# Linux驱动\_MTD

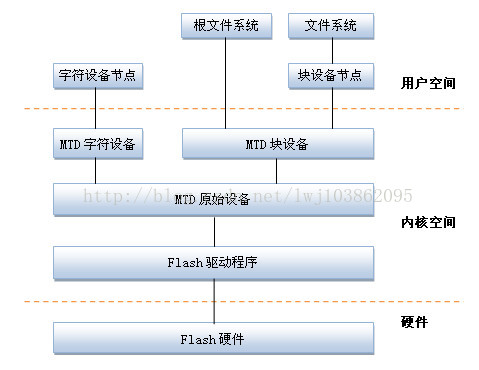
## 1. 基本知识

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。

(1)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;**

(2)擦除操作

**case** MEMERASE**:**

**{**

struct erase\_info **\***erase**;**

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

**return** **-**EPERM**;**

erase**=**kzalloc**(sizeof(**struct erase\_info**),**GFP\_KERNEL**);**

**if** **(!**erase**)**

ret **=** **-**ENOMEM**;**

**else** **{**

wait\_queue\_head\_t waitq**;**

DECLARE\_WAITQUEUE**(**wait**,** current**);**

init\_waitqueue\_head**(&**waitq**);**

**if** **(**copy\_from\_user**(&**erase**->**addr**,** argp**,**

**sizeof(**struct erase\_info\_user**)))** **{**

kfree**(**erase**);**

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

**}**

erase**->**mtd **=** mtd**;**

erase**->**callback **=** mtdchar\_erase\_callback**;**

erase**->**priv **=** **(**unsigned long**)&**waitq**;**

ret **=** mtd**->**erase**(**mtd**,** erase**);**

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

set\_current\_state**(**TASK\_UNINTERRUPTIBLE**);**

add\_wait\_queue**(&**waitq**,** **&**wait**);**

**if** **(**erase**->**state **!=** MTD\_ERASE\_DONE **&&**

erase**->**state **!=** MTD\_ERASE\_FAILED**)**

schedule**();**

remove\_wait\_queue**(&**waitq**,** **&**wait**);**

set\_current\_state**(**TASK\_RUNNING**);**

ret **=** **(**erase**->**state **==** MTD\_ERASE\_FAILED**)?-**EIO**:**0**;**

**}**

kfree**(**erase**);**

**}**

**break;**

**}**

(3)读写oob操作

**case** MEMWRITEOOB**:**

**{**

struct mtd\_oob\_buf buf**;**

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

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

**return** **-**EPERM**;**

**if** **(**copy\_from\_user**(&**buf**,** argp**,** **sizeof(**struct mtd\_oob\_buf**)))**

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

**if** **(**buf**.**length **>** 4096**)**

**return** **-**EINVAL**;**

**if** **(!**mtd**->**write\_oob**)**

ret **=** **-**EOPNOTSUPP**;**

**else**

ret **=** access\_ok**(**VERIFY\_READ**,** buf**.**ptr**,**

buf**.**length**)** **?** 0 **:** EFAULT**;**

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

**return** ret**;**

ops**.**ooblen **=** buf**.**length**;**

ops**.**ooboffs **=** buf**.**start **&** **(**mtd**->**oobsize **-** 1**);**

ops**.**datbuf **=** **NULL;**

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

**if** **(**ops**.**ooboffs **&&** ops**.**ooblen **>** **(**mtd**->**oobsize **-** ops**.**ooboffs**))**

**return** **-**EINVAL**;**

ops**.**oobbuf **=** kmalloc**(**buf**.**length**,** GFP\_KERNEL**);**

**if** **(!**ops**.**oobbuf**)**

**return** **-**ENOMEM**;**

**if** **(**copy\_from\_user**(**ops**.**oobbuf**,** buf**.**ptr**,** buf**.**length**))** **{**

kfree**(**ops**.**oobbuf**);**

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

**}**

buf**.**start **&=** **~(**mtd**->**oobsize **-** 1**);**

ret **=** mtd**->**write\_oob**(**mtd**,** buf**.**start**,** **&**ops**);**

**if** **(**copy\_to\_user**(**argp **+** **sizeof(**uint32\_t**),** **&**ops**.**oobretlen**,**

**sizeof(**uint32\_t**)))**

ret **=** **-**EFAULT**;**

kfree**(**ops**.**oobbuf**);**

**break;**

**}**

**case** MEMREADOOB**:**

**{**

struct mtd\_oob\_buf buf**;**

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

**if** **(**copy\_from\_user**(&**buf**,** argp**,** **sizeof(**struct mtd\_oob\_buf**)))**

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

**if** **(**buf**.**length **>** 4096**)**

**return** **-**EINVAL**;**

**if** **(!**mtd**->**read\_oob**)**

ret **=** **-**EOPNOTSUPP**;**

**else**

ret **=** access\_ok**(**VERIFY\_WRITE**,** buf**.**ptr**,**

buf**.**length**)** **?** 0 **:** **-**EFAULT**;**

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

**return** ret**;**

ops**.**ooblen **=** buf**.**length**;**

ops**.**ooboffs **=** buf**.**start **&** **(**mtd**->**oobsize **-** 1**);**

ops**.**datbuf **=** **NULL;**

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

**if** **(**ops**.**ooboffs **&&** ops**.**ooblen **>** **(**mtd**->**oobsize **-** ops**.**ooboffs**))**

**return** **-**EINVAL**;**

ops**.**oobbuf **=** kmalloc**(**buf**.**length**,** GFP\_KERNEL**);**

**if** **(!**ops**.**oobbuf**)**

**return** **-**ENOMEM**;**

buf**.**start **&=** **~(**mtd**->**oobsize **-** 1**);**

ret **=** mtd**->**read\_oob**(**mtd**,** buf**.**start**,** **&**ops**);**

**if** **(**put\_user**(**ops**.**oobretlen**,** **(**uint32\_t \_\_user **\*)**argp**))**

ret **=** **-**EFAULT**;**

**else** **if** **(**ops**.**oobretlen **&&** copy\_to\_user**(**buf**.**ptr**,** ops**.**oobbuf**,**

ops**.**oobretlen**))**

ret **=** **-**EFAULT**;**

kfree**(**ops**.**oobbuf**);**

**break;**

**}**

#### 3.1.2 MTD块设备

static struct mtd\_blktrans\_ops mtdblock\_tr **=** **{**

**.**name **=** "mtdblock"**,**

**.**major **=** 31**,**

**.**part\_bits **=** 0**,**

**.**blksize **=** 512**,**

**.**open **=** mtdblock\_open**,**

**.**flush **=** mtdblock\_flush**,**

**.**release **=** mtdblock\_release**,**

**.**readsect **=** mtdblock\_readsect**,**

**.**writesect **=** mtdblock\_writesect**,**

**.**add\_mtd **=** mtdblock\_add\_mtd**,**

**.**remove\_dev **=** mtdblock\_remove\_dev**,**

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

**};**

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

**{**

**return** register\_mtd\_blktrans**(&**mtdblock\_tr**);**

**}**

int register\_mtd\_blktrans**(**struct mtd\_blktrans\_ops **\***tr**)**

**{**

int ret**,** i**;**

//为每个设备注册注册notifier,下面会具体分析该函数

**if** **(!**blktrans\_notifier**.**list**.**next**)**

register\_mtd\_user**(&**blktrans\_notifier**);**

//分配mtd\_blkcore\_priv对象空间

tr**->**blkcore\_priv **=** kzalloc**(sizeof(\***tr**->**blkcore\_priv**),** GFP\_KERNEL**);**

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

//注册块设备，主设备号31，名称mtdblock

//它会在/proc/devices中被显示

ret **=** register\_blkdev**(**tr**->**major**,** tr**->**name**);**

//初始化访问请求队列的自旋锁

spin\_lock\_init**(&**tr**->**blkcore\_priv**->**queue\_lock**);**

//初始化请求队列,分配/设置队列: 提供读写能力,mtd\_blktrans\_request为读写请求

tr**->**blkcore\_priv**->**rq **=** blk\_init\_queue**(**mtd\_blktrans\_request**,** **&**tr**->**blkcore\_priv**->**queue\_lock**);**

tr**->**blkcore\_priv**->**rq**->**queuedata **=** tr**;**

//设置MTD块设备硬盘扇区的大小

blk\_queue\_hardsect\_size**(**tr**->**blkcore\_priv**->**rq**,** tr**->**blksize**);**

tr**->**blkshift **=** ffs**(**tr**->**blksize**)** **-** 1**;**

//初始化处理请求队列的的线程,下面会分析该函数

tr**->**blkcore\_priv**->**thread **=** kthread\_run**(**mtd\_blktrans\_thread**,** tr**,**

"%sd"**,** tr**->**name**);**

//初始化mtd设备链表

INIT\_LIST\_HEAD**(&**tr**->**devs**);**

//将tr添加到blktrans\_majors链表中

list\_add**(&**tr**->**list**,** **&**blktrans\_majors**);**

//遍历mtd\_table，并调用mtdblock\_add\_mtd，下面会分析函数

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

**if** **(**mtd\_table**[**i**]** **&&** mtd\_table**[**i**]->**type **!=** MTD\_ABSENT**)**

tr**->**add\_mtd**(**tr**,** mtd\_table**[**i**]);**

**}**

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

**return** 0**;**

**}**

##### 3.1.2.1 /dev/mtdblockx块设备生成相关

register\_mtd\_blktrans 会调用register\_mtd\_user(&blktrans\_notifier)，该函数的作用为：

(1)将blktrans\_notifier加入到mtd\_notifiers。

(2)遍历mtd\_table，对存在的分区调用blktrans\_notify\_add。

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

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

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

**};**

继续分析blktrans\_notify\_add函数。

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

**{**

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

**if** **(**mtd**->**type **==** MTD\_ABSENT**)**

**return;**

//遍历blktrans\_majors链表，这里mtdblock\_tr

list\_for\_each**(this,** **&**blktrans\_majors**)** **{**

struct mtd\_blktrans\_ops **\***tr **=** list\_entry**(this,** struct mtd\_blktrans\_ops**,** list**);**

//调用mtdblock\_tr中的add\_mtd函数mtdblock\_add\_mtd

tr**->**add\_mtd**(**tr**,** mtd**);**

**}**

**}**

最终对每个分区调用的是mtdblock\_add\_mtd函数。

static void mtdblock\_add\_mtd**(**struct mtd\_blktrans\_ops **\***tr**,** struct mtd\_info **\***mtd**)**

**{**

//分配mtd\_blktrans\_dev结构体

struct mtd\_blktrans\_dev **\***dev **=** kzalloc**(sizeof(\***dev**),** GFP\_KERNEL**);**

//mtd\_info赋给dev->mtd

dev**->**mtd **=** mtd**;**

dev**->**devnum **=** mtd**->**index**;**

//除以512，块设备按512个字节操作

dev**->**size **=** mtd**->**size **>>** 9**;**

//将mtdblock\_tr赋给dev->tr

dev**->**tr **=** tr**;**

**if** **(!(**mtd**->**flags **&** MTD\_WRITEABLE**))**

dev**->**readonly **=** 1**;**

//建立对应的块设备/dev/mtdblock

add\_mtd\_blktrans\_dev**(**dev**);**

**}**

mtdblock\_add\_mtd函数就是简单的分配了一个mtd\_blktrans\_dev结构体，并做了相应的初始化。

重头戏来了add\_mtd\_blktrans\_dev会真正的添加块设备，并将mtd\_blktrans\_dev结构体添加到tr->devs链表中。

int add\_mtd\_blktrans\_dev**(**struct mtd\_blktrans\_dev **\*new)**

**{**

struct mtd\_blktrans\_ops **\***tr **=** **new->**tr**;**

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

int last\_devnum **=** **-**1**;**

struct gendisk **\***gd**;**

**if** **(**mutex\_trylock**(&**mtd\_table\_mutex**))** **{**

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

BUG**();**

**}**

list\_for\_each**(this,** **&**tr**->**devs**)** **{**

struct mtd\_blktrans\_dev **\***d **=** list\_entry**(this,** struct mtd\_blktrans\_dev**,** list**);**

**if** **(new->**devnum **==** **-**1**)** **{**

/\* Use first free number \*/

**if** **(**d**->**devnum **!=** last\_devnum**+**1**)** **{**

/\* Found a free devnum. Plug it in here \*/

**new->**devnum **=** last\_devnum**+**1**;**

list\_add\_tail**(&new->**list**,** **&**d**->**list**);**

**goto** added**;**

**}**

**}** **else** **if** **(**d**->**devnum **==** **new->**devnum**)** **{**

/\* Required number taken \*/

**return** **-**EBUSY**;**

**}** **else** **if** **(**d**->**devnum **>** **new->**devnum**)** **{**

/\* Required number was free \*/

list\_add\_tail**(&new->**list**,** **&**d**->**list**);**

**goto** added**;**

**}**

last\_devnum **=** d**->**devnum**;**

**}**

**if** **(new->**devnum **==** **-**1**)**

**new->**devnum **=** last\_devnum**+**1**;**

**if** **((new->**devnum **<<** tr**->**part\_bits**)** **>** 256**)** **{**

**return** **-**EBUSY**;**

**}**

mutex\_init**(&new->**lock**);**

//将新的mtd\_blktrans\_dev设备添加到tr->devs链表中

list\_add\_tail**(&new->**list**,** **&**tr**->**devs**);**

added**:**

**if** **(!**tr**->**writesect**)**

**new->**readonly **=** 1**;**

//part\_bits为0，也就是该块设备只有一个分区

gd **=** alloc\_disk**(**1 **<<** tr**->**part\_bits**);**

//该块设备的主设备号

gd**->**major **=** tr**->**major**;**

//次设备号

gd**->**first\_minor **=** **(new->**devnum**)** **<<** tr**->**part\_bits**;**

//设置块设备操作函数

gd**->**fops **=** **&**mtd\_blktrans\_ops**;**

**if** **(**tr**->**part\_bits**)**

**if** **(new->**devnum **<** 26**)**

snprintf**(**gd**->**disk\_name**,** **sizeof(**gd**->**disk\_name**),**

"%s%c"**,** tr**->**name**,** 'a' **+** **new->**devnum**);**

**else**

snprintf**(**gd**->**disk\_name**,** **sizeof(**gd**->**disk\_name**),**

"%s%c%c"**,** tr**->**name**,**

'a' **-** 1 **+** **new->**devnum **/** 26**,**

'a' **+** **new->**devnum **%** 26**);**

**else**

//设置块设备名为/dev/mtdblockx

snprintf**(**gd**->**disk\_name**,** **sizeof(**gd**->**disk\_name**),**

"%s%d"**,** tr**->**name**,** **new->**devnum**);**

//设置块设备容量，需要除以512

set\_capacity**(**gd**,** **(new->**size **\*** tr**->**blksize**)** **>>** 9**);**

//将mtd\_blktrans\_dev设置为块设备的私有变量

gd**->**private\_data **=** **new;**

//反向设置

**new->**blkcore\_priv **=** gd**;**

//设置块设备的操作队列

gd**->**queue **=** tr**->**blkcore\_priv**->**rq**;**

**if** **(new->**readonly**)**

set\_disk\_ro**(**gd**,** 1**);**

//添加块设备/dev/mtdblockx

add\_disk**(**gd**);**

**return** 0**;**

**}**

看到这里，可能就会明白了当nandflash probe的时候，调用add\_mtd\_device就会产生对应字符设备和块设备了。

其中块设备的操作接口为：

struct block\_device\_operations mtd\_blktrans\_ops **=** **{**

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

**.**open **=** blktrans\_open**,**

**.**release **=** blktrans\_release**,**

**.**ioctl **=** blktrans\_ioctl**,**

**.**getgeo **=** blktrans\_getgeo**,**

**};**

当调用这些操作函数时，都会调用到mtdblock\_tr中对应的函数。open函数在加载驱动的时候就会调用。

##### 3.1.2.2 块设备的请求操作

当块设备有读写操作时，会触发mtd\_blktrans\_request函数。

static void mtd\_blktrans\_request**(**struct request\_queue **\***rq**)**

**{**

struct mtd\_blktrans\_ops **\***tr **=** rq**->**queuedata**;**

//唤醒mtd\_blktrans\_thread线程

wake\_up\_process**(**tr**->**blkcore\_priv**->**thread**);**

**}**

mtd\_blktrans\_request会唤醒mtd\_blktrans\_thread线程。

static int mtd\_blktrans\_thread**(**void **\***arg**)**

**{**

struct mtd\_blktrans\_ops **\***tr **=** arg**;**

struct request\_queue **\***rq **=** tr**->**blkcore\_priv**->**rq**;**

/\* we might get involved when memory gets low, so use PF\_MEMALLOC \*/

current**->**flags **|=** PF\_MEMALLOC **|** PF\_NOFREEZE**;**

spin\_lock\_irq**(**rq**->**queue\_lock**);**

**while** **(!**kthread\_should\_stop**())** **{**

struct request **\***req**;**

struct mtd\_blktrans\_dev **\***dev**;**

int res **=** 0**;**

req **=** elv\_next\_request**(**rq**);**

**if** **(!**req**)** **{**

set\_current\_state**(**TASK\_INTERRUPTIBLE**);**

spin\_unlock\_irq**(**rq**->**queue\_lock**);**

schedule**();**

spin\_lock\_irq**(**rq**->**queue\_lock**);**

**continue;**

**}**

//获取发生请求的mtd\_blktrans\_dev设备

dev **=** req**->**rq\_disk**->**private\_data**;**

//获取mtd\_blktrans\_dev设备的mtdblock\_tr

tr **=** dev**->**tr**;**

spin\_unlock\_irq**(**rq**->**queue\_lock**);**

mutex\_lock**(&**dev**->**lock**);**

//实际的申请函数

res **=** do\_blktrans\_request**(**tr**,** dev**,** req**);**

mutex\_unlock**(&**dev**->**lock**);**

spin\_lock\_irq**(**rq**->**queue\_lock**);**

end\_request**(**req**,** res**);**

**}**

spin\_unlock\_irq**(**rq**->**queue\_lock**);**

**return** 0**;**

**}**

最终会调用do\_blktrans\_request。

static int do\_blktrans\_request**(**struct mtd\_blktrans\_ops **\***tr**,**

struct mtd\_blktrans\_dev **\***dev**,**

struct request **\***req**)**

**{**

unsigned long block**,** nsect**;**

char **\***buf**;**

//要处理的扇区

block **=** req**->**sector **<<** 9 **>>** tr**->**blkshift**;**

//传送的扇区数目

nsect **=** req**->**current\_nr\_sectors **<<** 9 **>>** tr**->**blkshift**;**

//读写传送的buffer

buf **=** req**->**buffer**;**

**if** **(!**blk\_fs\_request**(**req**))**

**return** 0**;**

//检查是否在容量内

**if** **(**req**->**sector **+** req**->**current\_nr\_sectors **>** get\_capacity**(**req**->**rq\_disk**))**

**return** 0**;**

//判断方向

**switch(**rq\_data\_dir**(**req**))** **{**

**case** READ**:**

**for** **(;** nsect **>** 0**;** nsect**--,** block**++,** buf **+=** tr**->**blksize**)**

//调用mtdblock\_readsect

**if** **(**tr**->**readsect**(**dev**,** block**,** buf**))**

**return** 0**;**

**return** 1**;**

**case** WRITE**:**

**if** **(!**tr**->**writesect**)**

**return** 0**;**

**for** **(;** nsect **>** 0**;** nsect**--,** block**++,** buf **+=** tr**->**blksize**)**

//调用mtdblock\_writesect

**if** **(**tr**->**writesect**(**dev**,** block**,** buf**))**

**return** 0**;**

**return** 1**;**

**default:**

printk**(**KERN\_NOTICE "Unknown request %u\n"**,** rq\_data\_dir**(**req**));**

**return** 0**;**

**}**

**}**

根据请求数据的方向，分别调用mtdblock\_writesect和mtdblock\_readsect,在这里会完成数据的读写。

static int mtdblock\_readsect**(**struct mtd\_blktrans\_dev **\***dev**,**

unsigned long block**,** char **\***buf**)**

**{**

//mtdblks在mtdblock\_open的时候会将设备添加进去

//mtdblk\_dev用于数据缓冲

struct mtdblk\_dev **\***mtdblk **=** mtdblks**[**dev**->**devnum**];**

**return** do\_cached\_read**(**mtdblk**,** block**<<**9**,** 512**,** buf**);**

**}**

static int mtdblock\_writesect**(**struct mtd\_blktrans\_dev **\***dev**,**

unsigned long block**,** char **\***buf**)**

**{**

struct mtdblk\_dev **\***mtdblk **=** mtdblks**[**dev**->**devnum**];**

//如果mtdblk->cache\_data为空，则分配内存

**if** **(**unlikely**(!**mtdblk**->**cache\_data **&&** mtdblk**->**cache\_size**))** **{**

mtdblk**->**cache\_data **=** vmalloc**(**mtdblk**->**mtd**->**erasesize**);**

**if** **(!**mtdblk**->**cache\_data**)**

**return** **-**EINTR**;**

**}**

**return** do\_cached\_write**(**mtdblk**,** block**<<**9**,** 512**,** buf**);**

**}**

可以看到mtdblock\_tr结构体是整个mtd块设备操作的核心函数，几乎设备上的操作都会调用到里面相关的函数。

##### 3.1.2.3 mtdblock\_open

static int mtdblock\_open**(**struct mtd\_blktrans\_dev **\***mbd**)**

**{**

struct mtdblk\_dev **\***mtdblk**;**

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

int dev **=** mbd**->**devnum**;**

DEBUG**(**MTD\_DEBUG\_LEVEL1**,**"mtdblock\_open\n"**);**

**if** **(**mtdblks**[**dev**])** **{**

mtdblks**[**dev**]->**count**++;**

**return** 0**;**

**}**

/\* OK, it's not open. Create cache info for it \*/

mtdblk **=** kzalloc**(sizeof(**struct mtdblk\_dev**),** GFP\_KERNEL**);**

**if** **(!**mtdblk**)**

**return** **-**ENOMEM**;**

mtdblk**->**count **=** 1**;**

mtdblk**->**mtd **=** mtd**;**

mutex\_init**(&**mtdblk**->**cache\_mutex**);**

mtdblk**->**cache\_state **=** STATE\_EMPTY**;**

**if** **(** **!(**mtdblk**->**mtd**->**flags **&** MTD\_NO\_ERASE**)** **&&** mtdblk**->**mtd**->**erasesize**)** **{**

mtdblk**->**cache\_size **=** mtdblk**->**mtd**->**erasesize**;**

mtdblk**->**cache\_data **=** **NULL;**

**}**

mtdblks**[**dev**]** **=** mtdblk**;**

DEBUG**(**MTD\_DEBUG\_LEVEL1**,** "ok\n"**);**

**return** 0**;**

**}**

mtdblock\_open主要申请了mtdblk\_dev用于cache操作，并将mtdblk\_dev加入到mtdblks数组中。缓冲的长度为erasesize，也就是擦除块大小。

## 4. nandflash mtd驱动测试

mtd驱动的测试要使用mtd-utils里面的工具，后面会分析mtd-utils工具。

先新增一个分区用于测试操作，修改分区表，增加test分区。

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 **=** 0x03c00000**,**

**},**

**[**4**]** **=** **{**

**.**name **=** "test"**,**

**.**offset **=** MTDPART\_OFS\_APPEND**,**

**.**size **=** MTDPART\_SIZ\_FULL**,**

**}**

**};**

重新烧录修改了分区表的image。

# cat /proc/mtd

dev**:** size erasesize name

mtd0**:** 00040000 00020000 "bootloader"

mtd1**:** 00020000 00020000 "params"

mtd2**:** 00200000 00020000 "kernel"

mtd3**:** 03c00000 00020000 "root"

mtd4**:** 0c1a0000 00020000 "test"

将测试用的yaffs镜像拷贝到任意目录

# tftp -g -r yaffs 192.168.0.101

使用flash\_eraseall工具擦除测试分区

# flash\_eraseall /dev/mtd4

将测试用的yaffs镜像写入到/dev/mtd4中。

# nandwrite -o /dev/mtd4 yaffs

挂载。

# mount -t yaffs /dev/mtdblock4 test

这样可以在test目录下查看yaffs镜像中的内容。

## 5. mtd-utils

mtd-utils包含了很多工具，在应用层操作mtd分区进行擦除或者写入等操作。

下面来分析下这些工具的实现。

### 5.1 flash\_eraseall

擦除操作是按128k为单位进行的。

//打开/dev/mtd4字符设备

fd **=** open**(**mtd\_device**,** O\_RDWR**)**

//获取/dev/mtd4 nandflash信息

ioctl**(**fd**,** MEMGETINFO**,** **&**meminfo**)**

//获取擦除的块长度

erase**.**length **=** meminfo**.**erasesize**;**

//是否是nandflash

isNAND **=** meminfo**.**type **==** MTD\_NANDFLASH **?** 1 **:** 0**;**

//按块擦除

**for** **(**erase**.**start **=** 0**;** erase**.**start **<** meminfo**.**size**;** erase**.**start **+=** meminfo**.**erasesize**)** **{**

//调用擦除命令，操作驱动中mtd\_ioctl函数

**if** **(**ioctl**(**fd**,** MEMERASE**,** **&**erase**)** **!=** 0**)** **{**

fprintf**(**stderr**,** "\n%s: %s: MTD Erase failure: %s\n"**,** exe\_name**,** mtd\_device**,** strerror**(**errno**));**

**continue;**

**}**

**}**

### 5.2 nandwrite

//如果要写oob，就必须对齐，当pad为1时，代表操作长度可以不对齐，有冲突。

**if** **(**pad **&&** writeoob**)** **{**

fprintf**(**stderr**,** "Can't pad when oob data is present.\n"**);**

exit**(**1**);**

**}**

//获取mtd分区信息

ioctl**(**fd**,** MEMGETINFO**,** **&**meminfo**);**

//读取现在的oobinfo

ioctl **(**fd**,** MEMGETOOBSEL**,** **&**old\_oobinfo**);**

//如果不需要操作ecc，则将oobinfo设置为none\_oobinfo

**if** **(**noecc**)** **{**

ioctl **(**fd**,** MEMSETOOBSEL**,** **&**none\_oobinfo**);**

oobinfochanged **=** 1**;**

**}**

//打开要写入的文件

ifd **=** open**(**img**,** O\_RDONLY**));**

//获取文件长度

imglen **=** lseek**(**ifd**,** 0**,** SEEK\_END**);**

lseek **(**ifd**,** 0**,** SEEK\_SET**);**

//计算页长，如果需要写oob，页长应该加上oob

pagelen **=** meminfo**.**oobblock **+** **((**writeoob **==** 1**)** **?** meminfo**.**oobsize **:** 0**);**

//如果pad不为1，则需要检查img长度是否是页对齐

**if** **((!**pad**)** **&&** **((**imglen **%** pagelen**)** **!=** 0**))** **{**

fprintf **(**stderr**,** "Input file is not page aligned\n"**);**

**goto** closeall**;**

**}**

//检查img长度是否超出范围

**if** **(** **((**imglen **/** pagelen**)** **\*** meminfo**.**oobblock**)** **>** **(**meminfo**.**size **-** mtdoffset**))** **{**

fprintf **(**stderr**,** "Image %d bytes, NAND page %d bytes, OOB area %u bytes, device size %u bytes\n"**,**

imglen**,** pagelen**,** meminfo**.**oobblock**,** meminfo**.**size**);**

perror **(**"Input file does not fit into device"**);**

**goto** closeall**;**

**}**

**while** **(**imglen **&&** **(**mtdoffset **<** meminfo**.**size**))** **{**

//检查坏块并跳过

**while** **(**blockstart **!=** **(**mtdoffset **&** **(~**meminfo**.**erasesize **+** 1**)))** **{**

blockstart **=** mtdoffset **&** **(~**meminfo**.**erasesize **+** 1**);**

offs **=** blockstart**;**

baderaseblock **=** 0**;**

**if** **(!**quiet**)**

fprintf **(**stdout**,** "Writing data to block %x\n"**,** blockstart**);**

/\* Check all the blocks in an erase block for bad blocks \*/

**do** **{**

**if** **((**ret **=** ioctl**(**fd**,** MEMGETBADBLOCK**,** **&**offs**))** **<** 0**)** **{**

perror**(**"ioctl(MEMGETBADBLOCK)"**);**

**goto** closeall**;**

**}**

**if** **(**ret **==** 1**)** **{**

baderaseblock **=** 1**;**

**if** **(!**quiet**)**

fprintf **(**stderr**,** "Bad block at %x, %u block(s) from %x will be skipped\n"**,** **(**int**)** offs**,** blockalign**,** blockstart**);**

**}**

**if** **(**baderaseblock**)** **{**

mtdoffset **=** blockstart **+** meminfo**.**erasesize**;**

**}**

offs **+=** meminfo**.**erasesize **/** blockalign **;**

**}** **while** **(** offs **<** blockstart **+** meminfo**.**erasesize **);**

**}**

//如果pad=1，则表示可以操作任意长度文件的写入，需要将不足页的数据，手动添加0xff

//0xff代表不改变nandflash的内容

readlen **=** meminfo**.**oobblock**;**

**if** **(**pad **&&** **(**imglen **<** readlen**))**

**{**

readlen **=** imglen**;**

memset**(**writebuf **+** readlen**,** 0xff**,** meminfo**.**oobblock **-** readlen**);**

**}**

//读取数据

cnt **=** read**(**ifd**,** writebuf**,** readlen**);**

//如果要写oob

**if** **(**writeoob**)** **{**

//读oob

cnt **=** read**(**ifd**,** oobreadbuf**,** meminfo**.**oobsize**);**

//写oob

ioctl**(**fd**,** MEMWRITEOOB**,** **&**oob**);**

**}**

//写数据

pwrite**(**fd**,** writebuf**,** meminfo**.**oobblock**,** mtdoffset**);**

//更新写入位置和剩余写入长度

imglen **-=** readlen**;**

mtdoffset **+=** meminfo**.**oobblock**;**

**}**

### 5.3 nanddump

作用:dump出nand flash一些信息，如：block size,erasesize,oobblock 大小，oob data ,page data等；同时也会作坏块检测。

**./**nanddump **-**p **-**f nandinfo**.**txt **/**dev**/**mtd0

//dump出nand flash /dev/mtd0数据并保存到 nandinfo.txt

### 5.4 mtd\_debug

#./mtd\_debug info /dev/mtd0

// 输出/dev/mtd0上的一些信息,这里必须用mtdx

#./mtd\_debug erase /dev/mtd0 0x0 0x40000

// 擦出/dev/mtd0 分区上 从0x0开始的  ， 128K＊2 大小的数据

#./mtd\_debug write /dev/mtdblock0 ox0 0x360810 cq8401.img

//向mtdblock0分区，写入 3.6M 大小的文件系统cq8401.img,这里最好用mtdblockx

#./mtd\_debug read  /dev/mtdblock0 ox0 0x360810 read.img

//从mtdblock0中读出 3.6M 数据保存到read.img

# cmp -l cq8401.img read.img

// 验证write to flash 和 read from flash 中的数据是否一致;也可以使用diff命令来比较

另外针对nand flash，mtd\_debug这个工具来[测试](http://lib.csdn.net/base/softwaretest)mtd驱动也不是很好，用nandwrite和nanddump这两个工具或许更好点。然后可以用cmp这个命令来比较一下nanddump出来的数据和nandwrite写入的数据是否一致。mtd\_debug并不跳过坏块。

**mtd\_debug测试：**

//创建一个新的文件

# cat 1.c

12345678abcde

//将1.c的14个字节写入/dev/mtdblock4

# mtd\_debug write /dev/mtdblock4 0x0 14 /tmp/1.c

//从/dev/mtdblock4读14个字节到2.c

# mtd\_debug read /dev/mtdblock4 0x0 14 /tmp/2.c

//查看读到的数据，和写入的一样

# cat 2.c

12345678abcde

//从/dev/mtdblock4偏移为5的地方读14个字节到3.c

# mtd\_debug read /dev/mtdblock4 0x5 14 /tmp/3.c

//查看数据，12345没有了，正确

# cat 3.c

678abcde

//从/dev/mtd4偏移5的地方读取14个字节

# mtd\_debug read /dev/mtd4 0x5 14 /tmp/4.c

//结果正常

# cat 4.c

678abcde

//从/dev/mtd4中读取14个字节

# mtd\_debug read /dev/mtd4 0x0 14 /tmp/5.c

//结果正常

# cat 5.c

12345678abcde

//但是写/dev/mtd4任意数据和偏移就会出错，需要2048对齐

# mtd\_debug write /dev/mtd4 0x0 14 /tmp/5.c

to**:**3e60000**,**len**:**e

nand\_write**:** Attempt to write **not** page aligned data

file\_to\_flash**:** write**,** size 0xe**,** n 0xe

write**():** Invalid argument

测试中发现mtd\_debug write指令，如果操作mtdblock设备，可以写任意字节任意长度的数据，如果操作mtd字符设备，则必须2048对齐，因为mtdblock有缓冲设计。

JFFS2在flash上没有物理超级块, 它进行挂载文件系统的时候需要遍历flash存储。所以当操作flash\_eraseall一个mtd设备后，可以直接mount –t jffs2 /dev/mtdblockx tmp直接挂载。

## 6. norflash驱动分析

整个norflash驱动的代码自己要完成的部分非常的少，只需构造好map\_info结构体和分区表。

static struct map\_info **\***s3c\_nor\_map**;**

static struct mtd\_info **\***s3c\_nor\_mtd**;**

static struct mtd\_partition s3c\_nor\_parts**[]** **=** **{**

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

**.**name **=** "bootloader\_nor"**,**

**.**size **=** 0x00040000**,**

**.**offset **=** 0**,**

**},**

**[**1**]** **=** **{**

**.**name **=** "root\_nor"**,**

**.**offset **=** MTDPART\_OFS\_APPEND**,**

**.**size **=** MTDPART\_SIZ\_FULL**,**

**}**

**};**

static int s3c\_nor\_init**(**void**)**

**{**

/\* 1. 分配map\_info结构体 \*/

s3c\_nor\_map **=** kzalloc**(sizeof(**struct map\_info**),** GFP\_KERNEL**);;**

/\* 2. 设置: 物理基地址(phys), 大小(size), 位宽(bankwidth), 虚拟基地址(virt) \*/

s3c\_nor\_map**->**name **=** "s3c\_nor"**;**

s3c\_nor\_map**->**phys **=** 0**;**

s3c\_nor\_map**->**size **=** 0x1000000**;** /\* >= NOR的真正大小 \*/

s3c\_nor\_map**->**bankwidth **=** 2**;**

s3c\_nor\_map**->**virt **=** ioremap**(**s3c\_nor\_map**->**phys**,** s3c\_nor\_map**->**size**);**

simple\_map\_init**(**s3c\_nor\_map**);**

/\* 3. 使用: 调用NOR FLASH协议层提供的函数来识别 \*/

printk**(**"use cfi\_probe\n"**);**

s3c\_nor\_mtd **=** do\_map\_probe**(**"cfi\_probe"**,** s3c\_nor\_map**);**

/\* 4. add\_mtd\_partitions \*/

add\_mtd\_partitions**(**s3c\_nor\_mtd**,** s3c\_nor\_parts**,** 2**);**

**return** 0**;**

**}**

s3c\_nor\_map主要定义了norflash的物理地址，大小，位宽，虚拟地址等关键参数。再进行probe操作，得到norflash一些常用操作函数，构建mtd\_info结构体，并添加到mtd子系统中。

我们下将重点分析do\_map\_probe过程。

struct mtd\_info **\***do\_map\_probe**(**const char **\***name**,** struct map\_info **\***map**)**

**{**

struct mtd\_chip\_driver **\***drv**;**

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

//获取cfi\_chipdrv

drv **=** get\_mtd\_chip\_driver**(**name**);**

//调用cfi\_chipdrv->probe

ret **=** drv**->**probe**(**map**);**

module\_put**(**drv**->**module**);**

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

**return** ret**;**

**return** **NULL;**

**}**

首先调用get\_mtd\_chip\_driver获取cfi\_probe注册的mtd\_chip\_driver结构体。

那么get\_mtd\_chip\_driver 里面的内容是什么，cfi\_probe对应的mtd\_chip\_driver是在哪儿注册的呢？

get\_mtd\_chip\_driver函数如下：

static struct mtd\_chip\_driver **\***get\_mtd\_chip\_driver **(**const char **\***name**)**

**{**

struct list\_head **\***pos**;**

struct mtd\_chip\_driver **\***ret **=** **NULL,** **\*this;**

spin\_lock**(&**chip\_drvs\_lock**);**

list\_for\_each**(**pos**,** **&**chip\_drvs\_list**)** **{**

**this** **=** list\_entry**(**pos**,** typeof**(\*this),** list**);**

**if** **(!**strcmp**(this->**name**,** name**))** **{**

ret **=** **this;**

**break;**

**}**

**}**

**if** **(**ret **&&** **!**try\_module\_get**(**ret**->**module**))**

ret **=** **NULL;**

spin\_unlock**(&**chip\_drvs\_lock**);**

**return** ret**;**

**}**

这个函数非常的简单，从chip\_drvs\_list链表中获取mtd\_chip\_driver结构体。

cfi\_probe注册的相关代码如下：

void register\_mtd\_chip\_driver**(**struct mtd\_chip\_driver **\***drv**)**

**{**

spin\_lock**(&**chip\_drvs\_lock**);**

list\_add**(&**drv**->**list**,** **&**chip\_drvs\_list**);**

spin\_unlock**(&**chip\_drvs\_lock**);**

**}**

static struct mtd\_chip\_driver cfi\_chipdrv **=** **{**

**.**probe **=** cfi\_probe**,**

**.**name **=** "cfi\_probe"**,**

**.**module **=** THIS\_MODULE

**};**

static int \_\_init cfi\_probe\_init**(**void**)**

**{**

register\_mtd\_chip\_driver**(&**cfi\_chipdrv**);**

**return** 0**;**

**}**

再次回到do\_map\_probe函数，接着会调用mtd\_chip\_driver的probe函数，这里为cfi\_probe。

static struct chip\_probe cfi\_chip\_probe **=** **{**

**.**name **=** "CFI"**,**

**.**probe\_chip **=** cfi\_probe\_chip

**};**

struct mtd\_info **\***cfi\_probe**(**struct map\_info **\***map**)**

**{**

**return** mtd\_do\_chip\_probe**(**map**,** **&**cfi\_chip\_probe**);**

**}**

继续来看mtd\_do\_chip\_probe函数。

struct mtd\_info **\***mtd\_do\_chip\_probe**(**struct map\_info **\***map**,** struct chip\_probe **\***cp**)**

**{**

//map = s3c\_nor\_map

//cp = cfi\_chip\_probe

struct mtd\_info **\***mtd **=** **NULL;**

struct cfi\_private **\***cfi**;**

//探测芯片信息

cfi **=** genprobe\_ident\_chips**(**map**,** cp**);**

map**->**fldrv\_priv **=** cfi**;**

//获取和初始化芯片命令集

mtd **=** check\_cmd\_set**(**map**,** 1**);** /\* First the primary cmdset \*/

kfree**(**cfi**->**cfiq**);**

kfree**(**cfi**);**

map**->**fldrv\_priv **=** **NULL;**

**return** **NULL;**

**}**

genprobe\_ident\_chips用于norflash芯片的探测，并得到cfi\_private信息，里面保存了norflash操作的一些私有信息，最后map\_info的fldrv\_priv指针指向该结构体。

struct cfi\_private **{**

uint16\_t cmdset**;**

void **\***cmdset\_priv**;**

int interleave**;**

int device\_type**;**

int cfi\_mode**;** /\* Are we a JEDEC device pretending to be CFI? \*/

int addr\_unlock1**;**

int addr\_unlock2**;**

struct mtd\_info **\*(\***cmdset\_setup**)(**struct map\_info **\*);**

struct cfi\_ident **\***cfiq**;** /\* For now only one. We insist that all devs

must be of the same type. \*/

int mfr**,** id**;**

int numchips**;**

unsigned long chipshift**;** /\* Because they're of the same type \*/

const char **\***im\_name**;** /\* inter\_module name for cmdset\_setup \*/

struct flchip chips**[**0**];** /\* per-chip data structure for each chip \*/

**};**

genprobe\_ident\_chips()函数如果不考虑多芯片串连的情况，那只需看前面的genprobe\_new\_chip()调用，完成后cfi.chipshift=cfi.cfiq->DevSize，2^chipshift=FLASH大小。

genprobe\_new\_chip()枚举各种不同的芯片位宽和背靠背数量，结合配置设定依次调用cfi\_probe\_chip。这里我们假定访问内存芯片时数据宽度为4字节，但是，就具体芯片而言，其数据宽度却未必是32位，所以实际上可能需要有若干芯片并列凑成所需的数据宽度。例如：可以采用4个8位芯片并列，也可以采用2个16位芯片并列，当然，也可以采用单个32为芯片。

static int genprobe\_new\_chip**(**struct map\_info **\***map**,** struct chip\_probe **\***cp**,**struct cfi\_private **\***cfi**)**

**{**

int min\_chips **=** **(**map\_bankwidth**(**map**)/**4**?:**1**);** /\* At most 4-bytes wide. \*/

int max\_chips **=** map\_bankwidth**(**map**);** /\* And minimum 1 \*/

int nr\_chips**,** type**;**

**for** **(**nr\_chips **=** max\_chips**;** nr\_chips **>=** min\_chips**;** nr\_chips **>>=** 1**)**

**{**

**if** **(!**cfi\_interleave\_supported**(**nr\_chips**))**

**continue;**

cfi**->**interleave **=** nr\_chips**;**

/\* Minimum device size. Don't look for one 8-bit device

in a 16-bit bus, etc. \*/

type **=** map\_bankwidth**(**map**)** **/** nr\_chips**;**

**for** **(;** type **<=** CFI\_DEVICETYPE\_X32**;** type**<<=**1**)** **{**

cfi**->**device\_type **=** type**;**

**if** **(**cp**->**probe\_chip**(**map**,** 0**,** **NULL,** cfi**))**

**return** 1**;**

**}**

**}**

**return** 0**;**

**}**

其中：

map**->**bankwidth：MTD设备的接口总线宽度，其值可以为1，2或4。

cfi**->**interleave：交错数，几块芯片平行连接成一块芯片，使bankwidth变大。

cfi**->**device\_type：单个芯片的总线宽度，其值为1、2或4。

cfi\_probe\_chip做了一点什么呢？

首先发送如下指令：

cfi\_send\_gen\_cmd**(**0xF0**,** 0**,** base**,** map**,** cfi**,** cfi**->**device\_type**,** **NULL);**

cfi\_send\_gen\_cmd**(**0xFF**,** 0**,** base**,** map**,** cfi**,** cfi**->**device\_type**,** **NULL);**

cfi\_send\_gen\_cmd**(**0x98**,** 0x55**,** base**,** map**,** cfi**,** cfi**->**device\_type**,** **NULL);**

0xf0是芯片进入随机读取状态。

0x98是芯片开始一个CFI查询流程。

cfi\_send\_gen\_cmd源码如下：

static inline uint32\_t cfi\_send\_gen\_cmd**(**u\_char cmd**,** uint32\_t cmd\_addr**,** uint32\_t base**,**struct map\_info **\***map**,** struct cfi\_private **\***cfi**,**int type**,** map\_word **\***prev\_val**)**

**{**

map\_word val**;**

uint32\_t addr **=** base **+** cfi\_build\_cmd\_addr**(**cmd\_addr**,** cfi\_interleave**(**cfi**),** type**);**

val **=** cfi\_build\_cmd**(**cmd**,** map**,** cfi**);**

**if** **(**prev\_val**)**

**\***prev\_val **=** map\_read**(**map**,** addr**);**

map\_write**(**map**,** val**,** addr**);**

**return** addr **-** base**;**

**}**

cfi\_build\_cmd\_addr和cfi\_build\_cmd这两个函数根据具体的芯片格局对地址和命令作出换算和调整。在有多个芯片并列时，对芯片的命令要写入到并列的每一个芯片中，所以需要根据具体的情况将命令重复几次。

CFI查询中所用的地址都是以芯片本身的存储单元为单位的，而CPU使用的32位地址则是字节地址，所以要根据芯片的宽度加以换算。当采用多个芯片并列时实际上相当于改变了芯片宽度，因此需要进一步加以换算。

如富士通的29LV650：其容量是8Mbyte，共128个sector，每个 sector的大小是64 kbyte。如果选择位宽为x8，设备总线的每个地址代表了一个byte的存储单元，其总线地址范围为8M(0x000000~0x7fffff)。如果 选择位宽为x16，设备总线的每个地址代表了两个byte的存储单元，固其总线地址范围为4M(0x000000~0x3fffff)。

这里就可以解释需要错位连接的原因:ARM处理器的每个地址对应的是一个BYTE 的数据单元，而 16-BIT 的 FLASH 的每个地址对应的是一个 HALF-WORD(16-BIT)的数据单元。为了保持匹配，所以必须错位连接。这样，从ARM处理器发送出来的地址信号的最低位A0对16-BIT FLASH来说就被屏蔽掉了。

现在我们使用的芯片位宽为x16，所以如果要访问norflash地址0x10，cpu地址总线上发出的地址应该为0x20。

进入查询模式后，接下来便是要读取0x10，0x11，0x12三个地址的内容了，如果是QRY，说明芯片检测到了。如果不正确，则再切换一种设定。

static int \_\_xipram qry\_present**(**struct map\_info **\***map**,** \_\_u32 base**,**

struct cfi\_private **\***cfi**)**

**{**

int osf **=** cfi**->**interleave **\*** cfi**->**device\_type**;** // scale factor

map\_word val**[**3**];**

map\_word qry**[**3**];**

qry**[**0**]** **=** cfi\_build\_cmd**(**'Q'**,** map**,** cfi**);**

qry**[**1**]** **=** cfi\_build\_cmd**(**'R'**,** map**,** cfi**);**

qry**[**2**]** **=** cfi\_build\_cmd**(**'Y'**,** map**,** cfi**);**

val**[**0**]** **=** map\_read**(**map**,** base **+** osf**\***0x10**);**

val**[**1**]** **=** map\_read**(**map**,** base **+** osf**\***0x11**);**

val**[**2**]** **=** map\_read**(**map**,** base **+** osf**\***0x12**);**

**if** **(!**map\_word\_equal**(**map**,** qry**[**0**],** val**[**0**]))**

**return** 0**;**

**if** **(!**map\_word\_equal**(**map**,** qry**[**1**],** val**[**1**]))**

**return** 0**;**

**if** **(!**map\_word\_equal**(**map**,** qry**[**2**],** val**[**2**]))**

**return** 0**;**

**return** 1**;** // "QRY" found

**}**

现在已经检测到了闪存芯片，接下来需要将芯片中的cfi信息读取出来，相关函数为cfi\_chip\_setup。CFI规定了对芯片的品种、规格和各种参数进行查询的流程，以及芯片在受到查询时提供的数据结构。CFI规定，芯片在受到查询时，CPU可以从芯片上的单元地址0x10开始读回一个信息块。

**for** **(**i**=**0**;** i**<(sizeof(**struct cfi\_ident**)** **+** num\_erase\_regions **\*** 4**);** i**++)** **{**

**((**unsigned char **\*)**cfi**->**cfiq**)[**i**]** **=** cfi\_read\_query**(**map**,**base **+** **(**0x10 **+** i**)\***ofs\_factor**);**

**}**

回到mtd\_do\_chip\_probe函数，接下来要做的就是根据上面的查询结果构造mtd\_info结构体。

static struct mtd\_info **\***check\_cmd\_set**(**struct map\_info **\***map**,** int primary**)**

**{**

struct cfi\_private **\***cfi **=** map**->**fldrv\_priv**;**

\_\_u16 type **=** primary**?**cfi**->**cfiq**->**P\_ID**:**cfi**->**cfiq**->**A\_ID**;**

**if** **(**type **==** P\_ID\_NONE **||** type **==** P\_ID\_RESERVED**)**

**return** **NULL;**

**switch(**type**){**

/\* We need these for the !CONFIG\_MODULES case,

because symbol\_get() doesn't work there \*/

#ifdef CONFIG\_MTD\_CFI\_INTELEXT

**case** 0x0001**:**

**case** 0x0003**:**

**case** 0x0200**:**

**return** cfi\_cmdset\_0001**(**map**,** primary**);**

#endif

#ifdef CONFIG\_MTD\_CFI\_AMDSTD

**case** 0x0002**:**

**return** cfi\_cmdset\_0002**(**map**,** primary**);**

#endif

#ifdef CONFIG\_MTD\_CFI\_STAA

**case** 0x0020**:**

**return** cfi\_cmdset\_0020**(**map**,** primary**);**

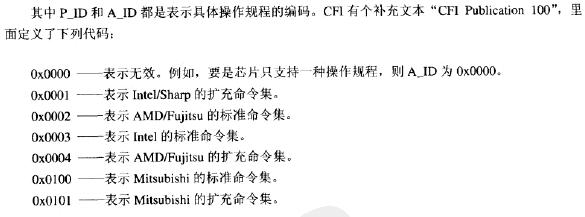
#endif

**default:**

**return** cfi\_cmdset\_unknown**(**map**,** primary**);**

**}**

**}**



我们这里使用的是cfi\_cmdset\_0002，下面摘取一些关键操作。

//mtd对应的实际操作函数

mtd**->**erase **=** cfi\_amdstd\_erase\_varsize**;**

mtd**->**write **=** cfi\_amdstd\_write\_words**;**

mtd**->**read **=** cfi\_amdstd\_read**;**

mtd**->**sync **=** cfi\_amdstd\_sync**;**

mtd**->**suspend **=** cfi\_amdstd\_suspend**;**

mtd**->**resume **=** cfi\_amdstd\_resume**;**

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

mtd**->**name **=** map**->**name**;**

mtd**->**writesize **=** 1**;**

//解锁命令

cfi**->**addr\_unlock1 **=** 0xaaa**;**

cfi**->**addr\_unlock2 **=** 0x555**;**

//region信息

**for** **(**j**=**0**;** j**<**cfi**->**numchips**;** j**++)** **{**

mtd**->**eraseregions**[(**j**\***cfi**->**cfiq**->**NumEraseRegions**)+**i**].**offset **=** **(**j**\***devsize**)+**offset**;**

mtd**->**eraseregions**[(**j**\***cfi**->**cfiq**->**NumEraseRegions**)+**i**].**erasesize **=** ersize**;**

mtd**->**eraseregions**[(**j**\***cfi**->**cfiq**->**NumEraseRegions**)+**i**].**numblocks **=** ernum**;**

**}**

//擦出块大小

mtd**->**erasesize **=** ersize**;**

到此为此norflash的mtd\_info已经构建好，驱动完成。

注意一点，norflash的每个region的大小可能是不一样的，在擦除的时候需要注意。

## 7. norflash mtd驱动测试

加载驱动的log信息：

# insmod s3c\_nor.ko

use cfi\_probe

s3c\_nor**:** Found 1 x16 devices at 0x0 in 16**-**bit bank

Amd**/**Fujitsu Extended Query Table at 0x0040

number of CFI chips**:** 1

cfi\_cmdset\_0002**:** Disabling erase**-**suspend**-**program due to code brokenness**.**

Creating 2 MTD partitions on "s3c\_nor"**:**

0x00000000**-**0x00040000 **:** "bootloader\_nor"

0x00040000**-**0x00200000 **:** "root\_nor"

查询分区状态：

# cat /proc/mtd

dev**:** size erasesize name

mtd0**:** 00040000 00020000 "bootloader"

mtd1**:** 00020000 00020000 "params"

mtd2**:** 00200000 00020000 "kernel"

mtd3**:** 0fda0000 00020000 "root"

mtd4**:** 00040000 00010000 "bootloader\_nor"

mtd5**:** 001c0000 00010000 "root\_nor"

擦除分区：

# flash\_eraseall /dev/mtd5

Erasing 64 Kibyte @ 1b0000 **--** 96 **%** complete**.**

挂载：

# mount -t jffs2 /dev/mtdblock5 test

写入：

# flashcp 1.c /dev/mtd5

读出：

# mtd\_debug read /dev/mtd5 0 6 /tmp/2.c

Copied 6 bytes from address 0x00000000 in flash to **/**tmp**/**2.c

# cat 2.c

123456

norflash的写入和读取操作就比较随意了，可以操作任意字节的写入和读取。

## 8. 总结

mtd驱动主要是构建mtd\_info结构体。nandflash通过nand\_chip协助mtd完成操作，而norflash通过map\_info协助mtd完成操作。

nandflash的读写操作都是按页来进行的，但是实际操作过程中，读是可以读任意地址/任意长度的，把整页读取出来，然后拷贝合适的部分。写操作则是必须地址对齐，长度可以不对齐，不需要操作的部分页数据填充0xff(nandwrite里面做了这些优化)。

norflash的读就比较自由了，可以随意操作。写的话一次性是写入两个字节，但是也可以写一个字节，将不操作的那个字节先读出来，再一并写进去。