基于Linux2.6.22.6内核，架构基于ARM

## 设备的分类

Linux的设备可分为以下4种分法：

1. 字符设备、块设备和网络设备
2. 混杂设备（misc device）
3. SPI、I2C、GPIO、LED、MTD等设备
4. 平台设备（platform device）

第一种分法是最基本的分法，字符设备是指可以按字节访问的设备，属性以“c”开头，比如“cwrx--r--r”，块设备是指读写以扇区为单位的设备，属性以“b”开头，比如“bwrx--r--r”，网络设备？

第二种分法，顾名思义，就是任何设备都可以归为混杂设备，混杂设备的主设备号固定为10，实现方法相对简单。不过混杂设备最终创建的是字符设备，这意味着所有混杂设备都是字符设备。

第三种分法是按照设备的功能来命名的，比如MTD设备，指一切闪存存储设备，比如NandFlash。Linux源码中提供了一套通用的操作NandFlash的函数，位于“linux-2.6.22.6/drivers/mtd/nand/nand\_base.c”。同理，SPI、LED等设备也是有一套通用的函数的。这类设备有些会以字符设备实现，有些是块设备，有些是网络设备，有些则是以上三种的组合。

第四种分法意思就是把设备要用到的所有资源（GPIO、中断、内存等）都统一用一个数据结构表达，再把设备的驱动函数用另一个数据结构表达。这样实现了资源和驱动的分离，有助于提高程序的模块化和可读性。

## 设备驱动的编译

设备驱动可以用以下两种方式编译，方法一需要手动加载，适用于驱动的研发阶段；方法二不需要手动加载，设备驱动如果调通了就可以改成编译进内核了。

方法一：编译成模块

方法二：编译进内核

## 内核的基本驱动

Linux里有着各个主流SOC厂商的BSP，大多数都写得有点乱，需要自己细致研究。

基本驱动代码的位置：

linux-2.6.22.6/arch/arm/mach-XXX/

linux-2.6.22.6/arch/arm/plat-XXX/

plat-XXX存放一个系列的SOC的公有驱动，mach-XXX则是具体某个SOC的自有驱动。也有些SOC没有plat-XXX这个目录的，它们把公有的和自有的驱动程序全部放在了mach-XXX里面。

一些常见驱动的位置：

NandFlash驱动：linux-2.6.22.6/drivers/mtd/nand/

SD卡驱动：linux-2.6.22.6/drivers/mmc/host/

网卡驱动：linux-2.6.22.6/drivers/net/

串口驱动：linux-2.6.22.6/drivers/serial/

SPI驱动：linux-2.6.22.6/drivers/spi/

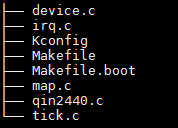
中断驱动：linux-2.6.22.6/include/asm-arm/arch-at91/entry-macro.S

## 移植Linux的基本要素

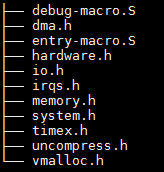
* 要有一个节拍定时器，一般是由SOC的某个硬件定时器来带动。
* 要有一个绑定为控制台的串口驱动。
* 要有一个Nandflash驱动。
* 要有一个MACHINE\_START结构体，它是一个SOC所有驱动程序的入口。
* 要有一个中断的注册程序，中断至少包含串口和节拍定时器中断。
* 要有一个SOC的寄存器地址的静态重映射程序，这个应该不是必须的，如果你用动态映射的话。
* 要有一个平台设备的注册程序，它要把串口跟Nandflash驱动注册进内核。

以上提到的所有代码都必须按照一定的文件结构放置，以下是一个最简单，最必须的文件结构（假设你的板子名叫qin2440）：

创建目录：linux-2.6.22.6/arch/arm/mach-qin2440，它的内容如下：



创建目录：linux-2.6.22.6/include/asm-arm/arch-qin2440，它的内容如下：



创建文件：linux-2.6.22.6/drivers/serial/qin2440\_serial.c。

创建文件：linux-2.6.22.6/drivers/mtd/nand/qin2440\_nand.c。

创建文件：linux-2.6.22.6/arch/arm/configs/qin2440\_defconfig。

下面逐个讲解各个文件：

* device.c 存放串口和Nandflash的平台设备注册程序
* irq.c 存放串口和节拍定时器的中断注册函数
* Kconfig qin2440的配置项，应该可以为空
* Makefile 控制哪些c文件需要被编译
* Makefile.boot 控制内核的链接地址
* map.c 存放qin2440的寄存器静态映射程序
* qin2440.c 存放MACHINE\_START结构体
* tick.c 存放节拍定时器的初始化程序
* debug-macro.S 存放打印串口的相关驱动。作用：在内核启动后，printk能用之前， 用这个临时的串口驱动打印一些信息。
* dma.h 存放DMA驱动，由于我只想写一个最小系统，所以可以不用DMA， 这里面只要写一个“#define MAX\_DMA\_ADDRESS 0xFFFFFFFF”即可。
* entry-macro.S 存放SOC的中断号获取函数。
* hardware.S 一般是存放SOC的寄存器地址。
* io.h 一些杂宏定义，参考代码便知。
* irqs.h SOC的各个中断号，以及中断的总数。
* memory.h 主要是规定了SOC的DRAM起始地址。
* system.h SOC的空闲和复位函数。
* timex.h SOC的节拍定时器的时基频率。
* uncompress.h 内核在解压的时候的串口驱动，此时内核还没跑起来，MMU也没建 立起映射，所以这里的串口驱动的地址都是真实物理地址。
* vmalloc.h 内核的内存分配的最大地址，这个地址要尽可能的大。一般来说，一 个32bit的SOC，4GB的虚拟空间，最尾部的空间存放SOC的寄存器 的静态虚拟映射，再往前就是这个文件内的宏VMALLOC\_END所规定 的内存分配最大地址。
* qin2440\_serial.c qin2440的串口驱动代码，请看后面的“串口设备”一节。
* qin2440\_nand.c qin2440的Nandflash驱动代码，请看后面的“NAND设备”一节。
* qin2440\_defconfig qin2440的配置文件。

补充：

实际上，你还得改内核里的一些Makefile和Kconfig文件来让内核“接纳”你新加入的代码。具体请看我写的代码。

### 核心资源注册入口

每个SOC的驱动都有一个入口点，这个入口点便是“MACHINE\_START”。

位于：linux-2.6.22.6/arch/arm/mach-XXX/<MACHINE>.c

MACHINE\_START(S3C2440, "SMDK2440")

.boot\_params = S3C2410\_SDRAM\_PA + 0x100,

.init\_irq = s3c24xx\_init\_irq,

.map\_io = smdk2440\_map\_io,

.init\_machine = smdk2440\_machine\_init,

.timer = &s3c24xx\_timer,

MACHINE\_END

* MACHINE\_START()：用来初始化一个结构体。它有两个形参，第一个形参展开后就是你的板子的机器码，这个机器码必须是linux-2.6.22.6/include/asm-arm/mach-types.h（由linux-2.6.22.6/arch/arm/tools/mach-types生成）中所有机器码的一个，否则编译不能通过。MACHINE\_START()需要和MACHINE\_END配对使用。
* .boot\_params：存放内核启动参数的位置。实际上，u-boot也会通过ATAG告知内核启动参数放在哪，这个变量是当u-boot没告知的时候，Linux自己去寻找的地址。
* .map\_io：建立起板子的地址映射，一般是SOC的寄存器地址的静态映射；有些板子还会在这个函数里做其他的初始化工作。
* .timer：节拍定时器初始化，大致流程是初始化一个硬件定时器，使它周期性地调用内核里的timer\_tick函数。在更高版本的内核里，定时器的周期也不一定是写死的，内核的节拍时钟可以配置成tickless模式，即定时器周期不固定，每次周期都是内核自己算出来的。
* .init\_irq：注册SOC的各种中断，每个中断都有自己的中断号、中断标志、中断处理函数。

Note：

关于开发板的机器码：Linux为很多SOC厂商的开发板分配了机器码。开发者必须确保u-boot传进来的机器码和内核自己的相等，否则内核不能启动。内核的机器码就由MACHINE\_START的第一个参数决定，

### 建立映射关系

Linux如果使能了MMU（绝大多数时候都是使能的），寄存器就不能直接访问物理地址了，要把物理地址映射成虚拟地址后去访问虚拟地址。

寄存器的映射有两种方式，一种静态映射一种动态映射，假设要访问的寄存器物理地址为0x80000008。

#### 静态映射

先建立一个map\_desc结构体。

.virtual 寄存器的虚拟地址，根据需要自己填写

.pfn 寄存器的物理地址，必须要用\_\_phys\_to\_pfn()包裹住

.length 地址映射的长度

.type 固定为MT\_DEVICE

static struct map\_desc smdk2440\_iodesc[] \_\_initdata = {

{

.virtual = 0x23000000,

.pfn = \_\_phys\_to\_pfn(0x80000008),

.length = 4,

.type = MT\_DEVICE,

}, {

...

...

}

};

然后用iotable\_init建立起映射即可。

iotable\_init(smdk2440\_iodesc, ARRAY\_SIZE(smdk2440\_iodesc));

映射成功后，0x23000000就是0x80000008映射后的地址，读写都是操作0x23000000这个地址。

#### 静态映射的限制

一个32位的SOC有4GB的虚拟地址空间，但你的静态映射是不能随便指定的，因为在这4GB的存储空间中，还有别的区域划分，你不能让寄存器的静态映射空间跟别的空间重叠。这“别的空间”一般就是vmalloc。

在linux-2.6.22.6/include/asm-arm/arch-XXX/vmalloc.h中有一个宏定义：

#define VMALLOC\_END 0xF6000000

这就是vmalloc的结束地址，至于vmalloc的起始地址，用户是不能指定的（至少我现在不知道）。为了不让寄存器的静态映射空间跟它重叠，一般的做法是把寄存器的静态映射空间全部放到这个VMALLOC\_END的后面。

#### 动态映射

先定义一个变量。

void \_\_iomem \*reg;

调用ioremap函数，“4”的意思是这个地址往后4个字节，根据自己需要填写。映射成功后就可以用writel和readl等函数操作reg变量了。

reg = ioremap(0x80000008, 4);

取消映射调用iounmap。

iounmap(0x80000008);

有的时候，在调用ioremap之前，会先调用request\_mem\_region，目的是为了使申请到的动态映射内存不能再被ioremap。

#### 寄存器的读写

后缀l long型，即4字节

后缀b char型，即1字节

后缀w short型，即2字节

#define readb(c)

#define readw(c)

#define readl(c)

#define writeb(v,c)

#define writew(v,c)

#define writel(v,c)

#define \_\_raw\_readb(c)

#define \_\_raw\_readw(c)

#define \_\_raw\_readl(c)

#define \_\_raw\_writeb(v,c)

#define \_\_raw\_writew(v,c)

#define \_\_raw\_writel(v,c)

#define ioread8(p)

#define ioread16(p)

#define ioread32(p)

#define iowrite8(v,p)

#define iowrite16(v,p)

#define iowrite32(v,p)

位于linux-2.6.22.6/include/asm/io.h。

形参v是数据；p、c是地址。

### 注册中断

ARM的中断向量表位于：linux-2.6.22.6/arch/arm/kernel/entry-armv.S。中断的处理函数中，中断号的获取跟硬件相关，获取函数为：get\_irqnr\_and\_base。位于：linux-2.6.22.6/include/asm-arm/arch-XXX/entry-macro.S。这个函数需要用户自己实现，大体思路是通过读取中断挂起寄存器以获得中断号，内核知道了中断号才能调用相关的中断处理程序。它的原型如下：

.macro get\_irqnr\_and\_base, irqnr, irqstat, base, tmp

…

.endm

可见，它是一个宏汇编函数。注意它的名字、形参的个数和顺序都是不能改的，你只能改它的实现部分。

它总共有4个形参：

* Irqnr：中断号。这个函数的任务就是算出中断号，所以务必确保这个函数退出时irqnr里面必须存放着当前发生的中断号，其它形参无所谓。
* Irqstat：内核提供的一个临时变量，让你可以存储中断寄存器的状态，当然你用作其它用途也是可以的。
* base：硬件寄存器的基地址，这个基地址加上你的SOC的中断寄存器的相对偏移就是你的中断寄存器的访问地址，用作其他用途也可以。
* tmp：一个临时变量，随便用。

中断的注册函数：

int set\_irq\_chip(unsigned int irq, structirq\_chip \*chip);

void set\_irq\_handler(unsigned intirq, irq\_flow\_handler\_t handle);

上面这两个函数可以注册一个中断，irq和intirq都是中断号，handle是中断处理函数，只能用Linux提供的这两个函数：

* handle\_edge\_irq：允许中断嵌套。
* handle\_level\_irq：不允许中断嵌套。

用户应当根据自己的需要决定使用哪个中断处理函数。

chip的原型为：

struct irq\_chip {

const char \*name;

unsigned int (\*startup)(unsigned int irq);

void (\*shutdown)(unsigned int irq);

void (\*enable)(unsigned int irq);

void (\*disable)(unsigned int irq);

void (\*ack)(unsigned int irq);

void (\*mask)(unsigned int irq);

void (\*mask\_ack)(unsigned int irq);

void (\*unmask)(unsigned int irq);

void (\*eoi)(unsigned int irq);

void (\*end)(unsigned int irq);

void (\*set\_affinity)(unsigned int irq, cpumask\_t dest);

int (\*retrigger)(unsigned int irq);

int (\*set\_type)(unsigned int irq, unsigned int flow\_type);

int (\*set\_wake)(unsigned int irq, unsigned int on);

/\* Currently used only by UML, might disappear one day.\*/

#ifdef CONFIG\_IRQ\_RELEASE\_METHOD

void (\*release)(unsigned int irq, void \*dev\_id);

#endif

/\*

\* For compatibility, ->typename is copied into ->name.

\* Will disappear.

\*/

const char \*typename;

};

位于：linux-2.6.22.6/include/linux/irq.h。

常用的成员有：

* name：中断的名字，必填。
* mask：中断的屏蔽函数，必填。
* unmask：中断的解屏蔽函数，必填。
* ack：中断的清除标志位函数，选填。

也可以用下面这个函数同时完成上面两个函数的工作：

void set\_irq\_chip\_and\_handler(unsigned int irq, struct irq\_chip \*chip,

irq\_flow\_handler\_t handle);

位于linux-2.6.22.6/include/linux/irq.h。

各个形参都跟上面两个相对应。

最后设置中断的标志位即可：

void set\_irq\_flags(unsigned int irq, unsigned int iflags);

位于linux-2.6.22.6/include/asm/mach/irq.h。

flags可取的值有：

#define IRQF\_VALID (1 << 0)

#define IRQF\_PROBE (1 << 1)

#define IRQF\_NOAUTOEN (1 << 2)

在一个SOC中，有些中断会比较特别，就拿s3c2440来说，它的外部中断EINT4~EINT7是共享一个中断号4的。按照常规思路，当4号中断发生时，内核并不知道EINT4~EINT7到底是哪个发生了中断。为了解决这个问题，Linux引入了父中断（parent irq）和子中断（child irq）这两个概念，在s3c2440这个例子中，可以把中断号4规定为父中断，它是SOC真正能产生的中断号，规定EINT4~EINT7的中断号为51~54，它们是4的子中断，是用户自己规定的中断号，SOC是不能产生的。与之相关的函数为set\_irq\_chained\_handler：

static inline void

set\_irq\_chained\_handler(unsigned int irq, irq\_flow\_handler\_t handle);

它的形参中，irq就是中断号；handle是中断处理函数，由用户实现，注意这个中断处理函数可不是request\_irq里的那个中断处理函数，这个中断处理函数不是让你去处理业务逻辑的，而是让你告知内核子中断号到底是多少。在s3c2440这个例子中，这个函数的irq等于4，handle会读取EINTPEND寄存器从而获得子中断号，假设为n，n=[51,54]，最后调用desc\_handle\_irq(n, desc)来触发相应的子中断，实参desc又是一个知识点，它是handle的一个形参，用户要让它指向一个全局结构体数组的第irq个成员，这个全局结构体数组便是irq\_desc，它用每个元素来描述你的每个中断。你在linux-2.6.22.6/include/asm/arch-XXX/irqs.h中定义的宏NR\_IRQS（用户的SOC的中断总数，这个宏必须存在）就是irq\_desc的成员个数。

大致的代码如下：

static void trigger\_child\_irq(unsigned int irq, struct irq\_desc \*desc)

{

// 这步是不对的，我只是想表达一个大致的意思

unsigned int child\_irq = \_\_raw\_readl(EINTPEND);

desc = irq\_desc + child\_irq;

desc\_handle\_irq(child\_irq, desc);

}

在此做个总结，假设中断号3是一个普通中断（没有子中断的父中断），4是5的父中断，5是4的子中断，下面的表格列出了这3个中断的注册流程：

|  |  |  |
| --- | --- | --- |
| 中断号3 | 中断号4 | 中断号5 |
| set\_irq\_chip\_and\_handler | set\_irq\_chip | set\_irq\_chip\_and\_handler |
| set\_irq\_flags | set\_irq\_chained\_handler | set\_irq\_flags |

Note：

* 父中断一旦被设置成功，就是不可申请的，永远使能的。
* 父中断和子中断要分别有自己的irq\_chip结构体。

中断的注册完成了以后，后续的驱动就可以申请中断了，申请中断使用下面这个函数：

int request\_irq(unsigned int irq, irq\_handler\_t handler,

unsigned long irqflag, const char \*devname, void \*dev\_id);

它位于linux-2.6.22.6/include/linux/interrupt.h。

各形参解释：

* irq：中断号。
* handler：中断处理函数，处理用户的业务逻辑。原型如下：

irqreturn\_t xxx(int irq, void \*dev\_id);

* irqflags：可取的值如下，下面这些宏位于linux-2.6.22.6/include/linux/interrupt.h。

#define IRQF\_DISABLED 0x00000020

#define IRQF\_SAMPLE\_RANDOM 0x00000040

#define IRQF\_SHARED 0x00000080

#define IRQF\_PROBE\_SHARED 0x00000100

#define IRQF\_TIMER 0x00000200

#define IRQF\_PERCPU 0x00000400

#define IRQF\_NOBALANCING 0x00000800

#define IRQF\_IRQPOLL 0x00001000

* devname：中断的名字，只要不重复就行。
* dev\_id：用来传给中断处理函数handler的dev\_id的。

### 注册DMA

？

### 初始化节拍定时器

大致代码如下：

static irqreturn\_t timer\_interrupt(int irq, void \*dev\_id)

{

timer\_tick();

return IRQ\_HANDLED;

}

static struct irqaction timer\_irq = {

.name = "Timer Tick",

.flags = IRQF\_DISABLED | IRQF\_TIMER,

.handler = timer\_interrupt,

};

void \_\_init timer\_init(void)

{

setup\_irq(IRQ\_TIMER2, &timer\_irq);

/\* 启动硬件定时器 \*/

...

}

static struct sys\_timer tick\_timer = {

.init = timer\_init,

};

MACHINE\_START(XXX, "XXX")

...

.timer = &tick\_timer,

...

MACHINE\_END

可以看出，这段代码最主要的作用是申请了一个中断号为IRQ\_TIMER2的中断，然后让这个中断的处理函数执行timer\_tick函数，而timer\_tick函数正是内核的时钟节拍函数。

这个节拍定时器的中断频率由3个宏来控制：

* HZ：内核的节拍频率，它的取值是受Linux限制的，常见的取值有100、250、1000。
* CLOCK\_TICK\_RATE：硬件定时器的时钟频率（假如节拍定时器的时基为1ms，那么这个宏等于1000000）。位于linux-2.6.22.6/include/asm/arch-XXX/timex.h。
* LATCH：SOC硬件定时器的节拍数，可以直接把这个宏赋给硬件定时器的节拍寄存器（可能要减去1），使硬件定时器每经过LATCH个节拍就中断一次。位于linux-2.6.22.6/include/linux/jiffies.h。

它们之间的关系如下：

1/HZ = (1/CLOCK\_TICK\_RATE) \* LATCH

意思是内核需要硬件定时器每1/HZ秒产生一次中断，而硬件定时器每个节拍的时长为1/CLOCK\_TICK\_RATE秒，总共需要LATCH个节拍以满足内核的要求。

在内核中，它们的定义用了个小窍门：HZ/2

#define LATCH ((CLOCK\_TICK\_RATE + HZ/2) / HZ)

本质上，应该是LATCH=CLOCK\_TICK\_RATE/HZ，让CLOCK\_TICK\_RATE先加HZ/2再除以HZ是为了提高除法的精度。

注意，HZ和CLOCK\_TICK\_RATE两个宏是可以用make menuconfig来配置的。

### 初始化打印串口

见“串口设备”一章。

## 平台设备

Linux源码中绝大部分的设备驱动都是用平台设备驱动实现的，而且以后的设备驱动编写也应该尽量用平台设备实现。

设备的硬件资源包括：中断、时钟、DMA、GPIO、内存等，平台设备使用resource结构体来描述这些资源；platform\_device结构体用来描述一个设备，resource最终会被包含在platform\_device中，而platform\_device会被platform\_device\_register函数注册进内核。

上面的描述只是把一个设备的资源注册到了内核，它还需要驱动函数，驱动函数使用platform\_driver结构体来描述。platform\_driver提供了设备的探测、挂起、恢复等操作函数，其中最重要的就是探测函数­­——probe，探测函数所做的工作包括创建字符设备、块设备、网络设备以及申请中断、申请时钟等，以及提供设备的最基本的读写、初始化操作。platform\_driver最终由platform\_driver\_register函数注册进内核。

platform\_device中有一个.dev.platform\_data指针，它是一个无类型指针，专门用来指向用户私有数据的。实际上，Linux驱动中的很多数据结构都使用了私有数据这种东西，它的主要作用有两个：

* 可以提供更多有关驱动的信息。
* 使不同的驱动函数可以互通数据。诚然，使用全局变量可以互通数据，但在Linux驱动中并不推荐这种做法，因为全局变量无法释放。比如：用platform\_set\_drvdata把私有数据跟device结构体关联后，别的驱动函数可以用platform\_get\_drvdata函数获取私有数据，从而实现数据的互通。

platform\_device和platform\_driver两个结构体是靠各自的名字来关联的，即：platform\_device.name和platform\_driver.driver.name。只要名字一样就能关联上。platform\_device被注册了多少个，platform\_drive.probe函数就会被执行多少次，platform\_drive.probe函数根据platform\_device.id来区分各个设备，platform\_device.id并不会被内核使用，这个变量只是为了方便用户区分不同的设备。

## 混杂设备

## 字符设备

## 块设备

## 串口设备

串口设备驱动的代码位于: linux-2.6.22.6/drivers/serial。

串口设备的platform\_device结构体的初始化及注册略过。

下面这段代码就是串口驱动的主体架构，关键元素如下：

* uart\_port结构体（提供串口的各项属性和操作函数）
* uart\_driver结构体（提供串口的字符设备名称、主次设备号等）

static struct mcf\_uart mcf\_ports;

static struct uart\_driver mcf\_driver = {

.owner = THIS\_MODULE,

.driver\_name = "mcf",

.dev\_name = "ttyS",

.major = TTY\_MAJOR,

.minor = 64,

.nr = MCF\_MAXPORTS,

.cons = MCF\_CONSOLE,

};

static int \_\_devexit mcf\_remove(struct platform\_device \*pdev)

{

struct uart\_port \*port;

port = &mcf\_ports.port;

uart\_remove\_one\_port(&mcf\_driver, port);

return 0;

}

static int \_\_devinit qin2440\_probe(struct platform\_device \*pdev)

{

struct mcf\_platform\_uart \*platp = pdev->dev.platform\_data;

struct uart\_port \*port;

int i;

port = &mcf\_ports.port;

port->line = 0;

port->type = PORT\_MCF;

port->mapbase = platp[i].mapbase;

port->membase = (platp[i].membase) ? platp[i].membase :

(unsigned char \_\_iomem \*) platp[i].mapbase;

port->iotype = SERIAL\_IO\_MEM;

port->irq = platp[i].irq;

port->uartclk = MCF\_BUSCLK;

port->ops = &mcf\_uart\_ops;

port->flags = ASYNC\_BOOT\_AUTOCONF;

uart\_add\_one\_port(&mcf\_driver, port);

return 0;

}

static struct platform\_driver qin2440\_platform\_driver = {

.probe = qin2440\_probe,

.remove = \_\_devexit\_p(qin2440\_remove),

.driver = {

.name = "qin2440\_uart",

.owner = THIS\_MODULE,

},

};

static int \_\_init qin2440\_init(void)

{

uart\_register\_driver(&qin2440\_driver);

platform\_driver\_register(&qin2440\_platform\_driver);

return 0;

}

static void \_\_exit qin2440\_exit(void)

{

platform\_driver\_unregister(&qin2440\_platform\_driver);

uart\_unregister\_driver(&qin2440\_driver);

}

代码不是很复杂，用platform\_driver的probe函数来注册上文提到的两个关键结构体。uart\_driver结构体很方便，你可以把串口字符设备的主次设备号信息放在里面，uart\_register\_driver函数会自动创建相应的字符设备。绝大多数的SOC的串口都带有FIFO，这个FIFO相当于一个专属DMA，所以串口驱动程序一般不用写DMA。port->mapbase和port->membase，前者是真实物理寄存器地址，后者是前者映射后的虚拟地址。port->ignore\_status\_mask这个成员大概就是忽略串口接收状态寄存器的某些bit的意思，Linux中关于它写得很绕，没看懂，一般设为0就行了。

接下来讲解port结构体的ops成员，它包含了串口的底层操作函数，以及串口的中断处理函数。

port结构体的ops成员被这个结构体赋值：

static const struct uart\_ops mcf\_uart\_ops = {

.tx\_empty = mcf\_tx\_empty,

.get\_mctrl = mcf\_get\_mctrl,

.set\_mctrl = mcf\_set\_mctrl,

.start\_tx = mcf\_start\_tx,

.stop\_tx = mcf\_stop\_tx,

.stop\_rx = mcf\_stop\_rx,

.enable\_ms = mcf\_enable\_ms,

.break\_ctl = mcf\_break\_ctl,

.startup = mcf\_startup,

.shutdown = mcf\_shutdown,

.set\_termios = mcf\_set\_termios,

.type = mcf\_type,

.request\_port = mcf\_request\_port,

.release\_port = mcf\_release\_port,

.config\_port = mcf\_config\_port,

.verify\_port = mcf\_verify\_port,

};

成员很多，挑重点讲解：

* config\_port、request\_port、startup都是用来初始化串口硬件的。
* set\_termios用来设置串口的熟悉，比如波特率、数据位等。
* start\_tx、stop\_tx、stop\_rx通常只需使能或禁能tx、rx中断即可。
* tx\_empty用来判断串口的硬件发送FIFO是否为空。
* break\_ctl用来发送一个break信号。

set\_termios比较复杂，它的原型如下：

void (\*set\_termios)(struct uart\_port \*, struct ktermios \*new,

struct ktermios \*old);

Linux有一套操作串口的用户层API，用户用这些API对串口所做的配置最终会送到set\_termios。set\_termios会解析用户的配置，这些配置我只挑了下面几个比较难的讲解：

① baud = uart\_get\_baud\_rate(port, termios, old, 0, port->uartclk/16);

② quot = uart\_get\_divisor(port, baud);

③ uart\_update\_timeout(port, termios->c\_cflag, baud);

1. 获得波特率，0和port->uartclk/16分别指的是最小最大波特率。
2. 获得分频值，似乎所有的SOC的串口分频值都等于串口时钟/波特率/16。
3. 设置串口超时，基本是固定格式。

中断处理函数如下：

static void mcf\_rx\_chars(struct mcf\_uart \*pp)

{

struct uart\_port \*port = &pp->port;

unsigned char status, ch, flag;

while ((status = readb(port->membase + MCFUART\_USR)) & MCFUART\_USR\_RXREADY) {

ch = readb(port->membase + MCFUART\_URB);

flag = TTY\_NORMAL;

port->icount.rx++;

if (status & MCFUART\_USR\_RXERR) {

writeb(MCFUART\_UCR\_CMDRESETERR,

port->membase + MCFUART\_UCR);

if (status & MCFUART\_USR\_RXBREAK) {

port->icount.brk++;

if (uart\_handle\_break(port))

continue;

} else if (status & MCFUART\_USR\_RXPARITY) {

port->icount.parity++;

} else if (status & MCFUART\_USR\_RXOVERRUN) {

port->icount.overrun++;

} else if (status & MCFUART\_USR\_RXFRAMING) {

port->icount.frame++;

}

status &= port->read\_status\_mask;

if (status & MCFUART\_USR\_RXBREAK)

flag = TTY\_BREAK;

else if (status & MCFUART\_USR\_RXPARITY)

flag = TTY\_PARITY;

else if (status & MCFUART\_USR\_RXFRAMING)

flag = TTY\_FRAME;

}

if (uart\_handle\_sysrq\_char(port, ch))

continue;

uart\_insert\_char(port, status, MCFUART\_USR\_RXOVERRUN, ch, flag);

}

tty\_flip\_buffer\_push(port->state->port.tty);

}

static void mcf\_tx\_chars(struct mcf\_uart \*pp)

{

struct uart\_port \*port = &pp->port;

struct circ\_buf \*xmit = &port->state->xmit;

if (port->x\_char) {

writeb(port->x\_char, port->membase + MCFUART\_UTB);

port->x\_char = 0;

port->icount.tx++;

return;

}

while (readb(port->membase + MCFUART\_USR) & MCFUART\_USR\_TXREADY) {

if (xmit->head == xmit->tail)

break;

writeb(xmit->buf[xmit->tail], port->membase + MCFUART\_UTB);

xmit->tail = (xmit->tail + 1) & (UART\_XMIT\_SIZE -1);

port->icount.tx++;

}

if (uart\_circ\_chars\_pending(xmit) < WAKEUP\_CHARS)

uart\_write\_wakeup(port);

if (xmit->head == xmit->tail) {

pp->imr &= ~MCFUART\_UIR\_TXREADY;

writeb(pp->imr, port->membase + MCFUART\_UIMR);

}

}

static irqreturn\_t mcf\_interrupt(int irq, void \*data)

{

struct uart\_port \*port = data;

struct mcf\_uart \*pp = container\_of(port, struct mcf\_uart, port);

unsigned int isr;

irqreturn\_t ret = IRQ\_NONE;

isr = readb(port->membase + MCFUART\_UISR) & pp->imr;

spin\_lock(&port->lock);

if (isr & MCFUART\_UIR\_RXREADY) {

mcf\_rx\_chars(pp);

ret = IRQ\_HANDLED;

}

if (isr & MCFUART\_UIR\_TXREADY) {

mcf\_tx\_chars(pp);

ret = IRQ\_HANDLED;

}

spin\_unlock(&port->lock);

return ret;

}

mcf\_interrupt通过读取中断挂起寄存器来获知当前中断是发送还是接收中断，之后便执行发送或接收函数。

发送函数mcf\_tx\_chars通过if(port->x\_char)语句判断是不是只要发送单个字符，如果是，发送这个字符然后退出。如果不是，用一个while循环把发送缓冲中的所有字符都发出去。注意这两条语句是固定格式：

if (uart\_circ\_chars\_pending(xmit) < WAKEUP\_CHARS)

uart\_write\_wakeup(port);

Question：

既然发送函数位于中断函数中，这意味着内核必须触发串口发送中断来发送数据，而串口发送中断必须要发送字节才能触发，那么内核是怎样触发串口发送中断的呢？

Answer：

？

接收函数mcf\_rx\_chars用一个while循环不停地判断有没有接收到字符，如果有，接收这个字符，并发送给内核：

if (uart\_handle\_sysrq\_char(port, ch))

continue;

uart\_insert\_char(port, status, MCFUART\_USR\_RXOVERRUN, ch, flag);

while循环中还做了另外一件事情：统计接收状况。

根据接收状态寄存器来使port->icount.brk、port->icount.parity、port->icount.overrun、port->icount.frame这些变量自加。flag局部变量也要根据状态寄存器来置位，可取的值有：TTY\_BREAK、TTY\_PARITY、TTY\_FRAME，flag会被后面的uart\_insert\_char用到。接收函数的最后一句是固定格式，如下：

tty\_flip\_buffer\_push(port->state->port.tty);

如果要把一个串口设置成内核的打印串口，要怎么整呢？

关键要有个console结构体和一个console\_init函数，console结构体如下：

static struct console mcf\_console = {

.name = "ttyS",

.write = mcf\_console\_write,

.device = uart\_console\_device,

.setup = mcf\_console\_setup,

.flags = CON\_PRINTBUFFER,

.index = 0,

.data = &mcf\_driver,

};

各成员解释：

* name：终端的名字前缀
* write：终端的字符发送函数，不复杂，代码不贴出了
* device：等于uart\_console\_device，这是固定格式，无需关心
* setup：终端的串口初始化函数，主要是设置串口的波特率、数据位等基本属性，不复杂，代码不贴出了
* flags ：？
* index：指定哪个串口设备作为终端串口，这里原本是-1，seteup函数会把它置为0
* data：就是前文提到也贴出代码的那个包含有串口的主次设备号的mcf\_driver结构体的地址，实际上，mcf\_driver的.cons成员存放的也是这个console结构体的地址

console\_init函数如下（以QIN2440为例）：

static int \_\_init console\_init(void)

{

int i;

struct uart\_port \*port;

for(i = 0; i < QIN2440\_TOTAL\_PORTS; i++) {

port = &qin2440\_ports[i].port;

qin2440\_port\_init(port, i);

}

register\_console(&qin2440\_console);

return 0;

}

console\_initcall(console\_init);

console\_init做了两件事：

* 填充port结构体。这一步跟上文提到的mcf\_probe中的port结构体初始化是完全一样的，可以统一用一个函数来做。跟mcf\_probe不同的是，这一步填充完port结构体后并不会调用uart\_add\_one\_port。
* 调用register\_console注册console结构体。

最后千万别忘了用console\_initcall把console\_init加入Linux启动执行列表中。

## SPI设备

SPI设备驱动的关键是spi\_master结构体、中断和传输函数。

spi\_master结构体包含了SPI外设的所有硬件配置信息和底层操作函数（包括传输函数）。SPI的数据传输是一个难点：spi\_master的transfer函数接收一个spi\_message参数，后者以链表的形式提供SPI待传输的数据。transfer函数提取出spi\_message中的一个一个的节点然后调用硬件进行传输，传输完成会触发中断，在中断回调函数中调用spi\_message中的complete函数通知上层应用SPI的数据传输已完成。

SPI设备驱动分为4个部分（开发板以yl-9200为例）：

|  |  |  |
| --- | --- | --- |
| 序号 | 功能 | 位置 |
| 1 | 描述SPI外设资源情况 | linux-2.6.39\arch\arm\mach-at91\board-yl-9200.c |
| 2 | 提供SPI外设的操作函数 | linux-2.6.39\drivers\spi\atmel\_spi.c |
| 3 | 提供使用SPI外设的板上设备的操作函数 | linux-2.6.39\drivers\mtd\devices\m25p80.c |
| 4 | 描述第3部分的设备如何占用SPI外设 | linux-2.6.39\arch\arm\mach-at91\board-yl-9200.c |

### 描述SPI外设的资源

用一个resource结构体和一个platform\_device结构体描述了芯片的SPI外设的寄存器起止地址、中断号、名字，用platform\_device\_register函数把platform\_device结构体注册进内核。

static struct resource spi\_resources[] = {

[0] = {

.start = AT91RM9200\_BASE\_SPI,

.end = AT91RM9200\_BASE\_SPI + SZ\_16K - 1,

.flags = IORESOURCE\_MEM,

},

[1] = {

.start = AT91RM9200\_ID\_SPI,

.end = AT91RM9200\_ID\_SPI,

.flags = IORESOURCE\_IRQ,

},

};

static struct platform\_device at91rm9200\_spi\_device = {

.name = "atmel\_spi",

.id = 0,

.resource = spi\_resources,

.num\_resources = ARRAY\_SIZE(spi\_resources),

};

platform\_device\_register(&at91rm9200\_spi\_device);

### 注册spi\_master结构体

SPI的prob函数有些复杂，但总体思路很简单：先分配一个spi\_master结构体，再把SPI外设的操作函数挂上去，最后注册之。代码大体如下：

static int \_\_init atmel\_spi\_probe(struct platform\_device \*pdev)

{

struct resource \*regs;

int irq;

struct clk \*clk;

int ret;

struct spi\_master \*master;

struct atmel\_spi \*as;

① regs = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0);

irq = platform\_get\_irq(pdev, 0);

clk = clk\_get(&pdev->dev, "spi\_clk");

② master = spi\_alloc\_master(&pdev->dev, sizeof \*as);

master->mode\_bits = SPI\_CPOL | SPI\_CPHA | SPI\_CS\_HIGH;

master->bus\_num = pdev->id;

master->num\_chipselect = 4;

master->setup = atmel\_spi\_setup;

master->transfer = atmel\_spi\_transfer;

master->cleanup = atmel\_spi\_cleanup;

platform\_set\_drvdata(pdev, master);

③ as = spi\_master\_get\_devdata(master);

as->buffer = dma\_alloc\_coherent(&pdev->dev, BUFFER\_SIZE,

&as->buffer\_dma, GFP\_KERNEL);

spin\_lock\_init(&as->lock);

INIT\_LIST\_HEAD(&as->queue);

as->pdev = pdev;

as->regs = ioremap(regs->start, resource\_size(regs));

as->irq = irq;

as->clk = clk;

④ ret = request\_irq(irq, atmel\_spi\_interrupt, 0,

dev\_name(&pdev->dev), master);

clk\_enable(clk);

...

⑤ ret = spi\_register\_master(master);

return 0;

}

1. 使用platform\_get\_resource函数获取SPI外设的寄存器资源，platform\_get\_irq也是platform\_get\_resource的封装。
2. 分配一个spi\_master结构体，填充完后把它设为device的私有数据。注意这里规定了很多SPI的属性，比如SPI的外设总数为4（master->num\_chipselect = 4）。spi\_master最重要的成员函数就是transfer，它负责spi外设数据的收发。
3. spi\_master也有自己的私有数据，把它提取出来并填充更多的数据，作为一个补充。
4. 软件的数据都准备好了，这里是初始化硬件SPI，包括初始化寄存器、使能时钟、申请中断。
5. 注册spi\_master。

spi\_master由atmel\_spi\_probe注册，而后者又由platform\_driver\_probe注册进内核，注意这里的写法，probe函数一般是放在platform\_driver结构体里面再由platform\_driver\_register注册的，这样写的驱动可以热插拔，而下面这段代码的做法使得SPI设备不支持热插拔。atmel\_spi\_remove函数的具体内容是把前面申请到的各种资源都释放掉，具体不贴出来了。注意：时钟释放之前要先禁能；如果SPI还有未完成的传输队列就停止传输。

static int \_\_exit atmel\_spi\_remove(struct platform\_device \*pdev)

{

...

}

static struct platform\_driver atmel\_spi\_driver = {

.driver = {

.name = "atmel\_spi",

.owner = THIS\_MODULE,

},

.remove = \_\_exit\_p(atmel\_spi\_remove),

};

static int \_\_init atmel\_spi\_init(void)

{

return platform\_driver\_probe(&atmel\_spi\_driver, atmel\_spi\_probe);

}

static void \_\_exit atmel\_spi\_exit(void)

{

platform\_driver\_unregister(&atmel\_spi\_driver);

}

module\_init(atmel\_spi\_init);

module\_exit(atmel\_spi\_exit);

SPI的传输函数大体如下：

int atmel\_spi\_transfer(struct spi\_device \*spi, struct spi\_message \*msg)

{

struct atmel\_spi \*as;

struct spi\_transfer \*xfer;

unsigned long flags;

struct device \*controller = spi->master->dev.parent;

u8 bits;

struct atmel\_spi\_device \*asd;

① as = spi\_master\_get\_devdata(spi->master);

② list\_for\_each\_entry(xfer, &msg->transfers, transfer\_list) {

if (!msg->is\_dma\_mapped) {

if (atmel\_spi\_dma\_map\_xfer(as, xfer) < 0)

return -ENOMEM;

}

}

③ msg->status = -EINPROGRESS;

msg->actual\_length = 0;

spin\_lock\_irqsave(&as->lock, flags);

④ list\_add\_tail(&msg->queue, &as->queue);

atmel\_spi\_next\_message(spi->master);

spin\_unlock\_irqrestore(&as->lock, flags);

return 0;

}

1. 从spi\_master结构体中提取出私有数据。
2. 用一个spi\_transfer指针依次指遍spi\_message里的各个节点，对于每个节点，调用atmel\_spi\_dma\_map\_xfer函数给spi\_transfer指针里的tx\_dma和rx\_dma指定DMA传输地址，为之后的DMA传输做好准备。
3. spi\_message在正式传输前先准备一些成员变量。
4. 把spi\_message结构体加入spi\_master的私有数据的链表中，调用atmel\_spi\_next\_message启动SPI传输，当然前后是需要自旋锁保护的。

atmel\_spi\_next\_message函数的大体代码如下：

void atmel\_spi\_next\_message(struct spi\_master \*master)

{

struct atmel\_spi \*as = spi\_master\_get\_devdata(master);

struct spi\_message \*msg;

struct spi\_device \*spi;

① msg = list\_entry(as->queue.next, struct spi\_message, queue);

spi = msg->spi;

② cs\_activate(as, spi);

③ atmel\_spi\_next\_xfer(master, msg);

}

1. 从spi\_master结构体的链表中提取出spi\_message。
2. 使能SPI的片选脚，为之后的SPI传输做准备。
3. atmel\_spi\_next\_xfer即是SPI最底层的传输函数。

atmel\_spi\_next\_xfer函数的大体代码如下：

void atmel\_spi\_next\_xfer(struct spi\_master \*master, struct spi\_message \*msg)

{

struct atmel\_spi \*as = spi\_master\_get\_devdata(master);

struct spi\_transfer \*xfer;

u32 len, remaining;

u32 ieval;

dma\_addr\_t tx\_dma, rx\_dma;

① xfer = list\_entry(msg->transfers.next,

struct spi\_transfer, transfer\_list);

len = xfer->len;

② atmel\_spi\_next\_xfer\_data(master, xfer, &tx\_dma, &rx\_dma, &len);

remaining = xfer->len - len;

③ spi\_writel(as, RPR, rx\_dma);

spi\_writel(as, TPR, tx\_dma);

spi\_writel(as, RCR, len);

spi\_writel(as, TCR, len);

as->current\_remaining\_bytes = remaining;

④ spi\_writel(as, IER, ieval);

spi\_writel(as, PTCR, SPI\_BIT(TXTEN) | SPI\_BIT(RXTEN));

}

1. 从spi\_message结构体中提取出一个节点，节点的数据类型为spi\_transfer。
2. 找到先前为spi\_transfer结构体计算好的DMA地址。
3. 把DMA地址和长度信息装入DMA寄存器。
4. 使能DMA传输和DMA传输完成中断。至此，SPI的传输已经几乎完成了，接下来SPI传输完成中断会触发，在中断函数中再做一些收尾工作就全部完成了。

SPI的中断函数大体如下：

irqreturn\_t atmel\_spi\_interrupt(int irq, void \*dev\_id)

{

struct spi\_master \*master = dev\_id;

struct atmel\_spi \*as = spi\_master\_get\_devdata(master);

struct spi\_message \*msg;

struct spi\_transfer \*xfer;

u32 status, pending, imr;

int ret = IRQ\_NONE;

spin\_lock(&as->lock);

xfer = as->current\_transfer;

① msg = list\_entry(as->queue.next, struct spi\_message, queue);

imr = spi\_readl(as, IMR);

status = spi\_readl(as, SR);

pending = status & imr;

② if (pending & (SPI\_BIT(RXBUFF) | SPI\_BIT(ENDRX))) {

ret = IRQ\_HANDLED;

spi\_writel(as, IDR, pending);

③ if (as->current\_remaining\_bytes == 0) {

msg->actual\_length += xfer->len;

atmel\_spi\_dma\_unmap\_xfer(master, xfer);

if (xfer->delay\_usecs)

udelay(xfer->delay\_usecs);

atmel\_spi\_msg\_done(master, as, msg, 0,

xfer->cs\_change);

} else {

④ atmel\_spi\_next\_xfer(master, msg);

}

}

spin\_unlock(&as->lock);

return ret;

}

1. 从spi\_master的私有数据的链表中提取出spi\_message。
2. 如果是传输完成中断。
3. 如果传输完成，执行收尾工作。
4. 如果传输未完成，继续传输。

atmel\_spi\_msg\_done函数细节比较多，大体如下：

void atmel\_spi\_msg\_done(struct spi\_master \*master, struct atmel\_spi \*as,

struct spi\_message \*msg, int status, int stay)

{

cs\_deactivate(as, msg->spi);

list\_del(&msg->queue);

msg->status = status;

spin\_unlock(&as->lock);

msg->complete(msg->context);

spin\_lock(&as->lock);

as->current\_transfer = NULL;

as->next\_transfer = NULL;

spi\_writel(as, PTCR, SPI\_BIT(RXTDIS) | SPI\_BIT(TXTDIS));

}

把片选脚禁能。

把spi\_message从spi\_master的私有数据的链表中删除。

调用spi\_message中的complete函数通知上层函数。

### 板上SPI设备

spi\_master结构体包含了SPI硬件的底层操作函数，它会被spi.c里的各种函数调用，而spi.c里的函数，比如spi的传输，就是用户编写SPI板上驱动的时候可以用的，说白了，spi.c就是对SPI底层的二次封装。它位于：linux-2.6.39\drivers\spi\spi.c。

Linux源码提供的SPI板上驱动正是调用了spi.c里的驱动函数来操作板上SPI设备的，比如：

linux-2.6.39\drivers\mtd\devices\mtd\_dataflash.c

linux-2.6.39\drivers\mtd\devices\m25p80.c

linux-2.6.39\drivers\spi\spidev.c

以m25p80.c为例：

先初始化一个spi\_driver结构体，这个结构体的probe方法包含了m25p80的初始化操作，主要是mtd设备的添加，而mtd设备的读写、擦除使用的SPI底层操作函数全部来自spi.c。

这个结构体很重要，它决定了spi设备的数量，假设spi\_board\_info的数量为n，那么spi\_master->setup也会被执行n次。

static struct spi\_driver m25p80\_driver = {

.driver = {

.name = "m25p80",

.bus = &spi\_bus\_type,

.owner = THIS\_MODULE,

},

.id\_table = m25p\_ids,

.probe = m25p\_probe,

.remove = \_\_devexit\_p(m25p\_remove),

};

static int \_\_init m25p80\_init(void)

{

return spi\_register\_driver(&m25p80\_driver);

}

static void \_\_exit m25p80\_exit(void)

{

spi\_unregister\_driver(&m25p80\_driver);

}

### SPI外设与板上SPI设备的关联

一个芯片上的SPI外设往往不止一个，板上SPI设备的数量也不可预料，所以需要一段代码来把SPI外设和板上SPI设备关联起来。

先初始化一个spi\_board\_info结构体，里面包含所有的板上SPI设备各自占用的SPI外设编号，以及速度和别名等信息。注意这个别名不是随便取的，比如有个板上设备是m25p80，而内核已经支持这个设备了且内核给这个设备取名为m25p80，所以，如果你想用内核原生的驱动，就应该把这个别名取为跟内核一样。spi\_board\_info初始化完成后，由spi\_register\_board\_info注册进内核。

controller\_data是一个无类型指针，意味着它是用户可以使用的，Atmel利用这个指针来存储SPI设备的片选脚，在后续的函数中，从spi\_device结构体中提取出controller\_data就可以知道片选脚是多少了。而chip\_select指的是SPI外设的编号，Atmel把chip\_select代入一个一维数组求出片选脚，片选脚再赋给controller\_data，下面的代码中controller\_data本来是没有的，是需要后续的函数算出来的，为了便于理解，我加了上去。

注意chip\_select这个变量，它指的是SPI外设的编号，从0开始；而前文说的master->num\_chipselect指的是最后一个SPI外设的编号，所以chip\_select必须小于等于num\_chipselect。

static struct spi\_board\_info yl9200\_spi\_devices[] = {

{ /\* Touchscreen \*/

.modalias = "ads7846",

.chip\_select = 0,

.max\_speed\_hz = 5000 \* 26,

.irq = AT91\_PIN\_PB11,

.controller\_data = 3,

},

{ /\* CAN \*/

.modalias = "mcp2510",

.chip\_select = 1,

.max\_speed\_hz = 25000 \* 26,

.irq = AT91\_PIN\_PC0,

.controller\_data = 4,

}

};

spi\_register\_board\_info(yl9200\_spi\_devices, ARRAY\_SIZE(yl9200\_spi\_devices));

## IIC设备

## NAND设备

NandFlash驱动位于linux-2.6.39\drivers\mtd\nand\。NandFlash属于MTD（Memory Technology Device）设备，Linux中的文件系统（YAFFS、YAFFS2、JFFS、JFFS2、UBIFS）可以挂载在NandFlash上。NandFlash上面要有分区，就像Window系统上硬盘的分区是CDEF四个盘一样，NandFlash上的分区可以由u-boot指定也可以由Linux自己指定，如果两者都没有，那么Linux无法启动。用户编写NandFlash驱动一般只需要提供写命令、读写数据、判忙等少数几个底层就行，因为Linux已经封装了有关NandFlash的一整套函数集，这套函数集把NandFlash的识别、读写、擦除、ECC校验全部做进去了，兼容市面上大多数的NandFlash，它位于linux-2.6.39\drivers\mtd\nand\nand\_base.c。

下面是一个典型的NandFlash平台设备结构体，可见，一个NandFlash除了典型的resource、platform\_device结构体外，还要有个分区表­­­——mtd\_partition结构体，这个结构体一般是挂在platform\_device结构体的platform\_data下，它存放NandFlash上的各个分区的位置和大小等属性。

static struct resource nand\_resources[] = {

{

.start = NAND\_BASE,

.end = NAND\_BASE + SZ\_256M - 1,

.flags = IORESOURCE\_MEM,

}

};

static struct platform\_device at91rm9200\_nand\_device = {

.name = "atmel\_nand",

.id = -1,

.dev = {

.platform\_data = & yl9200\_nand\_partition,

},

.resource = nand\_resources,

.num\_resources = ARRAY\_SIZE(nand\_resources),

};

static struct mtd\_partition \_\_initdata yl9200\_nand\_partition[] = {

{

.name = "AT91 NAND partition 1, boot",

.offset = 0,

.size = SZ\_256K

},

{

.name = "AT91 NAND partition 2, kernel",

.offset = MTDPART\_OFS\_NXTBLK,

.size = (2 \* SZ\_1M) - SZ\_256K

},

{

.name = "AT91 NAND partition 3, filesystem",

.offset = MTDPART\_OFS\_NXTBLK,

.size = 14 \* SZ\_1M

},

{

.name = "AT91 NAND partition 4, storage",

.offset = MTDPART\_OFS\_NXTBLK,

.size = SZ\_16M

},

{

.name = "AT91 NAND partition 5, ext-fs",

.offset = MTDPART\_OFS\_NXTBLK,

.size = SZ\_32M

}

};

接下来是platform\_driver的注册（删除了部分代码，只保留关键部分）：

#ifdef CONFIG\_MTD\_CMDLINE\_PARTS

static const char \*part\_probes[] = { "cmdlinepart", NULL };

#endif

static int \_\_init orion\_nand\_probe(struct platform\_device \*pdev)

{

struct mtd\_info \*mtd;

struct nand\_chip \*nc;

struct orion\_nand\_data \*board;

struct resource \*res;

void \_\_iomem \*io\_base;

int ret = 0;

#ifdef CONFIG\_MTD\_PARTITIONS

struct mtd\_partition \*partitions = NULL;

int num\_part = 0;

#endif

① nc = kzalloc(sizeof(struct nand\_chip) + sizeof(struct mtd\_info), GFP\_KERNEL);

mtd = (struct mtd\_info \*)(nc + 1);

res = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0);

io\_base = ioremap(res->start, resource\_size(res));

② board = pdev->dev.platform\_data;

mtd->owner = THIS\_MODULE;

③ mtd->priv = nc;

④ nc->priv = board;

⑤ nc->IO\_ADDR\_R = nc->IO\_ADDR\_W = io\_base;

nc->cmd\_ctrl = orion\_nand\_cmd\_ctrl;

nc->read\_buf = orion\_nand\_read\_buf;

nc->ecc.mode = NAND\_ECC\_SOFT;

if (board->chip\_delay)

nc->chip\_delay = board->chip\_delay;

if (board->width == 16)

nc->options |= NAND\_BUSWIDTH\_16;

if (board->dev\_ready)

nc->dev\_ready = board->dev\_ready;

platform\_set\_drvdata(pdev, mtd);

⑥ if (nand\_scan(mtd, 1)) {

ret = -ENXIO;

goto no\_dev;

}

#ifdef CONFIG\_MTD\_PARTITIONS

#ifdef CONFIG\_MTD\_CMDLINE\_PARTS

mtd->name = "orion\_nand";

⑦ num\_part = parse\_mtd\_partitions(mtd, part\_probes, &partitions, 0);

#endif

/\* If cmdline partitions have been passed, let them be used \*/

if (num\_part <= 0) {

num\_part = board->nr\_parts;

partitions = board->parts;

}

if (partitions && num\_part > 0)

⑧ ret = add\_mtd\_partitions(mtd, partitions, num\_part);

else

⑨ ret = add\_mtd\_device(mtd);

#else

ret = add\_mtd\_device(mtd);

#endif

if (ret) {

nand\_release(mtd);

goto no\_dev;

}

return 0;

no\_dev:

platform\_set\_drvdata(pdev, NULL);

iounmap(io\_base);

no\_res:

kfree(nc);

return ret;

}

static int \_\_devexit orion\_nand\_remove(struct platform\_device \*pdev)

{

struct mtd\_info \*mtd = platform\_get\_drvdata(pdev);

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

nand\_release(mtd);

iounmap(nc->IO\_ADDR\_W);

kfree(nc);

return 0;

}

static struct platform\_driver orion\_nand\_driver = {

.remove = \_\_devexit\_p(orion\_nand\_remove),

.driver = {

.name = "orion\_nand",

.owner = THIS\_MODULE,

},

};

static int \_\_init orion\_nand\_init(void)

{

return platform\_driver\_probe(&orion\_nand\_driver, orion\_nand\_probe);

}

static void \_\_exit orion\_nand\_exit(void)

{

platform\_driver\_unregister(&orion\_nand\_driver);

}

位于linux-2.6.39\drivers\mtd\nand\orion\_nand.c。

重点讲orion\_nand\_probe函数：

1. 分配nand\_chip和mtd\_info结构体。
2. 获取dev的私有数据，名为board。
3. mtd\_info的前一个数据是nand\_chip。
4. nand\_chip的前一个数据是board。
5. 填充nand\_chip结构体，包括它的读写地址、读缓冲函数、发命令函数、延时函数、等待忙状态函数等。
6. 调用nand\_scan函数读取NandFlash硬件，如果你的片子符合linux-2.6.39\drivers\mtd\nand\nand\_ids.c中定义的各种NandFlash属性，那么内核就能识别出来，如果识别不出来，NandFlash就无法注册。
7. 分析u-boot传来的mtd分区信息，如果有，就获取mtd\_partition结构体指针和它的分区数目；如果没有，就从board中获取先前预设的mtd\_partition结构体指针以及它的分区数目。
8. 把mtd\_partition结构体指针以及它的分区数目作为实参，传给add\_mtd\_partitions函数注册分区。
9. 如果⑥⑦不使能，就执行add\_mtd\_device仅仅添加mtd设备，不注册分区。

## USB设备

## LCD显示器

**摄像头**

## 网卡

## 声卡

## SD卡