## 一.移植gd5f2gq

1. nand\_ids.c

struct nand\_flash\_dev {

char \*name;

union {

struct {

uint8\_t mfr\_id;

uint8\_t dev\_id;

};

uint8\_t id[NAND\_MAX\_ID\_LEN];

};

unsigned int pagesize; //2k + 128B

unsigned int chipsize; //256M+ 16M

unsigned int erasesize; //=blocksize 128K + 8k

unsigned int options;//

uint16\_t id\_len;//2 byte

uint16\_t oobsize;//128B

struct {

uint16\_t strength\_ds;

uint16\_t step\_ds;

} ecc;

int onfi\_timing\_mode\_default;?

};

{"GD5F2GQ 2Gb 3.3V 8-bit",

{ .id = {0xc8,0x52} },

SZ\_2K, SZ\_256M, SZ\_128K, SP\_OPTIONS, 2, 128,

NAND\_ECC\_INFO(4, SZ\_528), 0 },

NAND\_ECC\_INFO(4,SZ\_528); internal ECC 4bits /528Byte

目前spinand 驱动需要以下三部分

1. vsi\_spi.c 驱动着 芯原的spi controller 控制器 master [配置寄存器]
2. gd5f2\_spinand.c 驱动着 spi nand flash ——slave; [发送命令码，协议的设置]
3. nand\_base.c 内核中间层,注册(实现nand operation 函数) 提供API 实例给上层.

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

-----------------------------提供API接口给上次(mtd层) ------------------------

vsi\_spi.c 对应spi controller驱动-----Nand\_base.c 内核中间层---- > gd5f2\_spinand.c 对应spi nand 驱动

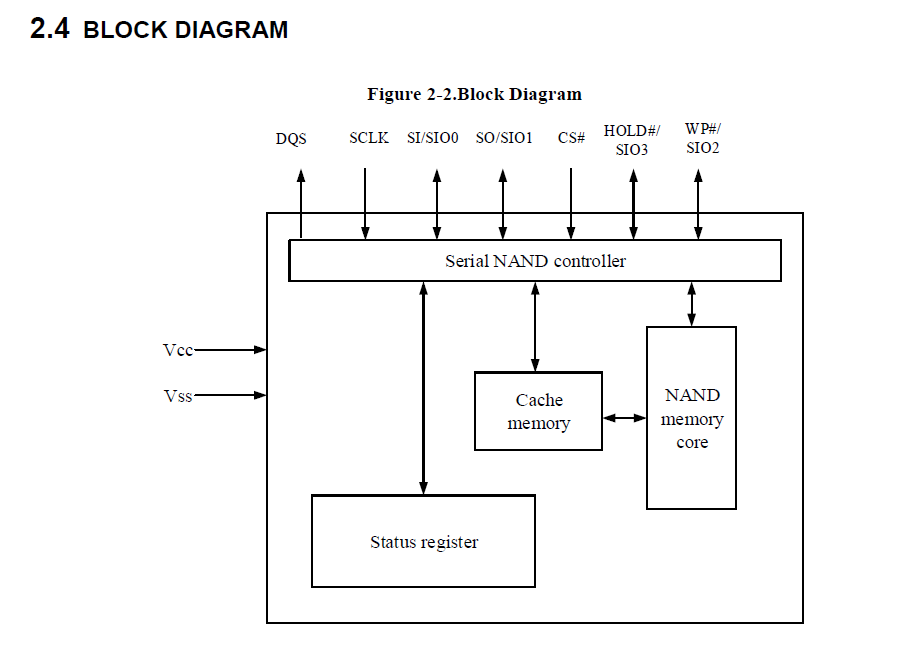
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

编程和读操作是基于 页的操作，擦除 操作是基于块的操作。

数据的传输(出和入) NAND memory 数组, page by page, 到 一个数据寄存器和一个cache 寄存器。

Cache 寄存器 是一个接近 I/O 控制的电路, 作为一个数据缓存区，给 NAND flash 内存的操作。Cache 内存的功能 作为一个缓存区 使能页和随机数据 读/写 以及拷贝回操作.

SPI nand flash 也使用 一个SPI 状态寄存器 ，来上报 operation 的状态.



1. 当内部 EEC 使能时, 使用者 可以编程 首个 64字节，后64字节不能编程. 但是整个128字节都是可读的。

2. 当内部的ECC 禁用时，使用者可以 读 和编程整个128 字节的spare area。

[ 1.543539] [zxf]spinand\_read\_id,device id 0xc8,0xd2

[ 1.548572] [zxf]spinand\_read\_id,device id 0xc8,0xd2

[ 1.553585] nand: device found, Manufacturer ID: 0xc8, Chip ID: 0xd2

[ 1.559989] nand: ESMT GD5F2GQ 2Gb 3.3V 8-bit

[ 1.564475] nand: 268435456 MiB, SLC, erase size: 128 KiB, page size: 2048, OOB size: 128

[ 1.572736] nand: WARNING: spi32766.0: the ECC used on your system is too weak compared to the one required by the NAND chip

[ 1.584039] ------------[ cut here ]------------

[ 1.588721] WARNING: CPU: 0 PID: 1 at mm/page\_alloc.c:3548 \_\_alloc\_pages\_nodemask+0x1ec/0x7e4

[ 1.597313] Modules linked in:

[ 1.600419] CPU: 0 PID: 1 Comm: swapper Not tainted 4.9.56 #16

[ 1.606299]

Call Trace:

[<80b7d7cc>] dump\_stack+0x10/0x20

[<809e2baa>] \_\_warn+0xc2/0xd0

[<80a29b70>] \_\_alloc\_pages\_nodemask+0x1ec/0x7e4

[<809e2c2a>] warn\_slowpath\_null+0x12/0x88

[<80a29b70>] \_\_alloc\_pages\_nodemask+0x1ec/0x7e4

[<8001b694>] initcall\_level\_names+0x0/0x20

[<8001b6b4>] initcall\_levels+0x0/0x24

[<80a39256>] kmalloc\_order+0x12/0x20

[<800010b8>] repair\_env\_string+0x0/0x78

[<80a4a6ba>] \_\_kmalloc+0xae/0xe8

[<80c348e4>] nand\_default\_bbt+0x6c/0x574

[<8001b694>] initcall\_level\_names+0x0/0x20

[<8001b694>] initcall\_level\_names+0x0/0x20

[<8001b6b4>] initcall\_levels+0x0/0x24

[<80c2fd48>] nand\_scan\_tail+0x588/0x5ac

[<800010b8>] repair\_env\_string+0x0/0x78

[<80c330de>] nand\_scan+0x16/0x18

[<80ceaac8>] spinand\_probe+0x120/0x1d0

[<80c39404>] spi\_drv\_probe+0x58/0x80

[<80bffb9c>] driver\_probe\_device+0x1d0/0x22c

[<80bffc66>] \_\_driver\_attach+0x6e/0x70

[<80bffbf8>] \_\_driver\_attach+0x0/0x70

[<80bfe716>] bus\_for\_each\_dev+0x36/0x58

[<80bff61e>] driver\_attach+0xe/0x10

[<80bff318>] bus\_add\_driver+0x98/0x128

[<80bfff94>] driver\_register+0x40/0x8c

[<80014df4>] spinand\_driver\_init+0x0/0x14

[<80c3938a>] \_\_spi\_register\_driver+0x2e/0x30

[<80014dfe>] spinand\_driver\_init+0xa/0x14

[<809dd0f0>] do\_one\_initcall+0x28/0x12c

[<809f4700>] parse\_args+0x18c/0x330

[<800010b8>] repair\_env\_string+0x0/0x78

[<8001b6b4>] initcall\_levels+0x0/0x24

[<8001b6b4>] initcall\_levels+0x0/0x24

[<80001832>] kernel\_init\_freeable+0xd2/0x1ac

[<800010b8>] repair\_env\_string+0x0/0x78

[<800209b8>] \_\_initcall\_spinand\_driver\_init6+0x0/0x4

[<80dfde94>] kernel\_init+0x0/0xc4

[<80dfde9c>] kernel\_init+0x8/0xc4

[<809dda88>] ret\_from\_kernel\_thread+0x8/0x10

[<800a0340>] \_\_initramfs\_start+0x7f610/0x9bb35d

[ 1.757265] ---[ end trace 67ccc1e16beb45b8 ]---

[ 1.769465] NET: Registered protocol family 17

在 nand\_base.c 文件中,

nand\_scan 流程如下:

[<80c2fd48>] nand\_scan\_tail+0x588/0x5ac

[<800010b8>] repair\_env\_string+0x0/0x78

[<80c330de>] nand\_scan+0x16/0x18

[<80ceaac8>] spinand\_probe+0x120/0x1d0

[<80c39404>] spi\_drv\_probe+0x58/0x80

[<80bffb9c>] driver\_probe\_device+0x1d0/0x22c

nand\_scan\_tail -- nand\_get\_flash\_type --chip->cmdfunc(mtd, NAND\_CMD\_READID, 0x00, -1); -----发送命令码读取NANDFLASH-CHIPID.

---将读取的id 与 注册的 **nand\_flash\_ids** 进行比较；看是id table 中是否添加。

**=======================================================**

## 二nand flash 背景知识

<https://www.crifan.com/files/doc/docbook/linux_nand_driver/release/html/linux_nand_driver.html>

<https://www.jianshu.com/p/89d4b6272281>

1. nand flash 基础常识

BBM ：bad block management 坏块管理

Nand Flash 由于其物理特性,只有有限的擦写次数，超过那个次数，基本上就是坏了。

当发现了，要及时将此block 标注为坏块，不再使用。

BBT: bad block table 坏块表

ECC：error correction code 错误校验 & 纠正机制

EEPROM：electrically erasable programmable read-only memory 电可擦除 只读存储器。

MLC : Multi level cell 多层单元

SLC: single level cell 单层单元

MTD: memory technology Device 内存技术设备

MOSFET： Metal-Oxide-semi-conductor field effect transistor 金属氧化物，半导体场效应晶体管。

NVM: non-volatile memory 非易失性 存储

NDA: none-disclosure agreement 非公开协议，保密协议。

OTP: one time programmable 一次性可编程(存储器)，一种非易失性存储器，但是只允许一次性写入数据，写入后，数据就不能修改了。

NAND ，硬盘则是MTP 高兴写几次就可以写几次。

坏块标记-Bad block mark

系统软件 初始化时，检测spare area 的首部的数据 是不是 0xFF;非0xFF- bad block.

，其中spare area 位于 每块的首页；检测spare area 要优先于任何对nand block 块操作。

Bad-block table 可以被创建，系统软件可以around 跳过这个bad-block area.

坏块表

如果块被擦除，坏块标记将不可能被恢复。

为了简化系统要求，和守护数据完整性。GigaDevice spi nand 提供了辅助assistant[auxiliary]管理坏块的选项。

在一个块内，对每一个页进行编程的话，必须是顺序的，而不能是随机的。

## 三坏块管理问题

对于坏块的管理，在linux 系统中，对应的会有一个表去记录。

在Linux 内核MTD 架构下的NAND Flash 驱动，在加载玩驱动之后，如果没有加入参数主动要求跳过坏块扫描的话，那么就会去主动扫描坏块，建立必要的BBT，以备后面坏块管理使用。

保证第一块是好的，缘因是用第一个块来存储 上面的BBT 坏块管理表。

1. create\_bbt() 函数

下面bbt\_mark\_entry(this,i ，BBT\_BLOCK\_FACTORY\_BAD); 检测工厂出厂时的坏块

nand\_scan\_tail() -----nand\_get\_flash\_type()

--chip->cmdfunc(mtd, NAND\_CMD\_READID, 0x00, -1);发送命令码

2.NAND\_SCAN\_BBT（）函数

Chip->scan\_bbt = nand\_default\_bbt；//chip->scan\_bbt会在每次nand\_scan\_tail 时被调用.

**插入一张内核默认坏块管理的数据结构类关系图**

类图(class diagram) 是显示了模型的静态结构，特别是模型中存在的类、类的内部结构以及它们与其他类的关系等。

类图是面向对象建模的主要组成部分。

<https://blog.csdn.net/HAOMCU/article/details/8181622?utm_medium=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel_param&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel_param>

闲来无事，追踪了下linux内核中对nand的坏块管理代码。大致记录一下。

内核中对nand的坏块管理是在nand的驱动中实现的，一般情况下，我们在实现nand控制器的驱动时不用考虑坏块的管理，这些机制已经在nand驱动的通用框架中实现了，我们要做的就是在nand驱动的框架上对接上nand控制器私有的操作与参数就可以了，例如读写函数以及nand控制器支持的ecc布局等。当然，这里主要是分析坏块管理的部分，就不多说驱动的开发了。

这里的分析是在bcm7231平台的nand驱动上面分析的。

nand驱动在加载的时候会调用nand\_scan\_tail函数，此函数的前面大部分是在确定所使用ecc方式和函数指针，以及一个页的读写需要作几次ecc校验等，这些都是根据nand驱动初始化的时候指定的一些参数决定的。在此函数的最后面开始关于BBT的操作：

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

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

return 0;

/\* Build bad block table \*/

return chip->scan\_bbt(mtd);

首先会检查chip->options中是否有NAND\_SKIP\_BBTSCAN选项，如果有，则直接返回，NAND\_SKIP\_BBTSCAN一般是在nand控制器驱动中选择是否被赋予的，一般不会赋予NAND\_SKIP\_BBTSCAN选项。如果没有，则调用chip->scan\_bbt()函数。chip->scan\_bbt()是一个函数指针，一般在nand\_set\_defaults中被赋值为nand\_default\_bbt。

首先我们看下struct nand\_chip结构体中关于BBT的部分：

/\*\*

\*... ...

\* @bbt: [INTERN] bad block table pointer

\* @bbt\_td: [REPLACEABLE] bad block table descriptor for flash

\* lookup.

\* @bbt\_md: [REPLACEABLE] bad block table mirror descriptor

\* @badblock\_pattern: [REPLACEABLE] bad block scan pattern used for initial

\* bad block scan.

\*... ...

\*/

struct nand\_chip {

……

uint8\_t \*bbt

struct nand\_bbt\_descr \*bbt\_td;

struct nand\_bbt\_descr \*bbt\_md;

struct nand\_bbt\_descr \*badblock\_pattern;

……

};

bbt是在内存中分配的一块内存（nand\_scan\_bbt中分配的，后面会说到），这块内存会存放BBT。如果定义了NAND\_BBT\_USE\_FLASH（这个可以在编写nand控制器驱动时选择），会先在nand中查找BBT是否存在，如果存在，然后读出放在bbt中，如果不存在（全新flash会有这种情况），则在bbt中建立BBT，然后写入nand中。如果没有定义NAND\_BBT\_USE\_FLASH，那么会直接在bbt指向的内存中建立BBT。

bbt\_td、bbt\_md和badblock\_pattern就是在nand\_default\_bbt函数中赋值的3个结构体。它们虽然是相同的结构体类型，但却有不同的作用和含义。其中bbt\_td和bbt\_md是主bbt和镜像bbt的描述符(镜像bbt主要用来对bbt的update和备份)，它们只在把bbt存储在NAND芯片的情况下使用，用来从NAND芯片中查找bbt。若bbt存储在内存中，bbt\_td和bbt\_md将会被赋值为NULL。

badblock\_pattern就是坏块信息的pattern，其中定义了坏块信息在oob中的存储位置，以及内容(即用什么值表示这个block是个坏块)。通常用1或2个字节来标志一个block是否为坏块，这1或2个字节就是坏块信息，如果这1或2个字节的内容是0xff，那就说明这个block是好的，否则就是坏块。对于坏块信息在NAND芯片中的存储位置，small page(每页512 Byte)和big page(每页2048 Byte)的两种NAND芯片不尽相同。一般来说，small page的NAND芯片，坏块信息存储在每个block的第一个page的oob的第六个字节中，而big page的NAND芯片，坏块信息存储在每个block的第一个page的oob的第1和第2个字节中。即使某种NAND芯片的坏块信息不是这样的存储方式也没关系，因为我们可以在badblock\_pattern中自己指定坏块信息的存储位置，以及用什么值来标志坏块(其实这个值表示的应该是“好块”，因为MTD会把从oob中坏块信息存储位置读出的内容与这个值做比较，若相等，则表示是个“好块”，否则就是坏块)。

下面是nand\_default\_bbt的实现：

/\*\*

\* nand\_default\_bbt - [NAND Interface] Select a default bad block table for the device

\* @mtd: MTD device structure

\*

\* This function selects the default bad block table support for the device and

\* calls the nand\_scan\_bbt function.

\*/

int nand\_default\_bbt(struct mtd\_info \*mtd)

{

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

/\*

\* Default for AG-AND. We must use a flash based bad block table as the

\* devices have factory marked \_good\_ blocks. Erasing those blocks

\* leads to loss of the good / bad information, so we \_must\_ store this

\* information in a good / bad table during startup.

\*/

if (this->options & NAND\_IS\_AND) {

/\* Use the default pattern descriptors \*/

if (!this->bbt\_td) {

this->bbt\_td = &bbt\_main\_descr;

this->bbt\_md = &bbt\_mirror\_descr;

}

this->bbt\_options |= NAND\_BBT\_USE\_FLASH;

return nand\_scan\_bbt(mtd, &agand\_flashbased);

}

/\* Is a flash based bad block table requested? \*/

if (this->bbt\_options & NAND\_BBT\_USE\_FLASH) {

/\* Use the default pattern descriptors \*/

if (!this->bbt\_td) {

if (this->bbt\_options & NAND\_BBT\_NO\_OOB) {

this->bbt\_td = &bbt\_main\_no\_bbt\_descr;

this->bbt\_md = &bbt\_mirror\_no\_bbt\_descr;

} else {

this->bbt\_td = &bbt\_main\_descr;

this->bbt\_md = &bbt\_mirror\_descr;

}

}

} else {

this->bbt\_td = NULL;

this->bbt\_md = NULL;

}

if (!this->badblock\_pattern)

nand\_create\_badblock\_pattern(this);

return nand\_scan\_bbt(mtd, this->badblock\_pattern);

}

在分析nand\_scan\_bbt函数之前先介绍下struct nand\_bbt\_descr的各成员的含义。其定义如下：

/\*\*

\* struct nand\_bbt\_descr - bad block table descriptor

\* @options: options for this descriptor

\* @pages: the page(s) where we find the bbt, used with option BBT\_ABSPAGE

\* when bbt is searched, then we store the found bbts pages here.

\* Its an array and supports up to 8 chips now

\* @offs: offset of the pattern in the oob area of the page

\* @veroffs: offset of the bbt version counter in the oob are of the page

\* @version: version read from the bbt page during scan

\* @len: length of the pattern, if 0 no pattern check is performed

\* @maxblocks: maximum number of blocks to search for a bbt. This number of

\* blocks is reserved at the end of the device where the tables are

\* written.

\* @reserved\_block\_code: if non-0, this pattern denotes a reserved (rather than

\* bad) block in the stored bbt

\* @pattern: pattern to identify bad block table or factory marked good /

\* bad blocks, can be NULL, if len = 0

\*

\* Descriptor for the bad block table marker and the descriptor for the

\* pattern which identifies good and bad blocks. The assumption is made

\* that the pattern and the version count are always located in the oob area

\* of the first block.

\*/

struct nand\_bbt\_descr {

int options;

int pages[NAND\_MAX\_CHIPS];

int offs;

int veroffs;

uint8\_t version[NAND\_MAX\_CHIPS];

int len;

int maxblocks;

int reserved\_block\_code;

uint8\_t \*pattern;

};

options：bad block table或者bad block的选项，可用的选择以及各选项具体表示什么含义，可以参考<linux/mtd/nand.h>。

pages：bbt专用。在查找bbt的时候，若找到了bbt，就把bbt所在的page号保存在这个成员变量中。若没找到bbt，就会把新建立的bbt的保存位置赋值给它。因为系统中可能会有多个NAND芯片，我们可以为每一片NAND芯片建立一个bbt，也可以只在其中一片NAND芯片中建立唯一的一个bbt，所以这里的pages是个维数为NAND\_MAX\_CHIPS的数值，用来保存每一片NAND芯片的bbt位置。当然，若只建立了一个bbt，那么就只使用pages[0]。

offs、len和pattern：MTD会从oob的offs中读出len长度的内容，然后与pattern指针指向的内容做比较，若相等，则表示找到了bbt，或者表示这个block是好的。

veroffs和version：bbt专用。MTD会从oob的veroffs中读出一个字节的内容，作为bbt的版本值保存在version中。

maxblocks：bbt专用。MTD在查找bbt的时候，不会查找NAND芯片中所有的block，而是最多查找maxblocks个block。

下面接着分析nand\_scan\_bbt函数。

bcm7231的nand控制器驱动中有这么一行：chip->bbt\_options |= NAND\_BBT\_USE\_FLASH | NAND\_BBT\_NO\_OOB;所以，上面函数的走向也很明确。

在最后调用了nand\_scan\_bbt函数。此函数的注释有这么一句话：

/\*\*

\* nand\_scan\_bbt - [NAND Interface] scan, find, read and maybe create bad block table(s)

... ...

\*/

可见，BBT的相关工作基本都是在这个函数内完成的。其函数实现如下：

int nand\_scan\_bbt(struct mtd\_info \*mtd, struct nand\_bbt\_descr \*bd)

{

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

int len, res = 0;

uint8\_t \*buf;

struct nand\_bbt\_descr \*td = this->bbt\_td;

struct nand\_bbt\_descr \*md = this->bbt\_md;

len = mtd->size >> (this->bbt\_erase\_shift + 2);

/\*

\* Allocate memory (2bit per block) and clear the memory bad block

\* table.

\*/

this->bbt = kzalloc(len, GFP\_KERNEL);

if (!this->bbt) {

printk(KERN\_ERR "nand\_scan\_bbt: Out of memory\n");

return -ENOMEM;

}

/\*

\* If no primary table decriptor is given, scan the device to build a

\* memory based bad block table.

\*/

if (!td) {

if ((res = nand\_memory\_bbt(mtd, bd))) {

pr\_err("nand\_bbt: can't scan flash and build the RAM-based BBT\n");

kfree(this->bbt);

this->bbt = NULL;

}

return res;

}

verify\_bbt\_descr(mtd, td);

verify\_bbt\_descr(mtd, md);

/\* Allocate a temporary buffer for one eraseblock incl. oob \*/

len = (1 << this->bbt\_erase\_shift);

len += (len >> this->page\_shift) \* mtd->oobsize;

buf = vmalloc(len);

if (!buf) {

printk(KERN\_ERR "nand\_bbt: Out of memory\n");

kfree(this->bbt);

this->bbt = NULL;

return -ENOMEM;

}

/\* Is the bbt at a given page? \*/

if (td->options & NAND\_BBT\_ABSPAGE) {

res = read\_abs\_bbts(mtd, buf, td, md);

} else {

/\* Search the bad block table using a pattern in oob \*/

res = search\_read\_bbts(mtd, buf, td, md);

}

if (res)

res = check\_create(mtd, buf, bd);

/\* Prevent the bbt regions from erasing / writing \*/

mark\_bbt\_region(mtd, td);

if (md)

mark\_bbt\_region(mtd, md);

vfree(buf);

return res;

}

上面已经说了，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。在内存中建立BBT的过程是这样的：

直接调用nand\_memory\_bbt函数，nand\_memory\_bbt函数的主要工作就是在内存中建立bbt，其实就是调用了create\_bbt函数。create\_bbt函数的工作方式很简单，就是扫描NAND芯片所有的block，读取每个block中第一个page的oob内容，然后根据oob中的坏块信息建立起bbt，这坏块信息就是根据badblock\_pattern来确定的。可以参见关于struct nand\_bbt\_descr中的offs、len和pattern成员变量的解释。

相对与BBT存储在内存中来说，BBT存储在nand中的过程就稍微复杂一点了。

几下来就是在flash中创建或读取BBT了。

verify\_bbt\_descr(mtd, td);与verify\_bbt\_descr(mtd, md);没有做实际的动作，只是检查一下相关的参数。接下来由if (td->options & NAND\_BBT\_ABSPAGE)可知，从flash上读取读取BBT又分为两种情况。

第一种情况是定义了NAND\_BBT\_ABSPAGE，那么调用read\_abs\_bbts函数直接从给定的page地址读取。那么这个page地址在什么时候指定呢？就是在你的NAND driver中指定。前文说过，在struct nand\_chip结构体中有两个成员变量，分别是bbt\_td和bbt\_md，MTD为它们附上了default的值，但是你也可以根据你的需要为它们附上你自己定义的值。假如你为bbt\_td和bbt\_md的options成员变量定义了NAND\_BBT\_ABSPAGE，同时又把你的bbt所在的page地址保存在bbt\_td和bbt\_md的pages成员变量中，MTD就可以直接在这个page地址中读取bbt了。值得一提的是，在实际使用时一般不这么干，因为你不能保证你保存bbt的那个block就永远不会坏，而且这样也不灵活。

第二种情况是调用那个search\_read\_bbts函数试着在NAND芯片的maxblocks(请见上文关于struct nand\_bbt\_descr中maxblocks的说明)个block中查找bbt是否存在，若找到，就可以读取bbt了。

MTD查找bbt的过程为：如果你在bbt\_td和bbt\_md的options 成员变量中定义了 NAND\_BBT\_LASTBLOCK，那么MTD就会从NAND芯片的最后一个block开始查找(在default情况下，MTD就是这么干的)，否则就从第一个block开始查找。

与查找oob中的坏块信息时类似，MTD会从所查找block的第一个page的oob中读取内容，然后与bbt\_td或bbt\_md中patter指向的内容做比较，若相等，则表示找到了bbt，否则就继续查找下一个block。顺利的情况下，只需查找一个block中就可以找到bbt，否则MTD最多会查找maxblocks个block。若找到了bbt，就把该bbt所在的page地址保存到bbt\_td或bbt\_md的pages成员变量中，否则pages的值为-1。

如果系统中有多片NAND芯片，并且为每一片NAND芯片都建立一个bbt，那么就会在每片NAND芯片上重复以上过程。

接着，nand\_scan\_bbt函数会调用check\_create函数，该函数会判断是否找到了bbt，其实就是判断bbt\_td或者bbt\_md中pages成员变量的值是否有效。若找到了bbt，就会把bbt从NAND芯片中读取出来，并保存到struct nand\_chip中bbt指针指向的内存中；若没找到，就会调用create\_bbt函数建立bbt(与bbt存储在内存中时情况一样)，同时把bbt写入到NAND芯片中去。

## 四 软件方面

<https://blog.csdn.net/feihongwang/article/details/8840109?utm_medium=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel_param&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel_param>

如果想要在linux 下编程nand flash驱动，需要搞清楚Linux 下，此部分的框架.

需要知道： 1. 系统是如何管理你的nand flash 的？

1. 系统帮你做了哪些工作？

剩下的驱动底层实现部分，要实现哪些功能？

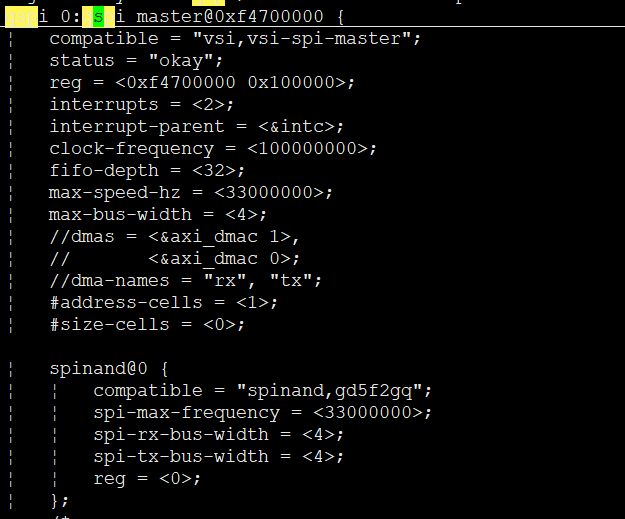
ONFI: open nand flash interface specification 公开 nand flash 接口规范。

ONFI是Intel主导的，其他一些厂家（Hynix，Micron，Numonyx，Phison ，SanDisk，Sony，Spansion等）参与制定的，统一了Nand Flash的操作接口。

Linux 下nand flash 驱动编写步骤

Step1. 驱动架构

Dts文件中对应：



驱动模块的匹配match

static const struct of\_device\_id spinand\_dt[] = {

{ .compatible = "spinand,gd5f2gq", },

{}

};

MODULE\_DEVICE\_TABLE(of, spinand\_dt);

/\*

\* Device name structure description

\*/

static struct spi\_driver spinand\_driver = {

.driver = {

.name = "gd5f2gq",

.of\_match\_table = spinand\_dt,

},

.probe = spinand\_probe, //真正的spi nand 驱动

.remove = spinand\_remove,

};

module\_spi\_driver(spinand\_driver);

MODULE\_DESCRIPTION("SPI NAND driver for GagiDevice Chip");

MODULE\_AUTHOR("Henry Pan <hspan@micron.com>");

MODULE\_LICENSE("GPL v2");

/\*\*

\* spinand\_probe - [spinand Interface]

\* @spi\_nand: registered device driver.

\*

\* Description:

\* Set up the device driver parameters to make the device available.

\*/

static int spinand\_probe(struct spi\_device \*spi\_nand)

{

/\*step1:驱动中struct nand\_chip chip变量的申请和初始化【如果没有，则表示使用linux 默认的接口】；\*/

chip->read\_buf = spinand\_read\_buf;

chip->write\_buf = spinand\_write\_buf;

chip->read\_byte = spinand\_read\_byte;

chip->cmdfunc = spinand\_cmdfunc;

chip->waitfunc = spinand\_wait;

chip->options |= NAND\_CACHEPRG;

chip->select\_chip = spinand\_select\_chip;

/\*step2: nand\_scan() 接口调用\*/

扫描硬件nand 设备；

读取到实际nand device 的chip id与nand\_ids.c 中的nand\_flash\_ids[] 表进行比对；比对成功，将设置表中的pagesize，chipsize，erasesize/block size ,oob size ,option等.

/\*step3. mtd\_device\_register() 接口调用\*/

注册mtd 设备；

解析分区表，然后创建MTD设备，从而被上层应用使用。

}

<https://www.cnblogs.com/hfyinsdu/p/5601381.html>

## MTD 子系统

设计的目的

致力于work on 一个通用的针对内存设备的子系统，主要是flash 设备。

MTD 子系统的目的是为了简化新硬件的驱动，通过提供通用的接口，介于硬件驱动和上层系统之间。 —— 硬件抽象层 hardware abstract layer

硬件驱动不需要知道存储的格式，仅仅只要提供简易的程序： read、write 和erase.

以适当的形式向用户展示设备的内容将由系统的上层处理。

Presentation of the device’s contents to the user in an appropriate form

MTD NAND Driver Programming Interface

<https://www.kernel.org/doc/html/latest/driver-api/mtdnand.html?highlight=nand>

**介绍**

通用的 NAND 驱动支持 大部分的 NAND chip 芯片，并将nand 连接到 MTD 内核子系统中.

此文档提供给开发者，想要实现板级驱动或文件系统驱动 合适于suitable for nand device.

文档提示hints

此函数和结构体的文档都是自动生成,每个函数和结构体成员都有一个简短的介绍，使用中括号[XXX] 来标识.以下章节解释了 identifiers 标志符的意义。

**基本的板级驱动**

至少，你应该提供 一个nand\_chip 结构体 以及 经过映射ioremap 过的 chip address 芯片地址。 Struct nand\_chip 结构体可以动态申请或静态申请，并且 NAND chip 结构体中嵌入了一个 struct mtd\_info 结构体，两者之间可以通过 引用和contain\_of() 来转换获取到。

高级 板级驱动函数

此nand 驱动可以控制 芯片组。 所以板级驱动 必须提供自己的 select\_chip() 函数。

Select\_chip() 函数 用于选择需要的chip, select\_chip 在结构体struct nand\_chip 中有他的函数指针，在调用 nand\_scan() 函数之前要设置好select\_chip 函数指针。

Nand\_scan() 函数，会遍历扫描 nand\_flash\_table【内核的全局变量】,与读取到的nand flash 的chip id 进行匹配，如果匹配成功，说明find the nand ，然后进行nand\_chip 内部数据的填充。匹配不成功，则没有找到该chip.

nand\_scan() 的参数 maxchip 代表要最多扫描多少颗芯片。—— 该驱动程序要驱动几个chip. 使用在一个板子上含有两个或两个以上同样的nand flash 芯片的情况下，兼容.

同时确保 select\_chip 函数可以 处理 在多个chip 之间切换。

## Linux 中宿主结构体 与 成员结构体 之间的相互转换

一、由 struct mtd\_info 获取到 struct nand\_chip 结构体变量;[成员结构体 -> 宿主结构体]

static inline struct nand\_chip \*mtd\_to\_nand(struct mtd\_info \*mtd)

{

return container\_of(mtd, struct nand\_chip, mtd);

}

二、由struct nand\_chip 结构体变量 获取到 struct mtd\_info 结构体变量.[宿主结构体->成员结构体]

static inline struct mtd\_info \*nand\_to\_mtd(struct nand\_chip \*chip)

{

return &chip->mtd;

}

其中， struct nand\_chip 包含struct mtd\_info mtd;

故而: 1. chip->mtd 直接获取struct mtd\_info，

2. container\_of(mtd, struct nand\_chip, mtd); 获取struct nand\_chip。

struct nand\_chip {

struct mtd\_info mtd;

void \_\_iomem \*IO\_ADDR\_R;

void \_\_iomem \*IO\_ADDR\_W;

## Linux 中大小端的相互转换

static int spinand\_program\_execute(struct spi\_device \*spi\_nand, u32 page\_id)

row = page\_id;

cmd.n\_addr = 3;

cmd.addr[0] = (u8)((row & 0xff000000) >> 24);//page is row address

cmd.addr[1] = (u8)((row & 0xff0000) >> 16);//MSB & LSB first addr is addr[23]

cmd.addr[2] = (u8)((row & 0xff00) >> 8);

根据时序图，先发送addr 23,22,21…0 ; 所以大小端进行切换.

