网名“鱼树”的学员聂龙浩，

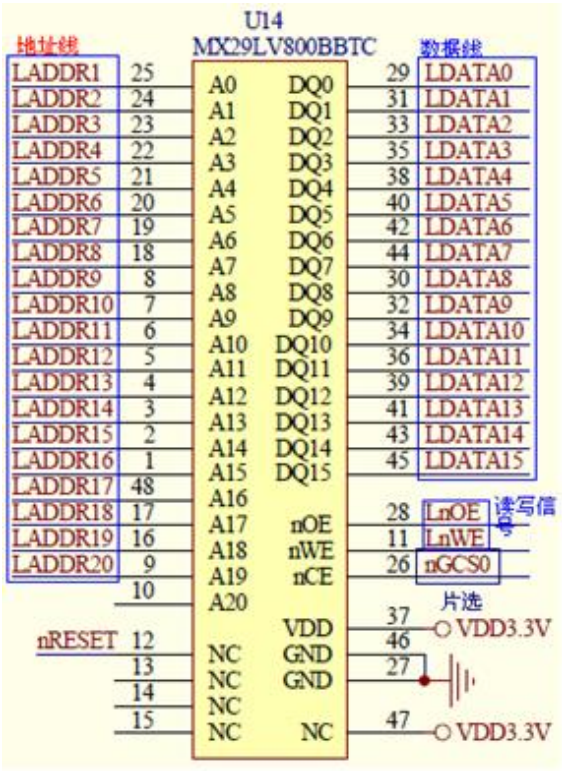
学习“韦东山 Linux 视频第 2 期”时所写的笔记很详细，供大家参考。

也许有错漏，请自行分辨。

目录

NOR 原理及硬件操作

NOR 在 UBOOT 中的操作：



NOR 的接口与内存的接口是一样的，而 NAND（数据线只有 7 条，发地址又发命令和数据等）。NOR 可以像内存一样的来读，但不能像内存一样的去写。
NOR 的烧写要发出某些特定的命令，在某地址上写某个值就称为命令。

NOR 存放关键性的代码，如 bootload\内核或文件系统。而 NADN 有位反转的缺点，如存一些海量数据如视频监控文件等。

用 NOR 启动时，CPU 看到的“0”地址是“NOR”FLASH。若为 NAND 启动，则 CPU 看到的“0”地址是 2440 片的 4K 内存。用 NAND 启动时 CPU 完全看不到 NOR，只有用 NOR 启动时，CPU 看到的“0”地址才是 NOR。

“XIP”（代码直接在芯片上运行）：
代码可以直接在 NOR 上运行，在 NAND 上不行。CPU 可以直接从 NOR 上取到指令来运行。而用 NAND 启动，2440 是有将 NAND 的前 4K 数据自动拷贝到 2440 的 4K 片内内存里，这时 CPU 再从片内 4K 内存中取指令运行。

使用 UBOOT 体验 NOR FLASH 的操作(开发板设为 NOR 启动，进入 UBOOT)

先使用 OpenJTAG 烧写 UBOOT 到 NOR FLASH

TABLE 5. MX29LV800BT/BB COMMAND DEFINITIONS

Command		Bus	First Bus Cycle		Second Bus Cycle		Third Bus Cycle		Fourth Bus Cycle		Fifth Bus Cycle		Sixth Bus Cycle	
		Cycle	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Reset		1	XXXH	F0H										
Read		1	RA	RD										
Read Silicon ID	Word	4	555H	AAH	2AAH	55H	555H	90H	ADI	DDI				
	Byte	4	AAAH	AAH	555H	55H	AAAH	90H	ADI	DDI				
Sector Protect Verify	Word	4	555H	AAH	2AAH	55H	555H	90H	(SA)	XX00H				
									x02H	XX01H				
	Byte	4	AAAH	AAH	555H	55H	AAAH	90H	(SA)	00H				
									x04H	01H				
Program	Word	4	555H	AAH	2AAH	55H	555H	A0H	PA	PD				
	Byte	4	AAAH	AAH	555H	55H	AAAH	A0H	PA	PD				
Chip Erase	Word	6	555H	AAH	2AAH	55H	555H	80H	555H	AAH	2AAH	55H	555H	10H
	Byte	6	AAAH	AAH	555H	55H	AAAH	80H	AAAH	AAH	555H	55H	AAAH	10H
Sector Erase	Word	6	555H	AAH	2AAH	55H	555H	80H	555H	AAH	2AAH	55H	SA	30H
	Byte	6	AAAH	AAH	555H	55H	AAAH	80H	AAAH	AAH	555H	55H	SA	30H
Sector Erase Suspend		1	XXXH	B0H										
Sector Erase Resume		1	XXXH	30H										
CFI Query	Word	1	55H	98										
	Byte	1	AAH	98										

Reset：复位，往“任何地址”写入“F0H”便是复位了。

Read Silicon ID：NOR 可以配置位宽为“byte-8 位”或“word-16 位”。

对于原理图上的 NOR 位宽是 16 位（看顶端原理图 NOR 有 16 条数据线）。

那么读 ID 是往“555H”这个地址写入“AAH”，接着再往“2AAH”地址写入“55H”，再往“555H”地址写入“90H”。然后就可以在地址“ADI”读到数据“DDI”。

ADI = Address of Device identifier;（设备识别的地址）

A1=0, A0 = 0 for manufacturer code： 当A1=0,A0=0时，读到的是“厂家ID”。

A1=0, A0 = 1 for device code.： 当A1=0,A0=1时，读到的是“设备ID”。

A2-A18=do not care.

(Refer to table 3)

DDI = Data of Device identifier（设备识别的数据） :

C2H for manufacture code（厂家ID是 C2H）, 22DA/DA(Top), and 225B/5B(Bottom) for device code.

X = X can be VIL or VIH

RA=Address of memory location to be read.

RD=Data to be read at location RA.

Table 9. Am29LV160D Command Definitions

Command Sequence (Note 1)			Cycles	Bus Cycles (Notes 2–5) First Addr	First Data	Second Addr	Second Data	Third Addr	Third Data	Fourth Addr	Fourth Data	Fifth Addr	Fifth Data	Sixth Addr	Sixth Data
Read (Note 6)			1	RA	RD										
Reset (Note 7)			1	XXX	F0										
Autoselect (Note 8)	Manufacturer ID	Word	4	555	AA	2AA	55	555	90	X00	01				
		Byte		AAA		555		AAA							
	Device ID, Top Boot Block	Word	4	555	AA	2AA	55	555	90	X01	22C4				
		Byte		AAA		555		AAA		X02	C4				
	Device ID, Bottom Boot Block	Word	4	555	AA	2AA	55	555	90	X01	2249				
		Byte		AAA		555		AAA		X02	49				
	Sector Protect Verify (Note 9)	Word	4	555	AA	2AA	55	555	90	(SA) X02	XX00 XX01				
		Byte		AAA		555		AAA		(SA) X04	00 01				

这一款 NORFLASH，当为 16 位位宽时，先从“555”处写入“AA”，再从“2AA”处写入“55”，接着在“555”处写入“90”,最后从 0 地址“X00”处可以读到“01”。

周期“First”和“Second”是“解锁”，周期“Third”是发出命令。

1. 读数据：

md.b 0 NOR 可以像内存一样读。

```
OpenJTAG> md.b 0
00000000: 17 00 00 ea 14 f0 9f e5 14 f0 9f e5 14 f0 9f e5    ................
00000010: 14 f0 9f e5 14 f0 9f e5 14 f0 9f e5 14 f0 9f e5    ................
00000020: 60 01 f8 33 c0 01 f8 33 20 02 f8 33 80 02 f8 33    `..3...3 ..3...3
00000030: e0 02 f8 33 00 04 f8 33 20 04 f8 33 ef be ad de    ...3...3 ..3....
OpenJTAG>
```

在 UBOOT 命令中读到的数据。

```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 17 00 00 EA 14 F0 9F E5 14 F0 9F E5 14 F0 9F E5 ; ...???????????
00000010h: 14 F0 9F E5 14 F0 9F E5 14 F0 9F E5 14 F0 9F E5 ; ????????????????
00000020h: 60 01 F8 33 C0 01 F8 33 20 02 F8 33 80 02 F8 33 ; `..3??? .?3.?3
00000030h: E0 02 F8 33 00 04 F8 33 20 04 F8 33 EF BE AD DE ; ??..? .?3??
00000040h: 00 00 F8 33 00 00 F8 33 AC 06 FB 33 04 79 FB 33 ; ..?..???.y?
00000050h: DE C0 AD 0B DE C0 AD 0B 00 00 00 00 DE C0 AD 0B ; ??????....??
00000060h: DE C0 AD 0B 00 00 0F E1 1F 00 C0 E3 D3 00 80 E3 ; ??...?.??e?
00000070h: 00 F0 29 E1 53 04 A0 E3 00 10 A0 E3 00 10 80 E5 ; .?酬.友..友..e?
```

Uboot.bin 烧写镜像上的数据。

内容完全一样。

2. 读 ID：解锁 与 命令，读数据。

NOR 手册上:16 位宽时

Read Silicon ID	Word	4	555H	AAH	2AAH	55H	555H	90H	ADI	DDI

（ADI = Address of Device identifier; A1=0, A0 = 0 for manufacturer code,A1=0, A0 = 1 for device code.）

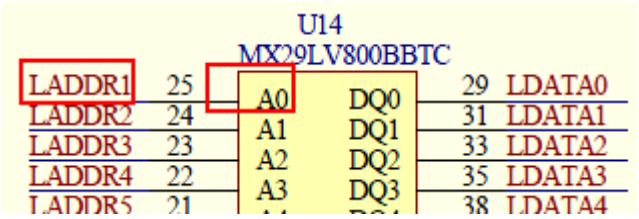
（DDI = Data of Device identifier : C2H for manufacture code, 22DA/DA(Top), and 225B/5B(Bottom) for

device code.

X = X can be VIL or VIH

）

往地址 555H 写 AAH
往地址 2AAH 写 55H
---上两个解锁
往地址 555H 写 90H
---命令
读 0 地址得到厂家 ID：C2H
读 1 地址得到设备 ID：22DAH 或 225BH
退出读 ID 状态：给任意地址写 F0H



2440 的地址 1 接到 NOR 的地址 0，并没有一一对应。而是相关“1 位”。

2440 的 A1 接到 NOR 的 A0，所以 2440 发出(555h<<1：相当于 2440 的 A0 地址补 0)，NOR 才能收到 555h 这个地址。

NOR 上看到地址“555h”，这时 CPU 发出来的地址是相当于（555h<<1）.有移位。

移位的原因：

1，地址线错位了 1 位。

2，因为 NOR 是 16 位的，那么 CPU 发一个地址到 NOR，从 NOR 上得到的是 2 字节的数据。CPU 里面认为的地址总是“一个地址 对应 一个字节”，而上面 NOR 有 16 条数据线，即位宽是 16 位，则 NOR 认为一个地址对应的是两个字节。

那么这里错了一位，就能有种对齐?

UBOOT 怎么操作？

左移一位相当于乘以 2，如 555h * 2 = AAAh

555h<<1，即 AAAh,往地址 AAAH 写 AAH　　mw.w aaa aa （mw.w 是以 16 位写）

2AAh<<1，即往地址 554 写 55H　　mw.w 554 55

---上两个解锁

555h<<1，即往地址 AAAH 写 90H　　mw.w aaa 90

---发命令

地址 0 还是 0，读 0 地址得到厂家 ID：C2H　　md.w 0 1

---从 0 地址读 1 次得到厂家 ID：C2H

地址 1 乘以 2 为 2，则读 2 地址得到设备 ID：22DAH 或 225BH　　md.w 2 1

退出读 ID 状态：（退出后才能读到 NORFLASH 中的其他数据）　　mw.w 0 f0

```
OpenJTAG> mw.w aaa aa
OpenJTAG> mw.w 554 55
OpenJTAG> mw.w aaa 90
OpenJTAG> md.w 0 1
00000000: 00c2    ..
OpenJTAG> md.w 2 1
00000002: 2249    I"
OpenJTAG>
```

解锁

发90命令
从0地址读1个数据

上面设备 ID 与手册上的不一致，这并没有什么。

3. NOR 有两种规范，jedec，cfi(common flash interface)：

jedec 规范，就包含了“MX29LV800BT/BB COMMAND DEFINITIONS”中的命令，如发命令可以识别 ID，擦除芯片：

Command		Bus Cycle	First Bus Cycle		Second Bus Cycle		Third Bus Cycle		Fourth Bus Cycle		Fifth Bus Cycle		Sixth Bus Cycle	
			Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Chip Erase	Word	6	555H	AAH	2AAH	55H	555H	80H	555H	AAH	2AAH	55H	555H	10H
	Byte	6	AAAH	AAH	555H	55H	AAAH	80H	AAAH	AAH	555H	55H	AAAH	10H

和擦除“扇区”：

Sector Erase	Word	6	555H	AAH	2AAH	55H	555H	80H	555H	AAH	2AAH	55H	SA	30H
	Byte	6	AAAH	AAH	555H	55H	AAAH	80H	AAAH	AAH	555H	55H	SA	30H

再或“烧写数据”：

Program	Word	4	555H	AAH	2AAH	55H	555H	A0H	PA	PD				
	Byte	4	AAAH	AAH	555H	55H	AAAH	A0H	PA	PD				

老式的 NORFLASH 会支持“jedec”规范。要知道容量有多大，就要读出 ID，与数组比较。这个数组有记录此 NORFLASH 芯片的信息。内核中有一个数组：

linux-2.6.22.6\drivers\mtd\chips\Jedec_probe.c ：

const struct amd_flash_info jedec_table[]

```
static const struct amd_flash_info jedec_table[] = {
    {
        .mfr_id      = MANUFACTURER_AMD,
        .dev_id      = AM29F032B,
        .name        = "AMD AM29F032B",
        .uaddr       = {
            [0] = MTD_UADDR_0x0555_0x02AA /* x8 */
        },
        .DevSize     = SIZE_4MiB,
        .CmdSet      = P_ID_AMD_STD,
        .NumEraseRegions= 1,
        .regions     = {
            ERASEINFO(0x10000,64)
        }
    }, {
```

以前老式的 NORFLASH 规范就必须读 ID 后，与这个内核中的 "jedec_table[]"结构数组比较，若是此数据中没有你的 NORFLASH 的信息，就得来修改此数组。这样就麻烦了。新的

NORFLASH 要支持“cfi”通用 flash 接口。就是说 NORFLASH 里面本身就带了这些属性信息（如容量、电压）。往相关地址发相关命令好可以得这些属性信息。

读取 CFI 信息：

NOR 手册：

进入 CFI 模式　　往 55H 写入 98H （这之后便可以读数据）
读数据:　　读 10H 得到 0051 ACSII 为“Q”
　　读 11H 得到 0052 ACSII 为“R”
　　读 12H 得到 0059 ACSII 为“Y”
　　读 27H 得到容量

Command		Bus Cycle	First Bus Cycle		Second Bus Cycle		Third Bus Cycle		Fourth Bus Cycle		Fifth Bus Cycle		Sixth Bus Cycle	
			Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
CFI Query	Word	1	55H	98										
	Byte	1	AAH	98										

TABLE 4-1. CFI mode: Identification Data Values
(All values in these tables are in hexadecimal)

Description	Address (Byte Mode)	Address (Word Mode)	Data
Query-unique ASCII string "QRY"	20	10	0051
	22	11	0052
	24	12	0059
Primary vendor command set and control interface ID code	26	13	0002
	28	14	0000
Address for primary algorithm extended query table	2A	15	0040
	2C	16	0000
Alternate vendor command set and control interface ID code (none)	2E	17	0000
	30	18	0000
Address for secondary algorithm extended query table (none)	32	19	0000
	34	1A	0000

从这里可知，往 55H 写入“98H”后进入“CFI”模式，从“10H”就可以读到数据“0051”。“0051”是"Q"的 ASCII 码。所以上面“QRY--查询”的 ASCII 为“51 52 59”。

TABLE 4-2. CFI Mode: System Interface Data Values
(All values in these tables are in hexadecimal)

Description	Address (Byte Mode)	Address (Word Mode)	Data
VCC supply, minimum (2.7V) 从"1B"处读到最小电压	36	1B	0027
VCC supply, maximum (3.6V) 从"1C"处读到最大电压	38	1C	0036
VPP supply, minimum (none)	3A	1D	0000
VPP supply, maximum (none)	3C	1E	0000
Typical timeout for single word/byte write (2^N us)	3E	1F	0004
Typical timeout for Minimum size buffer write (2^N us)	40	20	0000
Typical timeout for individual block erase (2^N ms)	42	21	000A
Typical timeout for full chip erase (2^N ms)	44	22	0000
Maximum timeout for single word/byte write times (2^N X Typ)	46	23	0005
Maximum timeout for buffer write times (2^N X Typ)	48	24	0000
Maximum timeout for individual block erase times (2^N X Typ)	4A	25	0004
Maximum timeout for full chip erase times (not supported)	4C	26	0000

Description 从"27H"处可以读到容量	Address (Byte Mode)	Address (Word Mode)	Data
Device size (2^N bytes)	4E	27	0014

从“27H”地址处读到一个值，这个值（如上为 14H）的 2 的 N 次方就是此 NORFLASH 的容量。这里是 14H 的十进制为 N。即 N=20,2 的 20 次方为“1048576”，再除以 1024*1024 即此 NORFLASH 为 1M 的容量。

2440 的 A1 接到 NOR 的 A0，所以 2440 发出(555h<<1)，NOR 才能收到 555h 这个地址
UBOOT 怎么操作？

进入 CFI 模式	往 AAH 写入 98H	mw.w aa 98
读数据:	读 20H 得到 0051	md.w 20 1
	读 22H 得到 0052	md.w 22 1
	读 24H 得到 0059	md.w 24 1
	读 4EH 得到容量	md.w 4e 1
	退出 CFI 模式	mw.w 0 f0 （往 0 地址写入“f0-复位”退出 CFI 模式）

```
OpenJTAG>  mw.w aa 98
OpenJTAG> md.w 20 1
00000020: 0051    Q.
OpenJTAG> md.w 22 1
00000022: 0052    R.
OpenJTAG> md.w 24 1
00000024: 0059    Y.
OpenJTAG> md.w 4e 1
0000004e: 0015    ..
OpenJTAG>
```

“15H”十进制为“21”，那么 2 的 21 次方是“2097152”即 2M（除 1024*1024）。

Erase block region 1 information (refer to the CFI publication 100)	5A	2D	0000
	5C	2E	0000
	5E	2F	0040
	60	30	0000
Erase block region 2 information	62	31	0001
	64	32	0000
	66	33	0020
	68	34	0000

NORFLASH 支持不同的区域每次擦除的大小不同，如某区域是 32K 每次的擦除，而另一区域是 64K 每次的擦除。

4. 写数据:

在地址 0x100000 写入 0x1234（在前面写会破坏 UBOOT，故选择了 1M 处。UBOOT 没有 1M.）这里的 0x100000 1M 是 CPU 看到的地址。

UBOOT 中写：

```
md.w 100000 1      // 得到ffff  读出10000（1M）处的数据为“FFFF”
mw.w 100000 1234
md.w 100000 1      // 还是ffff  即写不进去
```

NORFLASH 可以像内存一样的读，但不是像内存一样写。

```
OpenJTAG> md.w 100000 1
00100000: ffff    ..
OpenJTAG> mw.w 100000 1234
OpenJTAG> md.w 100000 1
00100000: ffff    ..
OpenJTAG> md.w 30000000 1
30000000: 5a7f    .Z
OpenJTAG> mw.w 30000000 1234
OpenJTAG> md.w 30000000 1
30000000: 1234    4.
OpenJTAG>
```


30000000 地址处对应的是内存，内存可以直接写，而 100000 是 NORFLASH 的 1M 处不能像内存一样写。

NORFLASH 的写前，要保证要的地址是擦除过的，即此地址处是“全 1”

NOR 手册：program 烧写

往地址 555H 写 AAH

往地址 2AAH 写 55H

--上两个为解锁

往地址 555H 写 AOH

往地址 PA 写 PD （PA：program address；PD:program data）

Command	Bus	First Bus Cycle		Second Bus Cycle		Third Bus Cycle		Fourth Bus Cycle		Fifth Bus Cycle		Sixth Bus Cycle	
	Cycle	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data

Program	Word	4	555H	AAH	2AAH	55H	555H	A0H	PA	PD				
	Byte	4	AAAH	AAH	555H	55H	AAAH	A0H	PA	PD				
Chip Erase	Word	6	555H	AAH	2AAH	55H	555H	80H	555H	AAH	2AAH	55H	555H	10H
	Byte	6	AAAH	AAH	555H	55H	AAAH	80H	AAAH	AAH	555H	55H	AAAH	10H
Sector Erase	Word	6	555H	AAH	2AAH	55H	555H	80H	555H	AAH	2AAH	55H	SA	30H
	Byte	6	AAAH	AAH	555H	55H	AAAH	80H	AAAH	AAH	555H	55H	SA	30H

2440 的 A1 接到 NOR 的 A0，所以 2440 发出(555h<<1)，NOR 才能收到 555h 这个地址。

UBOOT 怎么操作？

往地址AAAH写AAH	mw.w aaa aa
往地址554H写55H	mw.w 554 55
往地址AAAH写A0H	mw.w aaa a0
往地址0x100000写1234h	mw.w 100000 1234 （100000是CPU看到的地址）

```
OpenJTAG> mw.w aaa aa
OpenJTAG> mw.w 554 55
OpenJTAG> mw.w aaa a0
OpenJTAG> mw.w 100000 1234
OpenJTAG> md.w 100000 1
00100000: 1234
OpenJTAG>
```

NOR 驱动框架：

NOR 中有个协议层，发出“解锁命令”等，也是内核做好的。NOR 可以像内存一样读。
两个开发板之间，内存的差别就是内存的地址和位宽。

NOR 硬件操作：差异部分（基地址、位宽），如在“NOR 协议层”知道往“某地址”写“某数据”来“识别、擦除、烧写”，这个“某地址”就要自己提供。
　可以写一个通用驱动程序，“基地址”和“位宽”直接通过“make menuconfig”配置内核来实现。
linux-2.6.22.6\drivers\mtd\maps\Physmap.c 这是内核中自带的 NORFLASH 驱动程序（最底层-硬件相关层）。

"MTD"层不管下面是“NAND”或是“NOR”，因为它们都会提供“mtd_info 结构”给“MTD 层”。

一，NORFLASH 硬件操作：

1，分配结构：map_info（map 映射信息）结构体
2，设置结构：物理基地址(phys)，大小(size)，位宽(bankwidth)，虚拟基地址(virt) 。内核看到不同开发板 NOR 的最小差别。
3，使用：（调用 NOR FLASH 协议层提供的函数来识别）用 mtd_info 结构提供的函数来识别等等。
4,add_mtd_partitions 加上分区信息。

分配结构体：

结构体 map_info：其中有“名字、物理地址、大小、位宽”
struct physmap_flash_info 结构

```
struct physmap_flash_info {
    struct mtd_info     *mtd;
    struct map_info     map;
    struct resource     *res;
#ifdef CONFIG_MTD_PARTITIONS
    int             nr_parts;
    struct mtd_partition    *parts;
#endif
};
```

设置结构：

设置名字、物理地址、大小和位宽、虚拟地址等

```
info->map.name = dev->dev.bus_id;
info->map.phys = dev->resource->start;
info->map.size = dev->resource->end - dev->resource->start + 1;
info->map.bankwidth = physmap_data->width;
info->map.set_vpp = physmap_data->set_vpp;
```

将物理地址映射成虚拟地址：

```
info->map.virt = ioremap(info->map.phys, info->map.size);
```

简单的实始化：

```
simple_map_init(&info->map);
```

使用：

do_map_probe() 返回一个“mtd_info”结构体。Do_map_probe()是 NORFLASH 协议层提供的函数。

```
static const char *rom_probe_types[] = { "cfi_probe", "jedec_probe", "map_rom", NULL };
probe_type = rom_probe_types;
for (; info->mtd == NULL && *probe_type != NULL; probe_type++)
    info->mtd = do_map_probe(*probe_type, &info->map);
```

struct mtd_info *do_map_probe(const char *name, struct map_info *map)
参 1“名字”：是一个 rom_probe_types 数组中的某一项，如名字可以是“cfi_probe”
"jedec_probe"等规范。

识别出来后：

```
add_mtd_partitions(info->mtd, info->parts, err);
```

Physmap.c 分析：

```
int __init physmap_init(void)
-->platform_device_register(&physmap_flash); 平台设备（配置信息：基地址、长度、位宽等）
  physmap_flash:
  -->.resource = &physmap_flash_resource, 资源
资源：
    -->.start = CONFIG_MTD_PHYSMAP_START, 起始地址
       .end             = CONFIG_MTD_PHYSMAP_START + CONFIG_MTD_PHYSMAP_LEN - 1, 起始地址+长度-1
       .flags = IORESOURCE_MEM,
```

知道物理地址 ioremap。ioremap 需要基地址和长度

```
  -->.dev          = {.platform_data          = &physmap_flash_data,},
```

```
    physmap_flash_dat:
    -->.width  = CONFIG_MTD_PHYSMAP_BANKWIDTH, 位宽
-->platform_driver_register(&physmap_flash_driver); 平台驱动
  physmap_flash_driver：
  -->.probe          = physmap_flash_probe,
    分配 map_info 结构：
struct physmap_flash_info {
struct mtd_info                    *mtd;
struct map_info                    map;
struct resource                    *res;
#ifdef CONFIG_MTD_PARTITIONS
int                              nr_parts;
struct mtd_partition          *parts;
#endif
};
    -->struct physmap_flash_info *info = kzalloc(sizeof(struct physmap_flash_info),
GFP_KERNEL);
    设置 map_info 结构：
    -->info->map.name = dev->dev.bus_id;  名字
   info->map.phys = dev->resource->start; 物理地址
info->map.size = dev->resource->end - dev->resource->start + 1; 大小
    Info->map.size在后面nor协议层识别出来之后，可以和这里比较.
info->map.bankwidth = physmap_data->width; 位宽
info->map.virt = ioremap(info->map.phys, info->map.size);虚拟基地址使用
simple_map_init(&info->map); 简单的初始化
-->char *rom_probe_types[] = { "cfi_probe", "jedec_probe", "map_rom", NULL };
   probe_type = rom_probe_types;
   info->mtd = do_map_probe(*probe_type, &info->map) 以cfi_probe等类型识别NORFLASH。
```

二，利用内核自带的 NORFLASH 底层驱动程序“Physmap.c”做测试：

内核里有这一个底层的 NORFLASH 驱动程序，这时可以直接拿来做实验：
测试 1:通过配置内核支持 NOR FLASH

1. make menuconfig

```
-> Device Drivers
  -> Memory Technology Device (MTD) support        //内存技术设备支持
    -> Mapping drivers for chip access
    <M> CFI Flash device in physical memory map    //M作为模块
    (0x0) Physical start address of flash mapping  // 物理基地址 写为0
```

(0x1000000) Physical length of flash mapping // 长度 这里写大些为16M,此长度要大于等于NORFLASH的真实长度即可。因为它要ioremap。设置长度的意思是让它ioremap映射得到虚拟地址。

(2) Bank width in octets (NEW) // 位宽 16，单位是 octets（8 的意思）.故设为 2.

2. make modules（不需 make uImage）得到“physmap.ko”。

这是 NORFLASH 最底层硬件相关的驱动程序。

cp drivers/mtd/maps/physmap.ko /work/nfs_root/first_fs

```
book@book-desktop:/work/system/linux-2.6.22.6$ make modules
scripts/kconfig/conf -s arch/arm/Kconfig
drivers/serial/Kconfig:235:warning: multi-line strings not supported
  CHK     include/linux/version.h
  SYMLINK include/asm-arm/arch -> include/asm-arm/arch-s3c2410
make[1]: `include/asm-arm/mach-types.h' is up to date.
  CHK     include/linux/utsrelease.h
  CALL    scripts/checksyscalls.sh
<stdin>:91:1: warning: "__IGNORE_sync_file_range" redefined
In file included from <stdin>:2:
include/asm/unistd.h:444:1: warning: this is the location of the previous definition
<stdin>:1097:2: warning: #warning syscall fadvise64 not implemented
<stdin>:1265:2: warning: #warning syscall migrate_pages not implemented
<stdin>:1321:2: warning: #warning syscall pselect6 not implemented
<stdin>:1325:2: warning: #warning syscall ppoll not implemented
<stdin>:1365:2: warning: #warning syscall epoll_pwait not implemented
  CC [M]  drivers/mtd/maps/physmap.o
drivers/mtd/maps/physmap.c:204:2: warning: #warning using PHYSMAP compat code
  Building modules, stage 2.
  MODPOST 27 modules
  CC      drivers/mtd/maps/physmap.mod.o
  LD [M]  drivers/mtd/maps/physmap.ko
book@book-desktop:/work/system/linux-2.6.22.6$ cp drivers/mtd/maps/physmap.ko /work/nfs_root/first_fs
book@book-desktop:/work/system/linux-2.6.22.6$
```

3. 启动开发板

ls /dev/mtd*

insmod physmap.ko

```
# ls physmap.ko
physmap.ko
# insmod physmap.ko
physmap platform flash device: 01000000 at 00000000
physmap-flash.0: 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.
cmdlinepart partition parsing not available
RedBoot partition parsing not available
#
```

发现了一个 1X16 的设置，在 0 地址，位宽是 16 位。
最后说分区没找到。

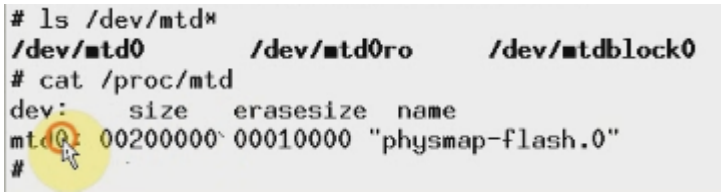
ls /dev/mtd*

```
# ls /dev/mtd*
/dev/mtd0       /dev/mtd0ro     /dev/mtdblock0
#
```

分区/dev/mtdblock0 出现，是把整个 NORFLASH 作成了一个分区。要构建分区：
设置一个“mtd_partition”s3c_nor_part 数组，划分好分区。识别出 FLASH 后：

add_mtd_partitions(s3c_mtd, s3c_nor_part, 4);

cat /proc/mtd



size 大小是 2M，擦除块的大小是 64K。

分区信息：

```
#ifdef CONFIG_MTD_PARTITIONS  //解析和设置分区信息
err = parse_mtd_partitions(info->mtd, part_probe_types, &info->parts, 0);
if (err > 0) {
add_mtd_partitions(info->mtd, info->parts, err);
return 0;
}

if (physmap_data->nr_parts) {
printk(KERN_NOTICE "Using physmap partition information\n");
add_mtd_partitions(info->mtd, physmap_data->parts,
physmap_data->nr_parts);
return 0;
}
#endif
```

```
char *part_probe_types[] = { "cmdlinepart", "RedBoot", NULL };
parse_mtd_partitions(info->mtd, part_probe_types, &info->parts, 0);解析分区
```

应该是通过命令行传入一些参数来解析分区，用命令行指定划分分区。这可能比较麻烦，直接用：
add_mtd_partitions(mtd_info 结构体，mtd_partition 分区信息数组，数组中分区个数)。

NOR 驱动程序编写

编译：

```
"Makefile" 10L, 176C written
book@book-desktop:/work/drivers_and_test/15th_nor/1th$ make
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M]  /work/drivers_and_test/15th_nor/1th/s3c_nor.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /work/drivers_and_test/15th_nor/1th/s3c_nor.mod.o
  LD [M]  /work/drivers_and_test/15th_nor/1th/s3c_nor.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
book@book-desktop:/work/drivers_and_test/15th_nor/1th$
```

卸载原来的驱动：

```
#  rmmod physmap
Device 'physmap-flash.0' does not have a release() function, it is broken and must be fixed.
WARNING: at drivers/base/core.c:107 device_release()
[<c002ede8>] (dump_stack+0x0/0x14) from [<c01b743c>] (device_release+0x84/0x98)
[<c01b73b8>] (device_release+0x0/0x98) from [<c0177714>] (kobject_cleanup+0x68/0x80)
[<c01776ac>] (kobject_cleanup+0x0/0x80) from [<c0177740>] (kobject_release+0x14/0x18)
 r7:00000000 r6:c3cde000 r5:c017772c r4:bf000f2c
[<c017772c>] (kobject_release+0x0/0x18) from [<c01784d0>] (kref_put+0x8c/0xa4)
[<c0178444>] (kref_put+0x0/0xa4) from [<c01776a4>] (kobject_put+0x20/0x28)
 r5:bf001060 r4:bf000ea4
[<c0177684>] (kobject_put+0x0/0x28) from [<c01b7b44>] (put_device+0x1c/0x20)
[<c01b7b28>] (put_device+0x0/0x20) from [<c01bc25c>] (platform_device_put+0x1c/0x20)
[<c01bc240>] (platform_device_put+0x0/0x20) from [<c01bc584>] (platform_device_unregister+0x1c/0x20)
[<c01bc568>] (platform_device_unregister+0x0/0x20) from [<bf000400>] (physmap_exit+0x14/0x28 [physmap])
 r4:c035d014
[<bf0003ec>] (physmap_exit+0x0/0x28 [physmap]) from [<c00610e4>] (sys_delete_module+0x214/0x29c)
[<c0060ed0>] (sys_delete_module+0x0/0x29c) from [<c002aea0>] (ret_fast_syscall+0x0/0x2c)
 r8:c002b044 r7:00000081 r6:0009fbac r5:bedb1ed4 r4:000a104c
#
```

这是说原来的驱动程序中，平台设备中需要一个“release()”函数。所以没法卸载（卸载平台设备？）。这个是个警告。

```
static void led_release(struct device * dev)
{
}

static struct platform_device led_dev = {
    .name           = "myled",
    .id         = -1,
    .num_resources      = ARRAY_SIZE(led_resource),
    .resource       = led_resource,
    .dev = {
        .release = led_release,
    },
};
```

而内核中的 NOR 硬件驱动的平台设备里没有提供“release()”函数：

```
static struct platform_device physmap_flash = {
    .name           = "physmap-flash",
    .id     = 0,
    .dev            = {
        .platform_data  = &physmap_flash_data,
    },
    .num_resources  = 1,
    .resource       = &physmap_flash_resource,
};
```

结果和内核自带的 NOR 硬件驱动""一样：

```
# 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.
#
```

上面显示是用“cfi_probe”规范识别出来 NORFLASH 的，没有用到“jedec”规范。

添加分区：

//4.1,定义分区信息结构数组.

```
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,
}
};
```

//4,add_mtd_partitions 加上分区信息

```
    add_mtd_partitions (s3c_nor_map, s3c_nor_parts, 2);
```

测试 2：使用自己写的驱动程序：

1. ls /dev/mtd*
2. insmod s3c_nor.ko
3. ls /dev/mtd*

```
# rmmod s3c_nor
# ls /dev/mtd*
ls: /dev/mtd*: No such file or directory
# 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"
#
```

```
# ls /dev/mtd*
/dev/mtd0       /dev/mtd1       /dev/mtdblock0
/dev/mtd0ro     /dev/mtd1ro     /dev/mtdblock1
# ls /dev/mtd* -l
crw-rw----    1 0        0          90,   0 Jan  1 00:21 /dev/mtd0
crw-rw----    1 0        0          90,   1 Jan  1 00:21 /dev/mtd0ro
crw-rw----    1 0        0          90,   2 Jan  1 00:21 /dev/mtd1
crw-rw----    1 0        0          90,   3 Jan  1 00:21 /dev/mtd1ro
brw-rw----    1 0        0          31,   0 Jan  1 00:21 /dev/mtdblock0
brw-rw----    1 0        0          31,   1 Jan  1 00:21 /dev/mtdblock1
#
```

4. **格式化**: flash_eraseall -j /dev/mtd1

格式化时用 “字符设备”

Flash_eraseall 格式 NAND 后，默认就是 yaffes 文件系统。

对于 NORFLASH，格式化时是 jffs2

```
# flash_eraseall --help
Usage: flash_eraseall [OPTION] MTD_DEVICE
Erases all of the specified MTD device.

  -j, --jffs2    format the device for jffs2
  -q, --quiet    don't display progress messages
      --silent   same as --quiet
      --help     display this help and exit
      --version  output version information and exit
#
```

5. mount -t jffs2 /dev/mtdblock1 /mnt

挂接时用 “块设备”。

在/mnt 目录下操作文件

```
# 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"
# ls /dev/mtd*
/dev/mtd0       /dev/mtd1       /dev/mtdblock0
/dev/mtd0ro     /dev/mtd1ro     /dev/mtdblock1
# mount -t jffs2 /dev/mtdblock1 /mnt
# cd /mnt
# ls
1.txt
# cat 1.txt
thisway.diy@163.com
17563039@qq.com
www.100ask.net
#
```

MTD 层不关心下层是 NAND 还是 NOR，只要有提供一个“mtd_info”结构到 MTD 层即可。那么可以用内存模拟一个 NOR。

用内存模拟 NOR：

linux-2.6.22.6\drivers\mtd\devices\Mtdram.c 直接分配和设置了一个 mtd_info 结构体再告诉 MTD 协议层

```
int __init init_mtdram(void)
-->mtd_info = kmalloc(sizeof(struct mtd_info), GFP_KERNEL); 分配 mtd_info  结构。
-->addr = vmalloc(MTDRAM_TOTAL_SIZE); 分配内存。
-->mtdram_init_device(mtd_info, addr, MTDRAM_TOTAL_SIZE, "mtdram test device");初始化
mtd_info
   -->mtd->erase = ram_erase; 直接提供了擦除函数。
      -->memset((char *)mtd->priv + instr->addr, 0xff, instr->len);擦除是直接memset为0xff。
   -->mtd->read = ram_read;
  -->memcpy(buf, mtd->priv + from, len); 直接拷贝。
   -->mtd->write = ram_write;
  -->memcpy((char *)mtd->priv + to, buf, len); 直接拷贝
   -->add_mtd_device(mtd)
```

NORFLASH 的识别过程：分析 do_mtd_probe()

```
s3c_nor_mtd = do_map_probe ("cfi_probe", s3c_nor_map); 假设以“cfi_probe”规范识别。
-->drv = get_mtd_chip_driver(name); 根据name=cfi_probe得到一个"mtd_chip_driver"结构。
  -->list_for_each(pos, &chip_drvs_list) 从这个chip_drvs_list链表中找。
-->ret = drv->probe(map); //probe函数原型：mtd_info *(*probe)(struct map_info *map);
```

查找这个probe()函数，搜索“chip_drvs_list”链表是在“register_mtd_chip_driver()”中，这个注册的mtd_chip_driver结构如下。

```
int __init cfi_probe_init(void)
-->struct mtd_chip_driver cfi_chipdrv = {
```

```
.probe          = cfi_probe,
.name                   = "cfi_probe",
.module          = THIS_MODULE
};
-->register_mtd_chip_driver(&cfi_chipdrv);
  -->list_add(&drv->list, &chip_drvs_list);
```

故而，ret = drv->probe(map) = cfi_probe(map).

```
struct mtd_info *cfi_probe(struct map_info *map)
-->mtd_do_chip_probe(map, &cfi_chip_probe);
  -->cfi = genprobe_ident_chips(map, cp); 通用的枚举
     -->genprobe_new_chip(map, cp, &cfi)
        -->cp->probe_chip(map, 0, NULL, cfi);
```

cp 即上面的结构“cfi_chip_probe”.进入“cif”规范模式：

```
static struct chip_probe cfi_chip_probe = {
     .name          = "CFI",
     .probe_chip = cfi_probe_chip
};
```

所以，cp->probe_chip(map, 0, NULL, cfi); 即为cfi_probe_chip(map,0,NULL,cfi).

```
cfi_probe_chip(struct map_info *map, __u32 base,unsigned long *chip_map, struct cfi_private
*cfi)
-->cfi_send_gen_cmd(0xF0, 0, base, map, cfi, cfi->device_type, NULL);   给0地址写0xF0,复位
   cfi_send_gen_cmd(0xFF, 0, base, map, cfi, cfi->device_type, NULL);   给0地址写0xFF
   cfi_send_gen_cmd(0x98, 0x55, base, map, cfi, cfi->device_type, NULL);给55地址写0x98
```

进入CFI模式　　往AAH写入98H　　mw.w aa 98

会根据位宽把 55 地址转成 AAH,因为是 2440 的 LADDR1 连 NOR 上的 A0 地址，故 NOR 上的 55≤1 即 AAH（左移即乘以 2）

```
-->qry_present(map,base,cfi) 看是否能读出"QRY"。
   -->
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); 读0x10地址是否得到‘Q’.存到val[0]中。
val[1] = map_read(map, base + osf*0x11); 读0x11地址是否得到‘R’.
val[2] = map_read(map, base + osf*0x12); 读0x12地址是否得到‘Y’.
-->xip_enable(base, map, cfi) 若上面读到的不是"Q""R""Y"，就退出。
-->cfi_chip_setup(map, cfi) 若是读到“Q”、“R”、“Y”，就读更多的信息。VID,PID,容量等。
int __xipram cfi_chip_setup(struct map_info *map,  struct cfi_private *cfi)
-->
cfi_send_gen_cmd(0xf0,     0, base, map, cfi, cfi->device_type, NULL);
cfi_send_gen_cmd(0xaa, 0x555, base, map, cfi, cfi->device_type, NULL);
cfi_send_gen_cmd(0x55, 0x2aa, base, map, cfi, cfi->device_type, NULL);
```

```
cfi_send_gen_cmd(0x90, 0x555, base, map, cfi, cfi->device_type, NULL);
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);
```

s3c_nor_mtd = do_map_probe ("jedec_probe", s3c_nor_map); 假设以“jedec_probe”规范识别。
根据名字“jedec_probe”找到一个“probe”.在“jedec_probe.c”中。

```
static struct mtd_chip_driver jedec_chipdrv = {
    .probe   = jedec_probe,
    .name    = "jedec_probe",
    .module  = THIS_MODULE
};
```

Do_map_probe()在“chipreg.c”中：

```
register_mtd_chip_driver(struct mtd_chip_driver *drv)
-->list_add(&drv->list, &chip_drvs_list); 添加进链表“chip_drvs_list”。
mtd_info *do_map_probe(const char *name, struct map_info *map)
-->drv = get_mtd_chip_driver(name);
    -->list_for_each(pos, &chip_drvs_list),
```

搜索这个“chip_drvs_list”链表。这里是扫描这个链表。上面有加入
register_mtd_chip_driver(struct mtd_chip_driver *drv)
所以可以搜索这个“register_mtd_chip_driver()”.

```
---- register_mtd_chip_driver Matches (16 in 7 files) ----
Cfi_probe.c (drivers\mtd\chips):    register_mtd_chip_driver(&cfi_chipdrv);
Jedec_probe.c (drivers\mtd\chips):  register_mtd_chip_driver(&jedec_chipdrv);
```

```
int __init jedec_probe_init(void)
-->register_mtd_chip_driver(&jedec_chipdrv);
```

```
static struct mtd_chip_driver jedec_chipdrv = {
    .probe   = jedec_probe,
    .name    = "jedec_probe",
    .module  = THIS_MODULE
};
```

```
struct mtd_info *jedec_probe(struct map_info *map)
-->mtd_do_chip_probe(map, &jedec_chip_probe);
```

```
static struct chip_probe jedec_chip_probe = {
    .name = "JEDEC",
    .probe_chip = jedec_probe_chip
};
```

```
    -->cfi = genprobe_ident_chips(map, cp); 通用的
        -->genprobe_new_chip(map, cp, &cfi)
            -->cp->probe_chip(map, 0, NULL, cfi), cp即&jedec_chip_probe


struct chip_probe jedec_chip_probe = {
.name = "JEDEC",
.probe_chip = jedec_probe_chip
```

```
};


int jedec_probe_chip(struct map_info *map, __u32 base, unsigned long *chip_map, struct cfi_private *cfi)
```

-->解锁地址

```
    cfi->addr_unlock1 = unlock_addrs[uaddr_idx].addr1;
    cfi->addr_unlock2 = unlock_addrs[uaddr_idx].addr2;
-->
if(cfi->addr_unlock1) {
cfi_send_gen_cmd(0xaa, cfi->addr_unlock1, base, map, cfi, cfi->device_type, NULL);
```

发命令：将解锁地址“cfi->addr_unlock1”放入地址“0xAA”.

```
cfi_send_gen_cmd(0x55, cfi->addr_unlock2, base, map, cfi, cfi->device_type, NULL);
```

发命令：将解锁地址“cfi->addr_unlock2”放入地址“0x55”.

}

往地址AAAH写AAH　　mw.w aaa aa
往地址554写55H　　mw.w 554 55
往地址AAAH写90H　　mw.w aaa 90

```
-->cfi_send_gen_cmd(0x90, cfi->addr_unlock1, base, map, cfi, cfi->device_type, NULL); 发读ID命令
-->cfi->mfr = jedec_read_mfr(map, base, cfi); 读到厂家ID
    cfi->id = jedec_read_id(map, base, cfi);    读到设备ID
-->jedec_match( base, map, cfi, &jedec_table[i] )  与组元为“amd_flash_info”结构的jedec_table[]数组比较
```

```
struct amd_flash_info {
    const __u16 mfr_id;
    const __u16 dev_id;
    const char *name;
    const int DevSize;
    const int NumEraseRegions;
    const int CmdSet;
    const __u8 uaddr[4];
    const ulong regions[6];
};
```


里面有“mfr_id - 厂家 ID”；“dev_id - 设备 ID”等。regions 是说擦除时以多大的块去擦除。

```
.regions    = {
    ERASEINFO(0x10000,31),
    ERASEINFO(0x08000,1),
    ERASEINFO(0x02000,2),
    ERASEINFO(0x04000,1)
```

如上面这个 NOR 是前面 31 个块是以“0x10000”块大小来擦除。后 1 个块以“0x8000”大小擦除。