-p file不需要对齐

//增加新的分区 mtd\_utils 分区升级yaffs2 jffs2 nandwrite写分区并挂载