SDIO/SPI WIFI 硬件原理及 Linux 驱动模型分析

华清远见深圳中心 学员 李枝果 lizhiguo0532@163.com 2010-5-03

版权声明：

以下文章是 华清远见 深圳培训中心 学员在学习期间的兴趣专题文章，版权属学员个人和华清远见 深圳培训中心共同所有，欢迎转载，但转载须保留华清远见 深圳培训中心和学员个人信息

目前市面上出现的 wifi 模块有 usb，uart，spi，sdio 四种接口，本文档只针对 sdio wifi 的驱动，而且内核的版本为 2.6.14

以下部分将涉及硬件原理，工作流程，驱动模型几个方面：

一、硬件原理

1. wm664-m 模块芯片（system in package (sip) module）

   wm664-m 是 cyberTAN 在 Marvell 的 88w8686 裸片的基础上集成的一款同时具有 spi 和 sdio 接口的 wifi 模块芯片。

   其重要特性如下：

Product Type：	802.11b/g LGA SiP module
Interface(s)：	SDIO 1.0 and Generic SPI
Main Chip(s)：	88W8686
Package：	49-pin LGA
Wireless Standard(s)：	IEEE 802.11b and 802.11g
Operating Frequency：	2412–2484 MHz
Data Rates：	802.11g: 6, 9, 12, 18, 24, 36, 48, and 54 Mbps. 802.11b: 1, 2, 5.5, and 11 Mbps.

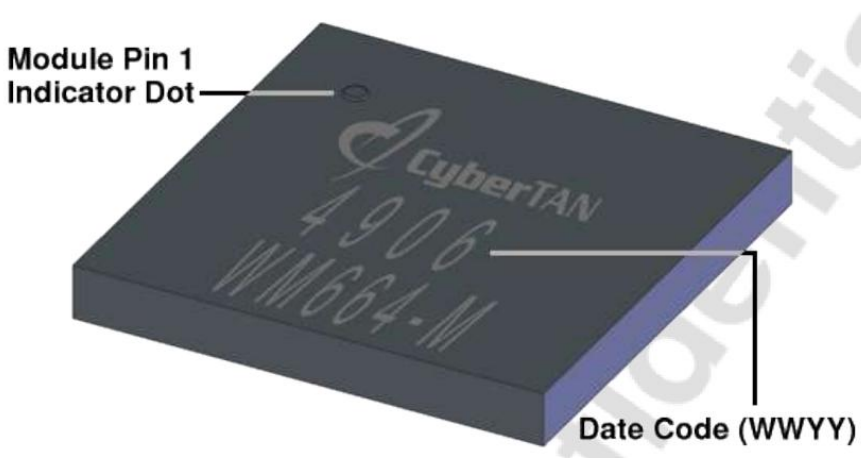
   ...

Operating System Support:	WinCE 5.0, Windows Mobile 5.0, Linux 2.6.9 and above
Power Requirements:	Standby mode: 160 mA Power saving mode (DTIM=1): 6 mA Tx mode: 265 mA (continuous transmission) Rx mode: 200 mA

   ...

   外观结构图：

Shenzhen Farsight Inc.



Interfaces

Pin	SDIO 4 bit mode		SDIO 1 bit mode		SPI		GSPI	
	Pin define	description	Pin define	description	Pin define	description	Pin define	description
1	CD/DAT[3]	data line 3	NC	Not used	CS	card Select	NC	Not used
2	CMD	command line	CMD	command line	DI	Data input	DI	Data input
3	VSS1	Ground	VSS1	Ground	VSS1	Ground	VSS1	Ground
4	VDD	Supply volteag	VDD	Supply	VDD	Supply voltage	VDD	Supply voltage
5	CLK	Clock	CLK	Clock	SCLK	Clock	SCLK	Clock
6	VSS2	Ground	VSS2	Ground	Vss2	Gound	Vss2	Gound
7	DAT[0]	data line 0	DATA	Data line	DO	Data output	CSn	card Select
8	DAT[1]	data line 1 or Interrupt(option)	IRQ	interrupt	IRQ	interrupt	DO	Data output
9	DAT[2]	data line 2 or Read Wait(Option)	RW	Read Wait(optional)	NC	Not used	IRQ	interrupt

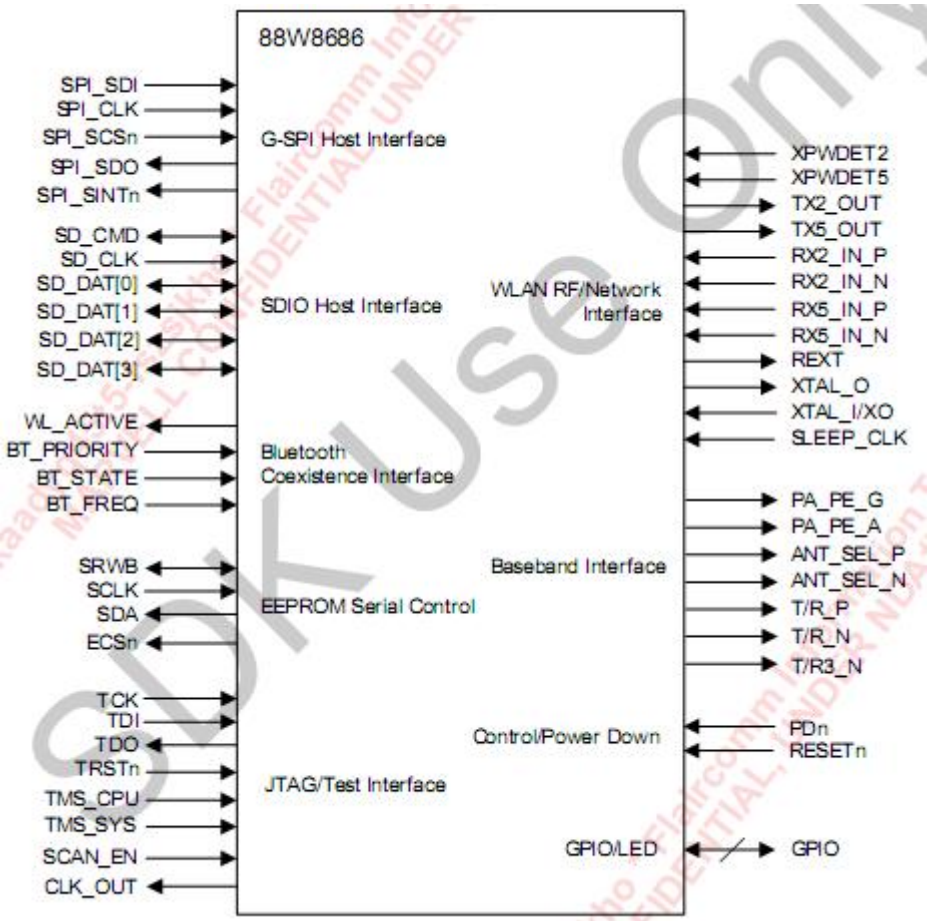
2. Marvell 88w8686 介绍

8686 芯片内部集成了两个 arm7 core(ARMv5TE)，128MHZ 的工作频率。一个 core 负责对网络数据的处理(接收来自 host 的 IP 层数据包，然后再加上 wifi 的协议头和 MAC 地址，或着解析数据包，发送给 host 断)，另一个 core 负责处理 RF 方面的工作，比如信源编码解码，信道编码解码，载波调制解调等，也就是将基带信号通过频带传输发送给对端，再反向解释信息。

88w8686 芯片的主要特性参考其数据手册。

88w8686 是 QFN 68-pin 封装

功能引脚框图



SDIO commands

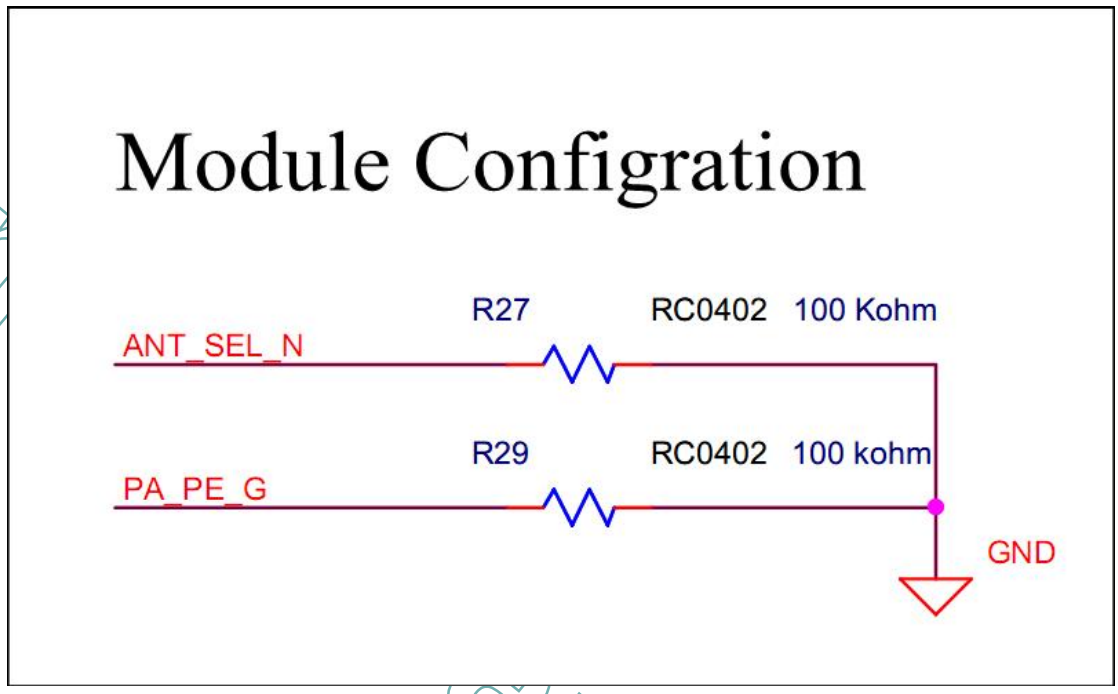
Table 41: SDIO Electrical Function Definition

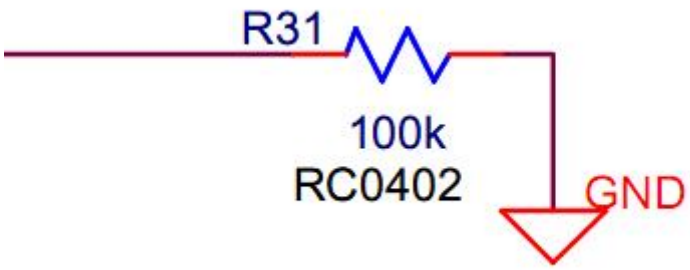
Pin	SDIO 4-bit Mode		SDIO 1-bit Mode		SPI Mode	
1	CD/DAT[3]	Data line 3	N/C	Not used	CS	Card Select
2	CMD	Command line	CMD	Command line	DI	Data input
3	VSS1	Ground	VSS1	Ground	VSS1	Ground
4	VDD	Supply voltage	VDD	Supply voltage	VDD	Supply voltage
5	CLK	Clock	CLK	Clock	SCLK	Clock
6	VSS2	Ground	VSS2	Ground	VSS2	Ground
7	DAT[0]	Data line 0	DATA	Data line	DO	Data output
8	DAT[1]	Data line 1 or Interrupt (optional)	IRQ	Interrupt	IRQ	Interrupt
9	DAT[2]	Data line 2 or Read Wait (optional)	RW	Read Wait (optional)	NC	Not used

3. spi 和 SDIO 接口的选择

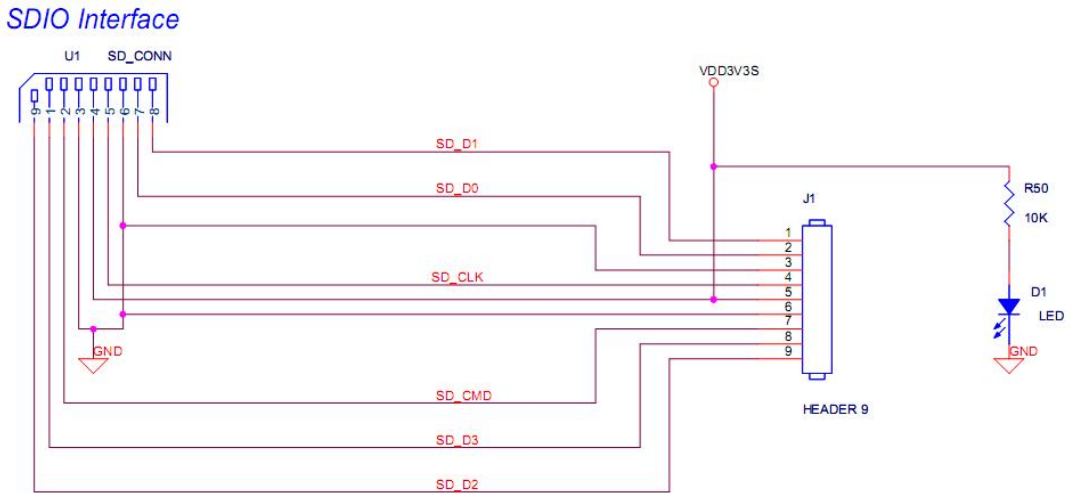
在 wm664 wifi 模块的原理图中通过跳线进行选择。如下图所说:

R27,R29,R31	Host Interface Select: 001 Generic SPI 110 SDIO (default in pad)
	0 : load 100 kohm 1 : not load





Shenzhen Farsight Inc.



4. s3c2410 SOC 的片上 MMC/SD/SDIO HOST CONTROLLER

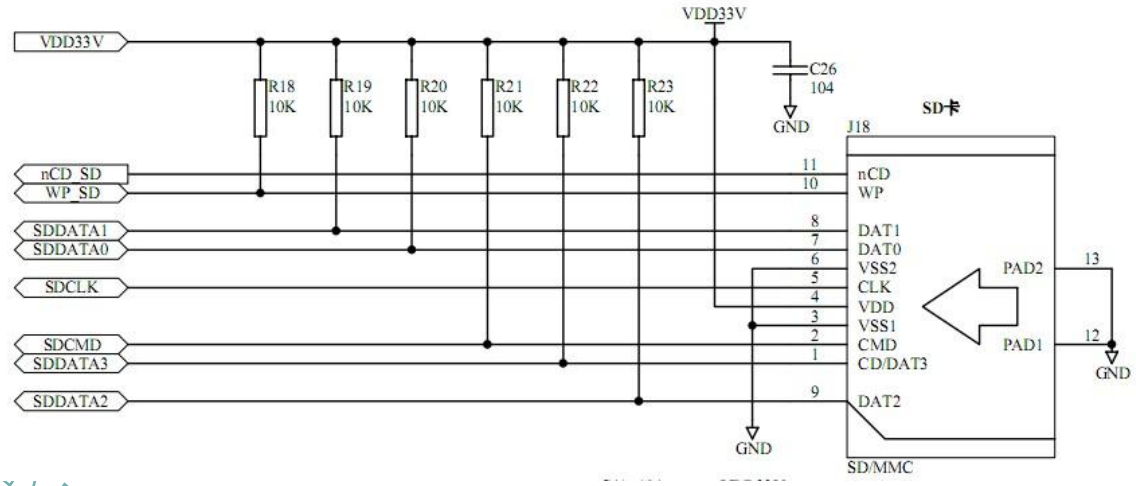
OVERVIEW

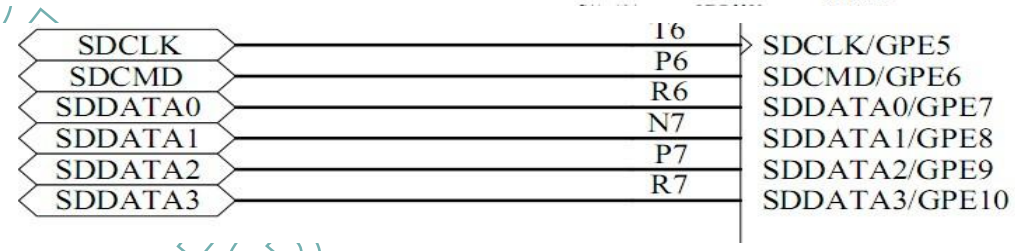
The S3C2410A SD Host controller can support MMC/SD card and SDIO devices.

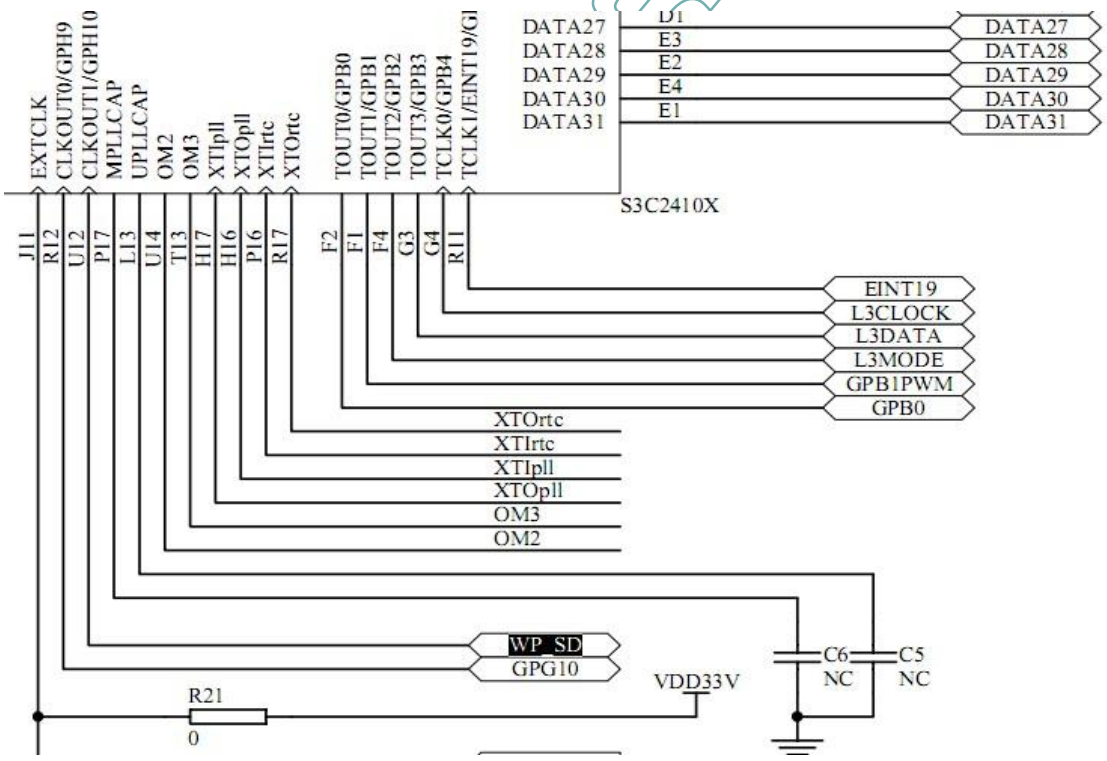
FEATURES

- SD Memory Card Spec. (ver. 1.0) / MMC Spec. (2.11) compatible 支持SD存储卡规范1.0、MMC卡规范2.11
- SDIO Card Spec (ver. 1.0) compatible 支持SDIO卡规范1.0
- 16 words (64 bytes) FIFO (depth 16) for data Tx/Rx 内部有16个字（64B）的FIFO，用于发送和接受
- 40-bit Command Register (SDICARG[31:0]+SDICCON[7:0]) 40位的命令寄存器
- 136-bit Response Register (SDIRSPn[127:0]+ SDICSTA[7:0]) 136位回应寄存器
- 8-bit Prescaler logic (Freq. = System Clock / (2(P + 1))) 8位的预分频逻辑电路
- CRC7 & CRC16 Generator CRC7和CRC16校验码产生器
- Polling, Interrupt and DMA Data Transfer Mode (Byte or Word transfer) 支持轮询，中断和DMA传输模式
- 1-bit / 4-bit (wide bus) Mode & Block / Stream Mode Switch support 数据总线的宽度可以是1bit和4bits，支持串流（stream）和区块（Block）传输方式
- Supports up to 25 MHz in data transfer mode for SD/SDIO 对于SD卡，SDIO卡，传输数据时最高时钟频率25MHZ
- Supports up to 20 MHz in data transfer mode for MMC 对于MMC卡，传输数据时最高时钟频率25MHZ

5. Fs2410 开发板上关于 sd 插槽的电路部分







由上面的电路可以看出，fs2410 上关于 sd 卡的电路很简单，总共就 11 根线：

Sd 卡插槽端	连接标号	S3C2410 端
1(CD/DAT3)pin	←----------SDDATA3----------→	SDDATA3/GPE10
2(CMD)pin	←----------SDCMD ----------	SDCMD/GPE6
3(VSS1)pin	GND	
4(VDD)pin	VDD33V(3.3V)	
5(CLK)	←--------- SDCLK ----------	SDCLK/GPE5
6(VSS2)pin	GND	
7(DAT0)pin	←----------SDDATA0----------→	SDDATA0/GPE7
8(DAT1)pin	←----------SDDATA1----------→	SDDATA1/GPE8
9(DAT2)pin	←----------SDDATA2----------→	SDDATA2/GPE9
10(WP_SD)pin	---------- WP_SD(写保护信号)-----→	CLKOUT1/GPH10
11(nCD_SD)pin	---------- nCD_SD(卡插拔中断)----→	EINT18/GPG10

注：C26 104 为 0.1uF 的去耦电容

6．MMC/SD/SDIO 的一些概念

MMC： MultiMedia Card 的缩写，即多媒体卡。24mm x 32mm x 1.4mm。以前的 MMC 规范的数据传输宽度只有 1 位，最新的 1.0 版 MMC 规范中拓宽了 4 位、8 位带宽，时钟达到 52MHZ，从而支持 50MHZ 的传输速率。对于 SD 开始的“数据保全”特性，MMC 协会接纳了具有竞争心的安全卡标准-----Secure MMC 1.1 版规范。

SD： Secure Digital Memory Card，即安全数码卡。他在 MMC 的基础上发展而来，并且增加了两个新的特色点：SD 卡强调数据的保全，可以设定所存数据的使用权限，防止他人复制；传输速率比 2.11 版本的 MMC 卡块了 4 倍。在数据传输和物理规范上，SD 卡向前兼容 MMC 卡，在外观上，SD 卡和 MMC 卡具有相同大小的正面投影，就是厚度上比 MMC 卡大了 0.7mm。最后，SD 卡和 2.11 版本的 MMC 卡完全兼容。

SDIO: SDIO 是在 SD 标准上定义了一种外设接口，它和 SD 卡规范之间的一个重要区别就是增加了低速设备。SDIO 卡只需要 SPI 和 1 位 SD 传输模式。低速卡的目标应用时以最小的硬件开支支持低速 I/O 能力。对于“组合卡”（存储卡+SDIO）而言，全速和 4 位操作对卡内存储器和 SDIO 部分都是强制要求的。

MMC/SD/SDIO 这 3 种存储卡都支持两种接口：对于 MMC 卡，称为 MMC 接口和 SPI 接口；对于 SD 卡，SDIO 卡，称为 SD 接口和 SPI 接口。SD 接口有 1 位和 4 位之分，上电时默认的使用 1 位模式，设置 SD 主机后可以使用 4 位模式。

MCI： MCI 是 MultiMedia Card Interface 的简称，即多媒体卡接口。上面的 MMC、

SD、SDIO 卡定义的接口都属于 MCI 接口。

7．SDIO 卡

Shenzhen Farsight Inc.

SDIO 是目前我们比较关心的技术，顾名思义，就是 SD 的 I/O 接口的意思。更具体一点就是，SD 本来就是记忆卡的标准，SD 卡本来是需要查到 SD 卡插槽上才能使用的，但是现在也可以在 SD 插槽上插上一些外围模块使用，这样的技术就是 SDIO。

SDIO 卡也就是外接的可以直接查到 SD 插槽上使用的模块（当然驱动程序是必须的）

目前常见的 SDIO 卡有：

Wi-Fi card（无线网络卡）

CMOS sensor card（照相模块）

GPS card

GSM/GPRS modem card

Bluetooth card

Radio/TV card

二、 整个流程框架

这里介绍的就只是一个大概的工作流程，因为还没有去仔细分析代码，也就省略了很多细节的东西。

这里首先澄清两个概念：IEEE 802.11 这个协议就是传说中的无线局域网协议，这个协议规定的是以空气作为传输介质的数据传输过程。另外一个就是 wm664-m 模块的数据传输协议，这个是定义在 SDIO 接口的基础上的，也就说规定了数据在 SDIO 口上传输必须遵循的规则。终归到底，这两种协议有很多的不同，首先表现在制定者身上，另外很明显的就是传输介质的不同了，最后就是沟通对象不同了。

网络数据在内核的协议栈中封成了 IP 包之后，将数据放在一个和 wifi 模块驱动知道的地方，然后 wifi 模块的驱动去将数据取出来，通过 wm664-m 模块的数据传输协议（当然协议的实现肯定已经封装成了函数，这个协议是基于 SDIO 接口的，所以最后数据肯定是通过 SOC（2410）的 SDIO 控制器发送给 wm664-m 模块的，所以 wifi 模块的驱动程序及必须的和 SDIO 控制器打交道了），将数据传给 wm664 的。

前面已经知道了 wm664 中封装了 88w8686，而 8686 中集成了两个 ARM7 的 core。Wm664 中负责处理网络数据的 core 受到数据之后，对数据检验，然后加上 MAC 地址和 802.11 协议规定的一些头，就会将数据转交给另外一个 core，最后由这个 core 负责将数据进行编码，再调制到高频带发送出去。

当然接收数据的过程和这个是大同小异的，关键在什么地方？关键就在需要明白整个过程中，那个驱动负责做了什么事情，在什么时候做，以什么方式去做。

三、 linux 驱动架构

说起这个话题就大了去了，但是如果对驱动的模型没有一点认识就去急于看或者写驱动的话，最后自己会崩溃掉(只针对喜欢打破砂锅问到底的人来说的)。

下面是我研究了几天驱动架构得到的一点心得，虽然没有将整个驱动模型研究透，但是我现在至少不在对那么多关键结构体的联系那么困惑了。

网上很多牛逼的人将驱动架构的，我的所得均是参透自那些前辈的经验。

Shenzhen Farsight Inc.

1．Sysfs 文件系统

Linux 2.6 内核开始引入了 sysfs 文件系统，它是一种虚拟的文件系统。

它的作用就是：把链接在系统上的设备和总线组成一个分级的文件，他们可以由用户空间存取，同时向用户空间到处内核数据结构和他们的属性。

Sys 目录下一般包括以下几个文件：block bus class dev devices firmware fs kernel module power。暂时先不谈这个文件什么作用，以后自然会明白。

2．相关结构体预览

struct kobject 内核总最基本的对象结构体
struct kset 内核对象的集合
struct subsystem 内核对象子系统，是一系列 kset 的集合
struct device 用来描述系统中任一个设备
struct device_driver 用来描述系统中的任意一个驱动程序
struct bus_type 用来描述系统中的一条总线
…

3．总线 bus

总线是处理器和一个或多个设备之间的通道，物理上是什么概念呢？简单点就是数据总线，地址总线，复杂点的 AHB，APB，北桥等，还记得 AHB 和 APB 上都挂了什么吗？请看下图

FCLK, HCLK, and PCLK

FCLK is used by ARM920T.
HCLK is used for AHB bus, which is used by the ARM920T, the memory controller, the interrupt controller, the LCD controller, the DMA and the USB host block.
PCLK is used for APB bus, which is used by the peripherals such as WDT, IIS, I2C, PWM timer, MMC interface, ADC, UART, GPIO, RTC and SPI.

这些总线都是物理上存在的，但是现在在我们的设备驱动模型中出现了一种虚拟总线 platform bus 的概念,这个总线物理上不存在，内核通过它来管理系统中数量种类繁多的各种设备和驱动。另外总线是可以相互插入的，理解这句话的关键需要知道，内核也把总线也看做是一种设备，不管是否物理上存在。它被用来连接处在仅有最少基本组件的总线上的那些设备. 这样的总线包括许多片上系统上的那些用来整合外设的总线，也包括一些"古董" PC 上的连接器；但不包括像 PCI 或 USB 这样的有庞大正规说明的总线.

在 Linux 设备模型中，总线由 bus_type 结构表示，定义在 include\linux\device.h 中

```
struct bus_type {
const char     * name;/*总线名称*/
struct subsystem  subsys;/*与该总线相关的子系统*/
struct kset       drivers; /*总线驱动程序的 kset*/
struct kset       devices; /* 挂在该总线的所有设备的 kset*/
struct klist      klist_devices; /*与该总线相关的驱动程序链表*/
struct klist      klist_drivers; /*挂接在该总线的设备链表*
struct bus_attribute * bus_attrs; /*总线属性*/
struct device_attribute  * dev_attrs;/*设备属性,指向为每个加入总线的设备建立的默认属性链表*/
```

```
    struct driver_attribute  * drv_attrs;/*驱动程序属性*/

    int    (*match)(struct device * dev, struct device_driver * drv);
    int    (*hotplug) (struct device *dev, char **envp,
                   int num_envp, char *buffer, int buffer_size);
    int    (*suspend)(struct device * dev, pm_message_t state);
    int    (*resume)(struct device * dev);
};
```

在 drivers\base\platform.c 中定义了一个虚拟总线，如下：

```
    struct device platform_bus = {
    .bus_id       = "platform",
    };
```

在注册 struct device 前，最少要设置 parent，bus_id，bus，和 release 成员，但是这里他只有个 bus_id，其余均为 NULL，说明什么问题呢？说明这条总线是一个顶层总线设备。

```
    struct bus_type platform_bus_type = {
    .name      = "platform",
    .match     = platform_match,
    .suspend       = platform_suspend,//暂停
    .resume        = platform_resume,//重新开始
    };
```

不要在乎怎么这个结构体的某些项没有被赋值呢？不用担心，编译器会自动处理的，指针会赋 NULL，整型赋 0。platform_bus 描述了一个设备，platform_bus_type 描述了一个总线。

同样在 platform.c 文件中，有如下函数

```
    int __init platform_bus_init(void)
    {
        device_register(&platform_bus);
        return bus_register(&platform_bus_type);
    }
```

platform_bus_init()首先调用 device_register()将 platform_bus 这个设备注册进系统的设备树中，该函数一旦调用完成，新总线 platform_bus 会在 sysfs 中 /sys/devices 下显示，任何挂到这个总线的设备会在 /sys/devices/platform 下显示。

接下来调用 bus_register()来注册 platform_bus_type 这条总线，调用可能失败，所以这里必须始终检查返回值（这里在调用 platform_bus_init 的函数里检查），若成功，新的总线子系统将被添加进系统，并可在 sysfs 的 /sys/bus 下看到。任何挂到这个总线上的设备都会在/sys/bus/platform/devices 下建立指向/sys/devices/ platform 下对应设备的链接文件，下图 1 所示。任何挂到这个总线的驱动程序都会在/sys/bus/platform/drivers 下显示。从图 2 可以看出，/sys/bus/platform/devices 中的每个设备的 driver 域均指向了/sys/bus/platform/drivers 下的某个对应的驱动，同是还可以看出在驱动对应

Shenzhen Farsight Inc.

的目录下还有一个链接文件指向/sys/bus/platform/devices 中对应的设备。比如，设备 floppy.0 和驱动 floppy 的关系

Ubuntu 9.10 下验证：

```
lzg@lzg-desktop:/sys/devices/platform$ ll
总计 0
drwxr-xr-x 3 root root    0 2010-05-03 21:49 eisa.0
drwxr-xr-x 4 root root    0 2010-05-03 21:49 Fixed MDIO bus.0
drwxr-xr-x 4 root root    0 2010-05-03 21:49 floppy.0
drwxr-xr-x 5 root root    0 2010-05-03 21:49 i8042
drwxr-xr-x 3 root root    0 2010-05-03 21:49 pcspkr
drwxr-xr-x 2 root root    0 2010-05-03 21:51 power
drwxr-xr-x 4 root root    0 2010-05-03 21:49 serial8250
-rw-r--r-- 1 root root 4096 2010-05-03 21:49 uevent
lzg@lzg-desktop:/sys/devices/platform$ pwd
/sys/devices/platform
lzg@lzg-desktop:/sys/devices/platform$
```

搜搜月历 带你"门"缝里看人

```
lzg@lzg-desktop:/sys/bus/platform/devices$ ll
总计 0
lrwxrwxrwx 1 root root 0 2010-05-03 21:49 eisa.0 -> ../../../devices/platform/eisa.0
lrwxrwxrwx 1 root root 0 2010-05-03 21:49 Fixed MDIO bus.0 -> ../../../devices/platform/Fixed MDIO bus.0
lrwxrwxrwx 1 root root 0 2010-05-03 21:49 floppy.0 -> ../../../devices/platform/floppy.0
lrwxrwxrwx 1 root root 0 2010-05-03 21:49 i8042 -> ../../../devices/platform/i8042
lrwxrwxrwx 1 root root 0 2010-05-03 21:49 pcspkr -> ../../../devices/platform/pcspkr
lrwxrwxrwx 1 root root 0 2010-05-03 21:49 serial8250 -> ../../../devices/platform/serial8250
lzg@lzg-desktop:/sys/bus/platform/devices$ pwd
/sys/bus/platform/devices
lzg@lzg-desktop:/sys/bus/platform/devices$
```

图 1

```
lzg@lzg-desktop:/sys/bus/platform/drivers$ ls
dsa  floppy  i8042  mdio-gpio  parport_pc  serial8250  tw14030_reg
lzg@lzg-desktop:/sys/bus/platform/drivers$ cd floppy/
lzg@lzg-desktop:/sys/bus/platform/drivers/floppy$ ls
bind  floppy.0  uevent  unbind
lzg@lzg-desktop:/sys/bus/platform/drivers/floppy$ ll
总计 0
--w------- 1 root root 4096 2010-05-03 22:07 bind
lrwxrwxrwx 1 root root    0 2010-05-03 22:07 floppy.0 -> ../../../../devices/platform/floppy.0
--w------- 1 root root 4096 2010-05-03 22:07 uevent
--w------- 1 root root 4096 2010-05-03 22:07 unbind
lzg@lzg-desktop:/sys/bus/platform/drivers/floppy$
```

```
lzg@lzg-desktop:/sys/devices/platform/floppy.0$ ll
总计 0
drwxr-xr-x 3 root root    0 2010-05-03 21:49 block
-r--r--r-- 1 root root 4096 2010-05-03 21:49 cmos
lrwxrwxrwx 1 root root    0 2010-05-03 21:49 driver -> ../../../bus/platform/drivers/floppy
-r--r--r-- 1 root root 4096 2010-05-03 22:08 modalias
drwxr-xr-x 2 root root    0 2010-05-03 22:08 power
lrwxrwxrwx 1 root root    0 2010-05-03 21:49 subsystem -> ../../../bus/platform
-rw-r--r-- 1 root root 4096 2010-05-03 21:49 uevent
lzg@lzg-desktop:/sys/devices/platform/floppy.0$
```

图 2

几乎 linux 设备模型中的每一层都提供添加属性的函数，总线也不例外。

struct bus_attribute 继承自 struct attribute。

```
struct bus_attribute {
struct attribute attr;
ssize_t (*show)(struct bus_type *, char * buf);
ssize_t (*store)(struct bus_type *, const char * buf, size_t count);
};
```

可以看出 struct bus_attribute 和 struct attribute 很相似，其实大部分在 kobject 级上的设备模型层都是以这种方式工作。

内核提供了一个宏在编译时创建和初始化 bus_attribute 结构：

```
BUS_ATTR(_name,_mode,_show,_store)/*这个宏声明一个结构，将
bus_attr_ 作为给定 _name 的前缀来创建总线的真正名称*/
```

```
/*总线的属性必须显式调用 bus_create_file 来创建:*/
int bus_create_file(struct bus_type *bus, struct bus_attribute
*attr);
/*删除总线的属性调用:*/
void bus_remove_file(struct bus_type *bus, struct bus_attribute
*attr);
```

```
lzg@lzg-desktop:/sys$ ls
block  bus  class  dev  devices  firmware  fs  kernel  module  power
lzg@lzg-desktop:/sys$ cd bus/
lzg@lzg-desktop:/sys/bus$ ls
ac97  acpi  eisa  gameport  hid  i2c  isa  MCA  mdio_bus  mmc  pci  pci_express
lzg@lzg-desktop:/sys/bus$ cd platform/
lzg@lzg-desktop:/sys/bus/platform$ ls
devices  drivers  drivers_autoprobe  drivers_probe  uevent
lzg@lzg-desktop:/sys/bus/platform$
```

可以看到后面的 drivers_autoprobe drivers_probe 均是其属性文件，不确定 uevent 是不是。

回到前面的话题，所以说，系统总不管是物理总线还是虚拟的总线 platform bus 都是被看做设备在管理的，只不过这种是设备是总线设备，具有设备和总线的双重属性。

这一节出现了一个，将某设备或驱动“挂”在一个总线下面的话，这句话可能大多数人在没看后面的内容之前会比较晕，这里先提一下，所谓的“挂”就是指设备的结构体和驱动的结构体中的 bus_type 域指向这根总线。

5. 设备 device

从 linux2.6 内核开始就引入了新的设备管理和注册机制：platform_device 和 platform_driver。Linux 中的大部分驱动，都可以使用这一套机制，设备用 Platform_device 表示，驱动用 Platform_driver 进行注册。

platform driver 机制和传统的 device driver 机制(通过 driver_register 函数进行注册)相比，一个十分明显的优势在于 platform 机制将设备本身的资源注册进内核，由内核统一管理，在驱动程序中使用这些资源时通过 platform device 提供的标准接口进行申请和使用。

这样做提高了驱动和资源管理的独立性，并且拥有较好的可移植性和安全性。

a. Platform device 平台设备

平台设备通常指的是系统中的自治体，包括老式的基于端口的设备（比如简单的一个 LED 灯，或者一个蜂鸣器，这种直接依靠 port 口输出 0 或者 1 来控制的设备）和连接外设总线（外部扩展总线）的北桥(host bridges),以及集成在片上系统中的绝大多数控制器（如，nand 控制器，SDIO 控制器，LCD 控制器等）．它们通常拥有的一个共同特征是直接编址于 CPU 总线上．即使在某些罕见的情况下，平台设备会通过某段其他类型的总线连入系统，它们的寄存器也会被直接编址.平台设备会分到一个名称(用在驱动绑定中)以及一系列诸如地址和中断请求号(IRQ)之类的资源.

Shenzhen Farsight Inc.

b. Platform device 结构体定义

```
/* drivers/base/platform.c */
/* 2.6.14 中还没有 platform_driver*/
struct platform_device {
 const char * name;            /*设备名*/
                        /*系统正是通过这个名字来与驱动绑定的，
                       所以驱动里面相应的设备名必须与该项相符合*/
 u32     id;                  /* id 表示设备编号 */
 struct device dev;
 u32     num_resources;     /*资源数目*/
 struct  resource  * resource;
                        /*resource，Linux 设计了这个通用的数据
                        结构来描述各种 I/O 资源
                        (如：I/O 端口、外设内存、DMA 和 IRQ 等)*/
};
```

c. 关于 platform device 更详细的介绍请参考我的另一篇文档，当前目录下的：

platform 平台设备注册流程-lizgo

d. platform device 设备注册

系统中所有的 platform device 平台设备全部都是挂在 platform bus 虚拟总线上的，这里可以在下图中看出

```
/* 我们这里先只关心SDC 平台设备的注册
** 下面的注释有些是针对SDC 平台设备
*/
int platform_device_register(struct platform_device * pdev)
{
    int i, ret = 0;

    if (!pdev)
        return -EINVAL;     /* 野指针保护 */

    if (!pdev->dev.parent)      /* s3c_device_sdi->dev.parent ==NULL */
        pdev->dev.parent = & platform_bus ;

    pdev->dev.bus = & platform_bus_type ;  /*platform 虚拟总线描述*/

    if (pdev->id != -1)             /*s3c_device_sdi->id == -1*/
        snprintf(pdev->dev.bus_id, BUS_ID_SIZE, "%s.%u", pdev->name,
    else
```

再回头去看看 struct platform_device 结构体中，应经包含了一个 struct device dev 结构体，下面是 struct device 结构体的定义

```
struct device {
    struct klist      klist_children;
    struct klist_node   knode_parent;/* node in sibling list */
```

```
    struct klist_node   knode_driver;
    struct klist_node   knode_bus;
    struct device     *parent;/* 设备的 "父" 设备，该设备所属的设
备，通常一个父设备是某种总线或者主控制器. 如果 parent 是 NULL，则
该设备是顶层设备，较少见 */
   struct kobject kobj; /*代表该设备并将其连接到结构体系中的
kobject； 注意：作为通用的规则，device->kobj->parent 应等于
device->parent->kobj*/
   char bus_id[BUS_ID_SIZE];/*在总线上唯一标识该设备的字符串;例如:
PCI 设备使用标准的 PCI ID 格式，包含:域，总线，设备，和功能号.*/
    struct device_type    *type;
    unsigned        is_registered:1;
    unsigned        uevent_suppress:1;
    struct device_attribute uevent_attr;
    struct device_attribute *devt_attr;
    struct semaphore    sem;  /* semaphore to synchronize calls to
its driver. */
    struct bus_type    * bus;     /*标识该设备连接在何种类型的总线
上*/
    struct device_driver *driver;    /*管理该设备的驱动程序*/
    void        *driver_data;    /*该设备驱动使用的私有数据成员*/
    void        *platform_data;    /* Platform specific data,
device core doesn't touch it */
    struct dev_pm_info    power;
#ifdef CONFIG_NUMA
    int        numa_node;   /* NUMA node this device is close to */
#endif
    u64        *dma_mask;    /* dma mask (if dma'able device) */
    u64        coherent_dma_mask;/* Like dma_mask, but for
                  alloc_coherent mappings as
                  not all hardware supports
                  64 bit addresses for consistent
                  allocations such descriptors. */
    struct list_head    dma_pools;    /* dma pools (if dma'ble) */
    struct dma_coherent_mem    *dma_mem; /* internal for coherent
mem override */
    /* arch specific additions */
    struct dev_archdata    archdata;
    spinlock_t        devres_lock;
    struct list_head    devres_head;
    /* class_device migration path */
    struct list_head    node;
    struct class        *class;
    dev_t          devt;       /* dev_t, creates the sysfs "dev" */
```

Shenzhen Farsight Inc.

```
    struct attribute_group    **groups;    /* optional groups */
    void    (*release)(struct device * dev);/*当这个设备的最后引用被删除时，内核调用该方法；它从被嵌入的 kobject 的 release 方法中调用。所有注册到核心的设备结构必须有一个 release 方法，否则内核将打印错误信息*/设备的析构函数
};
/*在注册 struct device 前，最少要设置 parent，bus_id，bus，和 release 成员，这些成员虽然在 devs.c 中没有被赋值，但是在 platform_devices_register()函数中重新被赋予正确的值，参见上图*/
```

e. 2410 的 MMC/SD/SDIO 控制器设备注册

MMC/SD/SDIO 控制器设备的定义和其所需的资源定义都在 devs.c 文件中。

s3c2410-sdi 的 platform_device 是在系统启动时，在 cpu.c 里的 s3c_arch_init() 函数里进行注册的，这个函数申明为 arch_initcall(s3c_arch_init);会在系统初始化阶段被调用。

arch_initcall 的优先级高于 module_init。所以会在 platform driver 注册之前调用。(include/linux/init.h)

流程如下：

static int __init s3c_arch_init(void)

```
for (i = 0; i < board->devices_count; i++, ptr++) {
    ret = platform_device_register(*ptr);

    if (ret) {
        printk(KERN_ERR "s3c24xx: failed to add board device
    }
}
```

Platfrom_device_register()

```
ret = device_register(&pdev->dev);/*注册设备，并将设备添加到设备树
if (ret == 0)
    return ret;
```

```
int device_register(struct device *dev)
```

```
int device_register(struct device *dev)
{
    device_initialize(dev);
    return device_add(dev);
}
```

```
int device_add(struct device *dev)
```

```
    if ((error = device_pm_add(dev)))
        goto PMError;
    if ((error = bus_add_device(dev)))
        goto BusError;
```

```
int bus_add_device(struct device * dev)
{
    struct bus_type * bus = get_bus(dev->bus);
    int error = 0;

    if (bus) {
        pr_debug("bus %s: add device %s\n", bus->name, dev->bus_id);
        device_attach(dev);
        klist_add_tail(&dev->knode_bus, &bus->klist_devices);
        error = device_add_attrs(bus, dev);
        if (!error) {
            sysfs_create_link(&bus->devices.kobj, &dev->kobj, dev->
            sysfs_create_link(&dev->kobj, &dev->bus->subsys.kset.ko
        }
    }
    return error;
}
```

```
int device_attach(struct device * dev)
{
    int ret = 0;

    down(&dev->sem);
    if (dev->driver) {
        device_bind_driver(dev);
        ret = 1;
    } else
        ret = bus_for_each_drv(dev->bus, NULL, dev, __device_attach);
    up(&dev->sem);
    return ret;
}
```

上图可以看出，如果 device 结构体的 driver 域已经制定了 driver 的话，就不会去遍历整个 platform_bus 虚拟总线，如果没定义的话，就会去遍历整个虚拟总线。

下面的图开始时遍历整个虚拟总线的过程，bus_for_each_drv()函数中会传递一个函数指针进去__device_attach。

```
int bus_for_each_drv(struct bus_type * bus, struct device_driver *
            void * data, int (*fn)(struct device_driver *, void *))
{
    struct klist_iter i;
    struct device_driver * drv;
    int error = 0;

    if (!bus)
        return -EINVAL;

    klist_iter_init_node(&bus->klist_drivers, &i,
                 start ? &start->knode_bus : NULL);
    while ((drv = next_driver(&i)) && !error)
        error = fn(drv, data);
    klist_iter_exit(&i);
    return error;
}
```

上图中的阴影部分就是遍历的过程展现，下图是进入__device_attach 函数，传递给__device_attach 什么参数呢？可以从图中看出来，传递了一个 driver 指针（看到 drv = next_driver(&i)了吗？）和 data(实际上就是本设备指针)

```
static int __device_attach(struct device_driver * drv, void * data)
{
    struct device  * dev = data;
    return driver_probe_device(drv, dev);
}
```

看到 data 又被强制转换成了 struct device 的指针了吗？

```
 */
int driver_probe_device(struct device_driver * drv, struct device  *
{
    int ret = 0;

    if (drv->bus->match && !drv->bus->match(dev, drv))
        goto Done;/* name 没有匹配上的话立即返回 */

    pr_debug("%s: Matched Device %s with Driver %s\n",
         drv->bus->name, dev->bus_id, drv->name);
    dev->driver = drv;
    if (drv->probe) {
        ret = drv->probe(dev); /*+++++ 调用驱动程序的probe 函数++++++++++  */
        if (ret) {
            dev->driver = NULL;
            goto ProbeFailed;
        }
    }
    device_bind_driver(dev);/* 创建相应链接文件 */
    ret = 1;
    pr_debug("%s: Bound Device %s to Driver %s\n",
         drv->bus->name, dev->bus_id, drv->name);
    goto Done;

ProbeFailed:
```

在这个函数中，首先通过 driver 挂在的 bus 上的 match 函数来匹配设备名字和驱动的名字是否一样，不一样的话直接 goto done 了，如果一样的话接着就会执行 ret = drv->probe(dev);函数(如果 drv->probe! =NULL 的话，该函数为此设备的配置函数)，最后进入 device_bind_driver（）函数，进行一些链接文件和链表的处理。

这里有一点需要注意：__device_attach（）在我们的 device 只要匹配到了一个 driver 的之后，执行完相应的 probe 函数就立即返回 1，在遍历 bus 的那个循环里面（bus_for_each_drv()函数中）进行判断，之后循环退出。而后面要讲的驱动注册不一样，driver 注册的过程中，__driver_attach 会一直返回 0，知道 bus 被遍历完。所以才会出现了我们之前提到过的一句话，一个 device 最多对应一个 driver，但是一个 driver 却可以对应多个 devices。

f. 一点小总结

platfrom_device 平台设备都是通过 bus_id 挂接在虚拟的总线 platform_bus_type 上的，设备注册的时候同样会和驱动注册一样(后边会讲到)在总线上寻找相应的驱动，如果找到他也会试图绑定，绑定的过程同样是执行相应的 probe。多个 device 可以共用一个 driver，但是但是一个 device 不可以对应多个 driver。从 device 注册可以看出来

6. platform driver 平台设备驱动

目前 2.6.14 中还没有 platform driver 的机制，只有 device_driver 机制，以下关于它的内容基本都是在高版本的内核中出现。

参考高版本的内核代码中，实际上 struct platform_driver 结构体是继承于 struct device_driver，struct platform_driver 定义如下：

```
struct platform_driver {
int (*probe)(struct platform_device *);
int (*remove)(struct platform_device *);
void (*shutdown)(struct platform_device *);
int (*suspend)(struct platform_device *, pm_message_t state);
int (*suspend_late)(struct platform_device *, pm_message_t state);
int (*resume_early)(struct platform_device *);
int (*resume)(struct platform_device *);
struct device_driver driver;
};
/*
**对于我们的 SDC 驱动程序是 drivers/mmc/s3c2410mci.c
**中的 struct    device_driver    s3c2410sdi_driver 结构体
*/
struct device_driver {
    const char     * name;/*name 需要和 platfrom_devices 中的 name 一样，
                        内核正是通过这个一致性来为驱动程序找到资源*/
    struct bus_type   * bus;
    struct completion unloaded;
```


Shenzhen Farsight Inc.

```
    struct kobject        kobj;
    struct klist      klist_devices;
    struct klist_node knode_bus;
    struct module     * owner;/*owner 的作用是说明模块的所有者，
                          驱动程序中一般初始化为 THIS_MODULE*/
    int (*probe)   (struct device * dev);
    int (*remove)  (struct device * dev);
    void   (*shutdown)   (struct device * dev);
    int (*suspend) (struct device * dev, pm_message_t state, u32 level);
    int (*resume)  (struct device * dev, u32 level);
};
```

platform driver 的注册和注销

```
int platform_driver_register(struct platform_driver *drv);
int platform_driver_probe(struct platform_driver *drv,
     int (*probe)(struct platform_device *))
```

在 2.6.14 中，driver 的注册和注销时通过下面两个函数进行的

```
/*注册 device_driver 结构的函数是:*/
int driver_register(struct device_driver *drv);
void driver_unregister(struct device_driver *drv);
/*driver 的属性结构在:*/
struct driver_attribute {
struct attribute attr;
ssize_t (*show)(struct device_driver *drv, char *buf);
ssize_t (*store)(struct device_driver *drv, const char *buf, size_t
count);
};
DRIVER_ATTR(_name,_mode,_show,_store)
/*属性文件创建的方法:*/
int    driver_create_file(struct    device_driver    *    drv,    struct
driver_attribute * attr);
void    driver_remove_file(struct    device_driver    *    drv,    struct
driver_attribute * attr);
/*bus_type 结构含有一个成员( drv_attrs ) 指向一组为属于该总线的所有设
备创建的默认属性*/
```

Shenzhen Farsight Inc.

struct device_driver s3c2410sdi_driver 驱动的注册过程

in s3c2410mci.c 中

```
static struct device_driver s3c2410sdi_driver =
{
    //name 和devs.c 中struct    platform_device  s3c_device_sdi  中
    //定义的name 一样
    .name           = "s3c2410-sdi",
        .bus            = & platform_bus_type ,
       /*该驱动所管理的设备挂接的总线类型platform_bus_type*/
        .probe          = s3c2410sdi_probe,/*探测设备是否可被驱动程序管理
        .remove         = s3c2410sdi_remove,/*设备移除函数*/
};

static int __init s3c2410sdi_init(void)
{
    return driver_register(& s3c2410sdi_driver );
    /*
    **用driver_register () 向系统注册s3c2410sdi_driver  这个驱动程序
    **driver_register 会从s3c2410sdi_driver 中提取出name 信息作为搜索内容
    **搜索系统注册的device中有没有这个 platform_device
    **如果有注册，那么接着会执行s3c2410sdi_driver 里probe函数.在这里显然是s3c2410s
    **在probe函数里，用的最多和刚才platform_device有关的语句是platform_get_resourc
    **这条语句用于获取 platform_device里的resource资料.例如映射的IO地址,中断等.剩下

    **如果该sdi的驱动被编译成模块之后，则在执行insmod sdixxx 时被执行，
    **如果该adi 的驱动被编译进内核的话，则在系统启动的时候会注册该驱动
    */
}
```

当 insmod 或者系统启动加载驱动模块时自动调用 s3c2410sdi_init（），在其中又调用 driver_register(&s3c2410sdi_driver)，传进去一个 device_driver 结构体指针。

```
 */
int driver_register(struct device_driver * drv)
{
    klist_init(&drv->klist_devices, klist_devices_get, klist_devic
    init_completion(&drv->unloaded);
    return bus_add_driver(drv);
}
```

```
int bus_add_driver(struct device_driver * drv)
{
    struct bus_type * bus = get_bus(drv->bus);
    //获得该驱动对应设备注册的总线,没有，返回空
    int error = 0;

    if (bus) {
        pr_debug("bus %s: add driver %s\n", bus->name, drv->name);
        error = kobject_set_name(&drv->kobj, "%s", drv->name);
        if (error) {
            put_bus(bus);
            return error;
        }
        drv->kobj.kset = &bus->drivers;
        if ((error = kobject_register(&drv->kobj))) {
            put_bus(bus);
            return error;
        }

        driver_attach(drv);/*try to bind driver to devices, 并执行probe函数*/
        klist_add_tail(&drv->knode_bus, &bus->klist_drivers);
        module_add_driver(drv->owner, drv);

        driver_add_attrs(bus, drv);
```

```
 *   compatible pair.
 */
void driver_attach(struct device_driver * drv)
{
    bus_for_each_dev(drv->bus, NULL, drv, __driver_attach);
}
```

各位，这里看起来是不是很眼熟呢？回忆以下前面的 device_attach()函数，这个这个函数的功能其实和前面 device 注册时相似函数的功能是一样的，有几点不同的地方在哪里呢？

1. 函数名字相似但不一样
2. 功能正好相反，前面是遍历 bus 的驱动链表，用设备匹配驱动；这里呢？则是遍历 bus 的设备链表，用驱动匹配设备。
3. 关键的不同在于，driver 可以管理多个不同的设备，但一个 device 只能应用一个驱动。
4. 目前我知道的就这些

```
int bus_for_each_dev(struct bus_type * bus, struct device * s
            void * data, int (*fn)(struct device *, void *))
{
    struct klist_iter i;
    struct device * dev;
    int error = 0;

    if (!bus)
        return -EINVAL;

    klist_iter_init_node(&bus->klist_devices, &i,
                 (start ? &start->knode_bus : NULL));
    while ((dev = next_device(&i)) && !error)
        error = fn(dev, data);  /* fn= __driver_attach 一直返回0, 说明, 虽然匹
                       一个相符的设备之后，还是需要继续匹配，直到
                       设备链表结束
                       一个驱动程序可以管理多个设备*/
    klist_iter_exit(&i);
    return error;
} ? end bus_for_each_dev ?
```

```
static int __driver_attach(struct device * dev, void * data)
{
    struct device_driver * drv = data;

    /*
     * Lock device and try to bind to it. We drop the error
     * here and always return 0, because we need to keep trying
     * to bind to devices and some drivers will return an error
     * simply if it didn't support the device.
     *
     * driver_probe_device() will spit a warning if there
     * is an error.
     */

    down(&dev->sem);
    if (!dev->driver)
        driver_probe_device(drv, dev);
    up(&dev->sem);

    return 0;
} ? end __driver_attach ?
```

看到这里了没，__driver_attach（）始终返回的 0，也就是在上上一幅图片中的 while 循环里 error 始终是 0，到这里也就验证了设备和驱动不同的应用关系了吧。

在后面的 driver_probe_device(drv, dev);和前面的 device 设备注册时一样的了。当然在这里会调用驱动的 probe 函数。然后做些和 sys 文件系统相关的工作就退出了注册过程。

7. 类子系统的简单介绍

类是一个设备的高层视图，它抽象出了底层的实现细节，从而允许用户空间使用设备所提供的功能，而不用关心设备是如何连接和工作的。类成员通常由上层代码所控制，而无需驱动的明确支持。但有些情况下驱动也需要直接处理类。几乎所有的类都显示在 /sys/class 目录中。出于历史的原因，有一个例外：块设备显示在 /sys/block 目录中。在许多情况，类子系统是向用户空间导出信息的最好方法。当类子系统创建一个类时，它将完全拥有这个类,根本不用担心哪个模块拥有那些属性，而且信息的表示也比较友好。
为了管理类，驱动程序核心导出了一些接口，其目的之一是提供包含设备号的属性以便自动创建设备节点，所以 udev 的使用离不开类。 类函数和结构与设备模型的其他部分遵循相同的模式，所以真正崭新的概念是很少的

暂时分析到这里

全文完

2010/5/3 lizgo 李枝果 lizhiguo0532@163.com
over

Shenzhen Farsight Inc.