# kernel hacker 修炼之道——李万鹏

**男儿立志出乡关， 学不成名死不还。 埋骨何须桑梓地， 人生无处不青山。 ——西乡隆盛诗**

Linux驱动修炼之道-SPI驱动框架源码分析(上)

分类： linux驱动编程 2011-06-29 09:51 7960人阅读 评论(11) 收藏 举报

努力成为linux kernel hacker的人李万鹏原创作品，为梦而战。转载请标明出处

http://blog.csdn.net/woshixingaaa/archive/2011/06/29/6574215.aspx

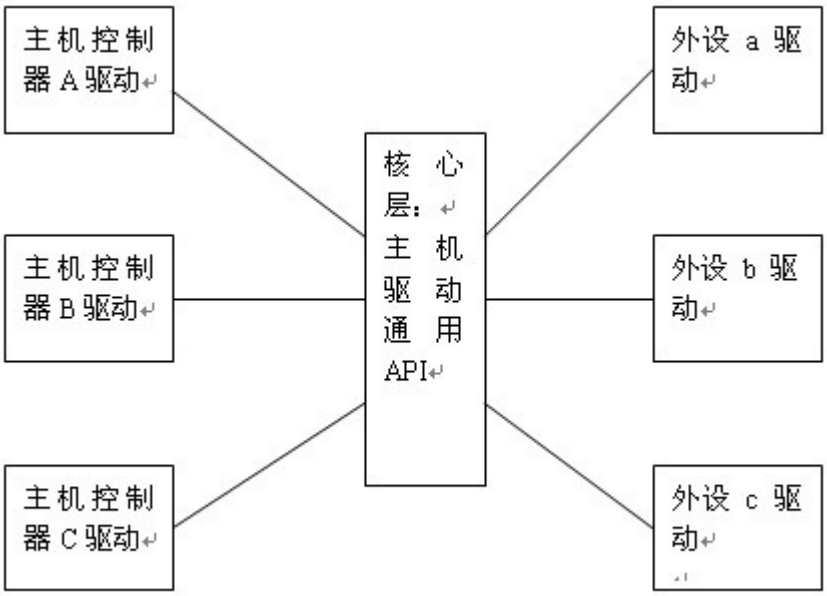
Linux驱动修炼之道-SPI驱动框架源码分析(中)

Linux驱动修炼之道-SPI驱动框架源码分析(下)

SPI协议是一种同步的串行数据连接标准，由摩托罗拉公司命名，可工作于全双工模式。相关通讯设备可工作于m/s模式。主设备发起数据帧，允许多个从设备的存在。每个从设备

有独立的片选信号，SPI一般来说是四线串行总线结构。
接口：
SCLK——Serial Clock(output from master)时钟(主设备发出)
MOSI/SIMO——Master Output, Slave Input(output from master)数据信号线mosi(主设备发出)
MISO/SOMI——Master Input,Slave Outpu(output from slave)数据信号线(从设备)
SS——Slave Select(active low;output from master)片选信号

下面来看一下Linux中的SPI驱动。在Linux设备驱动框架的设计中，有一个重要的主机，外设驱动框架分离的思想，如下图。




外设a,b,c的驱动与主机控制器A,B,C的驱动不相关，主机控制器驱动不关心外设，而外设驱动也不关心主机，外设只是访问核心层的通用的API进行数据的传输，主机和外设之间可以进行任意的组合。如果我们不进行如图的主机和外设分离，外设a,b,c和主机A,B,C进行组合的时候，需要9种不同的驱动。设想一共有个主机控制器，n个外设，分离的结构是需要m+n个驱动，不分离则需要m*n个驱动。

下面介绍spi子系统的数据结构：
在Linux中，使用spi_master结构来描述一个SPI主机控制器的驱动。

```
struct spi_master {
struct device dev;/*总线编号，从0开始*/
s16 bus_num;/*支持的片选的数量，从设备的片选号不能大于这个数量*/
u16 num_chipselect;
u16 dma_alignment;/*改变spi_device的特性如：传输模式，字长，时钟频率*/
int (*setup)(struct spi_device *spi);/*添加消息到队列的方法，这个函数不可睡眠，他的任务是安排发生的传送并且调用注册的回调函数complete()*/
int (*transfer)(struct spi_device *spi,struct spi_message *mesg);
void (*cleanup)(struct spi_device *spi);
};
```

分配，注册和注销的SPI主机的API由SPI核心提供：

```
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);
```

在Linux中用spi_driver来描述一个SPI外设驱动。

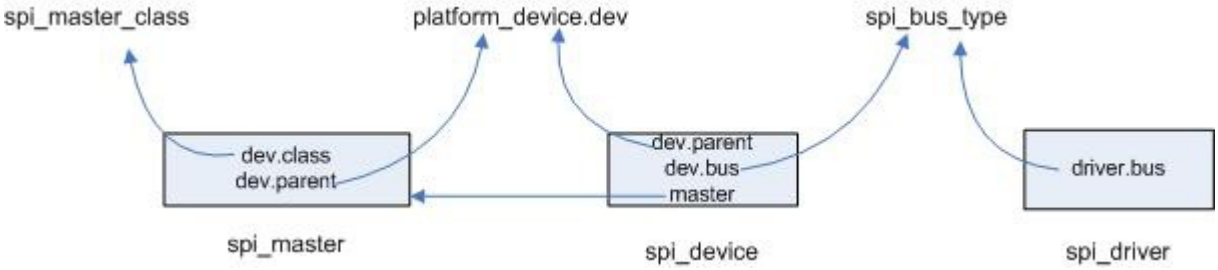
```
struct spi_driver {
int (*probe)(struct spi_device *spi);
int (*remove)(struct spi_device *spi);
void (*shutdown)(struct spi_device *spi);
int (*suspend)(struct spi_device *spi, pm_message_t mesg);
int (*resume)(struct spi_device *spi);
struct device_driver driver;
};
```

可以看出，spi_driver结构体和platform_driver结构体有极大的相似性，都有probe(),remove(),suspend(),resume()这样的接口。

Linux用spi_device来描述一个SPI外设设备。

```
 struct spi_device {
struct device dev;
struct spi_master *master; //对应的控制器指针u32
max_speed_hz; //spi通信的时钟u8
chip_select; //片选，用于区分同一总线上的不同设备
u8 mode;
#define SPI_CPHA 0x01 /* clock phase */
#define SPI_CPOL 0x02 /* clock polarity */
#define SPI_MODE_0 (0|0) /* (original MicroWire) */#define SPI_MODE_1 (0|SPI_CPHA)
#define SPI_MODE_2 (SPI_CPOL|0)
#define SPI_MODE_3 (SPI_CPOL|SPI_CPHA)#define SPI_CS_HIGH 0x04 /* chipselect active high? */
#define SPI_LSB_FIRST 0x08 /* per-word bits-on-wire */
#define SPI_3WIRE 0x10 /* SI/SO signals shared */
#define SPI_LOOP 0x20 /* loopback mode */
u8 bits_per_word; //每个字长的比特数
int irq; //使用的中断
void *controller_state;
void *controller_data;
char modalias[32]; //名字
```

}; 如下图，看这三个结构的关系，这里spi_device与spi_master是同一个父设备，这是在spi_new_device函数中设定的，一般这个设备是一个物理设备。




这里的spi_master_class,spi_bus_type又是什么呢，看下边两个结构体：

```
 struct bus_type spi_bus_type = {
.name = "spi",
.dev_attrs = spi_dev_attrs,
.match = spi_match_device,
.uevent = spi_uevent,
.suspend = spi_suspend,
.resume = spi_resume,
};
static struct class spi_master_class = {
.name = "spi_master",
.owner = THIS_MODULE,
.dev_release = spi_master_release,
};
```

spi_bus_type对应spi中的spi bus总线，spidev的类定义如下：
创建这个类的主要目的是使mdev/udev能在/dev下创建设备节点/dev/spiB.C。B代表总线，C代表片外设备的片选号。

下边来看两个板级的结构，其中spi_board_info用来初始化spi_device，s3c2410_spi_info用来初始化spi_master。这两个板级的结构需要在移植的时候在arch/arm/mach-s3c2440/mach-smdk2440.c中初始化。

```
 struct spi_board_info {
char modalias[32]; //设备与驱动匹配的唯一标识
const void *platform_data;
void *controller_data;
int irq;
u32 max_speed_hz;
u16 bus_num; //设备所归属的总线编号
u16 chip_select;
u8 mode;
};
struct s3c2410_spi_info {
int pin_cs; //芯片选择管脚
unsigned int num_cs; //总线上的设备数
int bus_num; //总线号
void (*gpio_setup)(struct s3c2410_spi_info *spi, int enable); //spi管脚配置函数
void (*set_cs)(struct s3c2410_spi_info *spi, int cs, int pol);
};
```

boardinfo是用来管理spi_board_info的结构，spi_board_info通过spi_register_board_info(struct spi_board_info const *info, unsigned n)交由boardinfo来管理，并挂到board_list链表上，list_add_tail(&bi->list,&board_list);

```
 struct boardinfo {
/*用于挂到链表头board_list上*/
struct list_head list;
/*管理的spi_board_info的数量*/
unsigned n_board_info;
/*存放结构体spi_board_info*/
struct spi_board_info board_info[0];
};
```

s3c24xx_spi是S3C2440的SPI控制器在Linux内核中的具体描述，该结构包含spi_bitbang内嵌结构，控制器时钟频率和占用的中断资源等重要成员，其中spi_bitbang具体负责SPI数据的传输。

```
 struct s3c24xx_spi {
/* bitbang has to be first */
struct spi_bitbang bitbang;
struct completion done;
void __iomem *regs;
int irq;
int len;
int count;
void (*set_cs)(struct s3c2410_spi_info *spi, int cs, int pol);
/* data buffers */const unsigned char *tx;
unsigned char *rx;
struct clk *clk;
struct resource *ioarea;
struct spi_master *master;
struct spi_device *curdev;
struct device *dev;
struct s3c2410_spi_info *pdata;
};
```

为了解决多个不同的SPI设备共享SPI控制器而带来的访问冲突，spi_bitbang使用内核提供的工作队列(workqueue)。workqueue是Linux内核中定义的一种回调处理方式。采用这种方式需要传输数据时，不直接完成数据的传输，而是将要传输的工作分装成相应的消息(spi_message)，发送给对应的workqueue，由与workqueue关联的内核守护线程(daemon)负责具体的执行。由于workqueue会将收到的消息按时间先后顺序排列，这样就是对设备的访问严格串行化，解决了冲突。

```
struct spi_bitbang {
struct workqueue_struct *workqueue; //工作队列头
struct work_struct work; //每一次传输都传递下来一个spi_message，都向工作队列头添加一个
workspinlock_t lock;
struct list_head queue; //挂接spi_message，如果上一次的spi_message还没有处理完，接下来的spi_message就
```

```
挂接在queue上等待处理
u8 busy; //忙碌标志
u8 use_dma;
u8 flags;
struct spi_master *master;/*一下3个函数都是在函数s3c24xx_spi_probe()中被初始化*/
int (*setup_transfer)(struct spi_device *spi,struct spi_transfer *t); //设置传输模式
void (*chipselect)(struct spi_device *spi, int is_on); //片选
#define BITBANG_CS_ACTIVE 1 /* normally nCS, active low */
#define BITBANG_CS_INACTIVE 0/*传输函数，由s3c24xx_spi_txrx来实现*/
int (*txrx_bufs)(struct spi_device *spi, struct spi_transfer *t);
u32 (*txrx_word[4])(struct spi_device *spi,unsigned nsecs,u32 word, u8 bits);
};
```

下面来看看spi_message：

```
struct spi_message {
struct list_head transfers; //此次消息的传输队列，一个消息可以包含多个传输段
struct spi_device *spi; //传输的目的设备
unsigned is_dma_mapped:1; //如果为真，此次调用提供dma和cpu虚拟地址
void (*complete)(void *context); //异步调用完成后的回调函数
void *context; //回调函数的参数
unsigned actual_length; //此次传输的实际长度
int status; //执行的结果，成功被置0，否则是一个负的错误码
struct list_head queue;
void *state;
};
```

在有消息需要传递的时候，会将spi_transfer通过自己的transfer_list字段挂到spi_message的transfers链表头上。spi_message用来原子的执行spi_transfer表示的一串数组传输请求。这个传输队列是原子的，这意味着在这个消息完成之前不会有其他消息占用总线。消息的执行总是按照FIFO的顺序。

下面看一看spi_transfer：

```
 struct spi_transfer {
const void *tx_buf; //要写入设备的数据(必须是dma_safe)，或者为NULL
void *rx_buf; //要读取的数据缓冲(必须是dma_safe)，或者为NULL
unsigned len; //tx和rx的大小(字节数),这里不是指它的和，而是各自的长度，他们总是相等的
dma_addr_t tx_dma; //如果spi_message.is_dma_mapped是真，这个是tx的dma地址
dma_addr_t rx_dma; //如果spi_message.is_dma_mapped是真，这个是rx的dma地址
unsigned cs_change:1; //影响此次传输之后的片选，指示本次tranfer结束之后是否要重新片选并调用setup改变设置，这个标志可以较少系统开销u8
bits_per_word; //每个字长的比特数，如果是0，使用默认值
u16 delay_usecs; //此次传输结束和片选改变之间的延时，之后就会启动另一个传输或者结束整个消息
u32 speed_hz; //通信时钟。如果是0，使用默认值
struct list_head transfer_list; //用来连接的双向链表节点
};
```