Linux SPI 子系统驱动程序结构分析

关键之：SPI、framework、platform、driver、device

Linux SPI 这个子系统系列的介绍会在 linux 驱动模型的基础上进行阐述，会偏重于 framework 的介绍，对于大牛可能会对这类文章不屑，但本系列仅当是一个知识备忘，当 linux 体系这张大网织的差不多了，会有一个全新的系列，来去繁就简，成之经典，毕竟，现阶段，对这些的感悟还不是太深，将原来的工作内容进行回忆，将现在工作碰到的问题补充，下一阶段会有更深的体会的。

由于这是这个系统的第一篇文章，可能零碎的东西介绍的会多些。

0，分层与分离

在面向对象的程序设计中，可以为某一类相似的事物定义一个基类，而具体的事物可以继承这个基类中的函数。Linux 内核中频繁使用到面向对象的设计思想。在设备驱动方面，往往为同类的设备设计了一个框架，而框架中的核心层则实现了该设备通用的一些功能。而且具体的设备不想使用核心层的函数，它可以重载之。这就是我们所说的在驱动设计中的分层思想。

此外，在驱动的设计中，我们还会使用分离的思想。如果一个设备的驱动和 host 的驱动休戚相关，那么，这就意味着这个普通的设备如果用在不同的 host 上，会采用 n 个版本的驱动。如果产品单一，也许感觉不到不使用分离思想来设计驱动的危害，但是我们想一下，这个世上被人们称道的多是什么？精品，艺术品！精品如何打造？注重细节，不只考虑单一需求！大家开发个东西不容易，怎么能随随便便就让它茫然众码矣呢，所以，何时何地，我们都要以打造精品的思想来要求自己，让自己的劳动力不浪费。

使用分离的思想来设计驱动的话，就够就是这样的：

外设驱动与主机控制器的驱动不相关，主机控制器的驱动不关心外设，而外设驱动也不关心主机，外设只是访问核心层的通用 API 进行数据传输，主机和外设之间可以进行任意的组合。相当于在控制器驱动和设备驱动之间增加一层核心层，对内对外都隐藏了对端的不确定性。仔细通读 USB，SPI，PCI 的代码就会发现这种思想的体现。

1，设备模型

在最新的设备驱动模型中，主要包含总线、设备和驱动三个实体，总线将设备和驱动绑定，在系统每注册一个设备的时候，会寻找与之匹配的驱动，反之，在系统每注册一个驱动的时候，会寻找与之匹配的设备，而匹配由总线完成。

所以，因此，由是之...(之所以写这么多，是因为自己在理解这个设备模型的时候，对照代码产生很多疑问，特别是在这儿。现在回过头来看，觉得很显而易见的啊，：）看来那个什么 Q 还是有些问题)，根据这个模型的需求，一个现实的 linux 设备和驱动通常都需要挂接在一种总线上，否则谁来管他们的匹配啊，注册驱动和注册设备都是由不同的 API 来完成的。对于本身依附于 PCI，USB，I2C，SPI 等的设备而言，这自然不是问题，但是在嵌入式系统里面，Controller 系统中集成的外设控制器，挂载在内存空间的外设确不依附于此类总线。那

咋整呢？而且这些东西还不少，像 SPI 控制器，PCI 控制器啊都是这些，这些很多都已经是主了，还让他们靠谁去？基于这一背景，Linux 发明了一种虚拟的总线（就是我们所说的除了政治领袖以外的精神领袖），称为 platform 总线，相应的设备称为 platform_device，而驱动称为 platform_driver。

2，platform 总线

这个框架中为 platform 总线定义了一个 bus_type 的实例 platform_bus_type:

```
1 struct bus_type platform_bus_type = {
2 .name = “platform”,
3 .dev_attrs = platform_dev_attrs,
4 .match = platform_match,
5 .uevent = platform_uevent,
6 .pm = PLATFORM_PM_OPS_PTR,
7 };
8 EXPORT_SYMBOL_GPL(platform_bus_type);
```

这里重点关注其 match 成员函数，正是此成员表明了 platform_device 和 platform_driver 之间如何匹配。

```
1 static int platform_match(struct device *dev, struct device_driver *drv)
2 {
3 struct platform_device *pdev;
4
5 pdev = container_of(dev, struct platform_device, dev);
6 return (strncmp(pdev->name, drv->name, BUS_ID_SIZE) == 0);
7 }
```

从代码中可以看出，匹配 platform_device 和 platform_driver 主要看两者的 name 字段是否相同。

对 platform_device 的定义通常在 BSP 包里面实现（即 arch 目录下的），在 BSP 文件中，将 platform_device 归纳为一个数组，最终通过 platform_add_devices()函数统一注册。platform_add_devices()函数可以讲平台设备添加到系统中。

3，SPI

讲了这么多才说道 SPI，看来老婆没有说错我啊，罗里啰嗦……因为该文档权当备忘，所以接下来还会罗嗦几句 SPI 的缘由

3.1 what is SPI

SPI（同步外设接口）是由摩托罗拉公司开发的全双工同步串行总线，其接口由 MISO（串行数据输入），MOSI（串行数据输出），SCK（串行移位时钟），SS（从使能信号）四种信号构成（当然了，现在芯片技术日新月异，SPI 模块的结构也在变化中，象 OMAP 系列中的 SPI 模块还支持 5 线的一种模式），SS 决定了唯一的与主设备通信的从设备，主设备通

过产生移位时钟来发起通讯。通讯时，数据由 MOSI 输出，MISO 输入，数据在时钟的上升或下降沿由 MOSI 输出，在紧接着的下降或上升沿由 MISO 读入，这样经过 8/16 次时钟的改变，完成 8/16 位数据的传输。

SPI 模块为了和外设进行数据交换，根据外设工作要求，其输出串行同步时钟极性（CPOL）和相位（CPHA）可以进行配置。如果 CPOL=0，串行同步时钟的空闲状态为低电平；如果 CPOL=1，串行同步时钟的空闲状态为高电平。如果 CPHA=0，在串行同步时钟的第一个跳变沿（上升或下降）数据被采样；如果 CPHA=1，在串行同步时钟的第二个跳变沿（上升或下降）数据被采样。

3.2 SPI Framework

#控制器设备和驱动

控制器设备在 BSP 的初始化中注册，驱动在 drvier/spi 中

#核心层驱动

核心层代码负责这个框架中通用的部分，满足分层的思想。主题承担的工作包括：注册 spi 总线，提供基本 SPI 总线操作 API。

#SPI 外设驱动

对于 SPI 的设备驱动，因为可爱的 linux driver framework 设计者的功劳，这里我们只需要用到 spi.h 中定义的方法就可以了，不用去修改 spi 控制器的代码。一般的，我们的设备驱动框架是使用 spi_regiser_driver 向系统进行注册，就可以让系统用你指定的与.name 相匹配的硬件交互并执行你的读写请求，满足分离的思想。

spi.h 中大部分函数中都会用到 struct spi_device *spi 这个指针，在 probe 函数中获得这个指针，保存好这个指针，就可以在驱动中的任何地方通过他去处理与 spi 设备相关的操作。

3.3 SPI 控制器驱动

SPI 控制器要挂载到 platform 总线上的，so，

需要在 BSP 文件中添加相应的资源代码：

通常，会在 xxxx.c 中添加：

```
static struct platform_device da850_spi_pdev1 = {
.name = “dm_spi”,
.id = 1,
.resource = da850_spi_resources1,
.num_resources = ARRAY_SIZE(da850_spi_resources1),
.dev = {
.platform_data = &da850_spi_pdata1,
},
};
```

然后会在 BSP 的 init 过程中使用 platform_device_register 将它注册进系统

MACHINE_START(DAVINCI_DA850_EVM, “Gemstones Platform version 0.03”)
.phys_io = IO_PHYS,
.io_pg_offst = (__IO_ADDRESS(IO_PHYS) >> 18) & 0xfffc,
.boot_params = (0xC0000100),
.map_io = da850_map_io,
.init_irq = da850_evm_irq_init,
.timer = &davinci_timer,
.init_machine = **da850_evm_init,**
MACHINE_END
-> da850_evm_init()->da850_init_spi1()->platform_device_register(&**da850_spi_pdev1**)
自此，将 SPI 控制器设备注册进了系统，name 字段为 dm_spi

我们还注意到，设备的资源信息还可以放在控制器的设备资源里面，利用 platform 提供的 platform_data 的支持，其中 platform_data 的形式是自定义的。设备可以通过以下方式拿到 platform_data 信息：

struct xxxx_plat_data *pdata = pdev->dev.platform_data;

其中，pdev 为 platform_device 的指针，当然也可以通过别的方法，只要能获得 device 的指针。

到这里先小总结一下：

控制器驱动的流程主要就是：

第一步：注册 platform_device；第二步：注册 platform_driver，然后 platform 总线会让两者匹配在一起。

使用 platform 总线在驱动中大体有以下几个好处：

a，使得设备被挂接在一个总线上，使配套的 sysfs 节点、设备电源管理都成为可能。

b, 隔离了 BSP 和驱动。BSP 中定义 platform 设备和设备使用的资源（可以使用 platform_data 的形式来包括 platform 设备的设备），设备的具体配置信息。而在驱动中，只需要通过通用 API 去获取资源和数据，做到了板相关代码和驱动代码的分离，使得驱动具有更好的可扩展性和跨平台性。

以上注册完设备，下面就要开始注册驱动。

注册 platform driver，首先要有 platform_driver 数据结构

static struct platform_driver davinci_spi_driver = {
.driver = {
.name = “dm_spi”,
.owner = THIS_MODULE,
},
.probe = davinci_spi_probe,

.remove = davinci_spi_remove,
.suspend = davinci_spi_suspend,
.suspend_late = davinci_spi_suspend_late,
.resume_early = davinci_spi_resume_early,
};
name 字段要和 device 保持一致。

然后将 probe，remove，suspend，suspend_late，resume_early 函数注册，driver framwork 会在合适的时候调用。

其中 probe 函数是比较重要的一个，该函数中将创建 davinci_spi 数据结构，这个数据结构在文档上描述为 spi 驱动的私有数据，该结构中包括 spi_bitbang 这个重要的数据结构，它的重要性在 probe 函数最后介绍。

在 Linux 中，每一个类型的驱动都会有一个相应的结构体来描述，这里的 **spi_master 就是用来描述 SPI 主机控制器驱动**的，其主要成员是 bus_num，cs，spi 模式和时钟设置用到的函数，数据传输用到的函数等。
分配、注册和注销 SPI 主机驱动结构体的 API 由 SPI 核心层提供：

struct spi_master * spi_alloc_master(struct device *host, unsigned size);

int spi_register_master(struct spi_master *master);

void spi_unregister_master(struct spi_master *master);

这里可能会有疑问，spi master controller 注册对应的驱动框架里的哪一层？按说，spi 主机控制器已经作为 platform 设备都注册过了。

这里就需要用驱动设计里面的分层思想来解释。使用到 platform 总线 API 注册到 platform 总线上的控制器设备和驱动，都是 common 的部分。specific 的部分，会在 platform driver 注册后，在 probe 函数里面基于注册的 common 部分的资源信息来具体实现。称之为 spi_master 的注册部分。

一个 master 对应一个 spi 总线，或者说是一个 spi 接口，

struct spi_master {
struct device dev;
s16 bus_num;

u16 num_chipselect;

int (*setup)(struct spi_device *spi);

int (*transfer)(struct spi_device *spi,
struct spi_message *mesg);
void (*cleanup)(struct spi_device *spi);
};
仔细浏览代码可以发现，在 SPI 向 platform 总线注册的时候，象 platform_device 中的 device_data 或 driver_data 都是空的，并没有将它赋值，这些都是在 probe 函数中完成的。

统观之，probe 函数里面的数据结构的设计和代码都是按照驱动框架的要求来实现的，主要包括 spi controller memory 的映射，IRQ 的注册，controller 的 reset 和对 register 进行初始化赋值。最后，它将调用 spi_bitbang_start()来创建一个 work queue，由此 SPI 设备驱动可以向这个工作队列注册 transfer 方法。

在啰唆下去，估计光控制器这块的内容就要溢出了，该系列仅作为一个工作备忘和总结，一个框架性的介绍就够了，正如 fudan_abc 大侠所说的，源码，specific，datasheet 是搞通 linux 的三件宝……

3.4 SPI 设备驱动

SPI 设备要挂载到 SPI 总线上，这个过程是由 SPI core 提供的一些 API 来完成的，spi_register_driver()，spi_register_device()。

module init 函数中使用 spi_register_driver 注册外设驱动，并注册 char 设备，导出 SPI 操作 API。

SPI 外设驱动遍布内核的 driver 的各个子目录下，SPI 是一种总线，spi_driver 的作用是将 spi 外设挂接到该总线上，因此在 spi_driver 的 probe()成员函数中，将注册 SPI 外设本身所属设备驱动的类型。和 platform_driver 对应着一个 platform_device 一样，spi_driver 也对应着一个 spi_device。platform_device 需要在 BSP 文件（即 arch）中添加板信息数据，而 spi_device 也需要，spi_device 的板级信息用 spi_board_info 结构体来描述，该结构体记录 SPI 外设使用的主机控制器序号、片选序号、比特率、SPI 传输模式等。在系统 startup 过程中，会通过前面提到的 init_machine 来对 spi_register_board_info()进行调用，将 spi 的设备信息注册进系统。

这里大家可能会又有疑问，这里只是注册了 board_info，那 device 在哪注册的？

内核中认为 SPI 是不可以热插拔的，SPI 设备的注册不是在 arch init 的时候初始化的，我们可以从代码中发现在控制器驱动加载的时候，probe 函数里面会使用 spi_register_master 来注册 SPI Master，在该函数中调用 scan_boardinfo->spi_new_device，在 spi_new_device 函数中，我们会发现 spi_device 数据结构，然后在代码中我们 会发现 device_register 将 spi_device 注册到系统（没有使用 spi_register_device，其实殊途同归）。

3.5 SPI 传输

在 SPI 外设驱动中，当透过 SPI 总线进行数据传输的时候，使用了一套与 CPU 无关的统一接口，这套接口最关键的一个数据结构就是 spi_transfer，它用于描述 SPI 传输

```
struct spi_transfer {
const void      *tx_buf;
void            *rx_buf;
unsigned        len;
dma_addr_t      tx_dma;
dma_addr_t      rx_dma;
unsigned        cs_change:1;
u8              bits_per_word;
```

```
u16         delay_usecs;
u32         speed_hz;
struct list_head transfer_list;
};
```

而一次完整的 SPI 传输流程可能不只包含 1 次 spi_transfer，它可能包含 1 个或多个 spi_transfer，这些 spi_transfer 最终通过 spi_message 组织在一起，其定义如代码

```
struct spi_message {
struct list_head     transfers;
struct spi_device     *spi;

unsigned         is_dma_mapped:1;

void             (*complete)(void *context);
void             *context;
unsigned         actual_length;
int              status;
struct list_head     queue;
void             *state;
};
```

通过 spi_message_init()可以初始化 spi_message，而将 spi_transfer 添加到 spi_message 队列的方法则是：

```
void spi_message_add_tail(struct spi_transfer *t, struct spi_message *m);
```

发起一次 spi_message 的传输有同步和异步两种方式，使用同步 API 时，会阻塞等待这个消息被处理完。同步操作时使用的 API 是：

```
int spi_sync(struct spi_device *spi, struct spi_message *message);
```

使用异步 API 时，不会阻塞等待这个消息被处理完，但是可以在 spi_message 的 complete 字段挂接一个回调函数，当消息被处理完成后，该函数会被调用。异步操作时使用的 API 是：

```
int spi_async(struct spi_device *spi, struct spi_message *message);
```

下面代码是非常典型的初始化 spi_transfer、spi_message 并进行 SPI 数据传输的例子，同时它们也是 SPI 核心层的 2 个通用 API，在 SPI 外设驱动中可以直接调用它们进行写和读操作。

```
static inline int
spi_write(struct spi_device *spi, const u8 *buf, size_t len)
{
struct spi_transfer     t = {
.tx_buf         = buf,
.len         = len,
};
struct spi_message     m;
```

```
spi_message_init(&m);
spi_message_add_tail(&t, &m);
return spi_sync(spi, &m);
}


static inline int
spi_read(struct spi_device *spi, u8 *buf, size_t len)
{
struct spi_transfer     t = {
.rx_buf         = buf,
.len         = len,
};
struct spi_message     m;
spi_message_init(&m);
spi_message_add_tail(&t, &m);
return spi_sync(spi, &m);
}
```

此外还有一个重要的结构体，新的驱动设计里被广泛使用

```
struct spi_bitbang {
struct workqueue_struct     *workqueue;
struct work_struct     work;
spinlock_t         lock;
struct list_head     queue;
u8             busy;
u8             use_dma;
u8             flags;         /* extra spi->mode support */
struct spi_master     *master;

int     (*setup_transfer)(struct spi_device *spi,
struct spi_transfer *t);
void     (*chipselect)(struct spi_device *spi, int is_on);
#define     BITBANG_CS_ACTIVE     1     /* normally nCS, active low */
#define     BITBANG_CS_INACTIVE     0
int     (*txrx_bufs)(struct spi_device *spi, struct spi_transfer *t);
```

这是一个完成数据传输的重要结构体，数据传输是 SPI 接口的任务，结构体 master 代表了一个接口，当一个 spi_message 从上层函数传递下来时，master 的成员函数 bitbang->master->transfer 将该数据传输任务添加到工作队列头。其中工作队列 struct workqueue_struct *workqueue;的创建和 struct work_struct work 的初始化都是在函数 spi_bitbang_start()中进行的。

每次数据传输，都将要传输的数据分成多个数据段，这些数据段由数据管理结构体 spi_transfer 来管理。结构体 spi_transfer 又将挂在 m->transfers。也就是说每次数据传输都将要传输的数据包装成一个结构体 spi_message 传输下来。

函数 hw_txbyte()将要传输的数据段的第一个数据写入 SPI 数据寄存器 XXXX_SPIDAT。即便是数据接收也得向数据寄存器写入数据才能触发一次数据的传输。只需将该数据段的第一个数据写入数据寄存器就可以触发数据传输结束中断，以后的数据就在中断处理函数中写入数据寄存器。

4，小结

分层，分离，注册平台设备，注册平台驱动，注册 spi master 设备，注册 spi 设备，注册 spi 驱动，注册 char 设备 export API to userspace。通过总线驱动获得对应的设备资源，根据一些经典框架，完成各个子系统和设备的功能。