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(下)

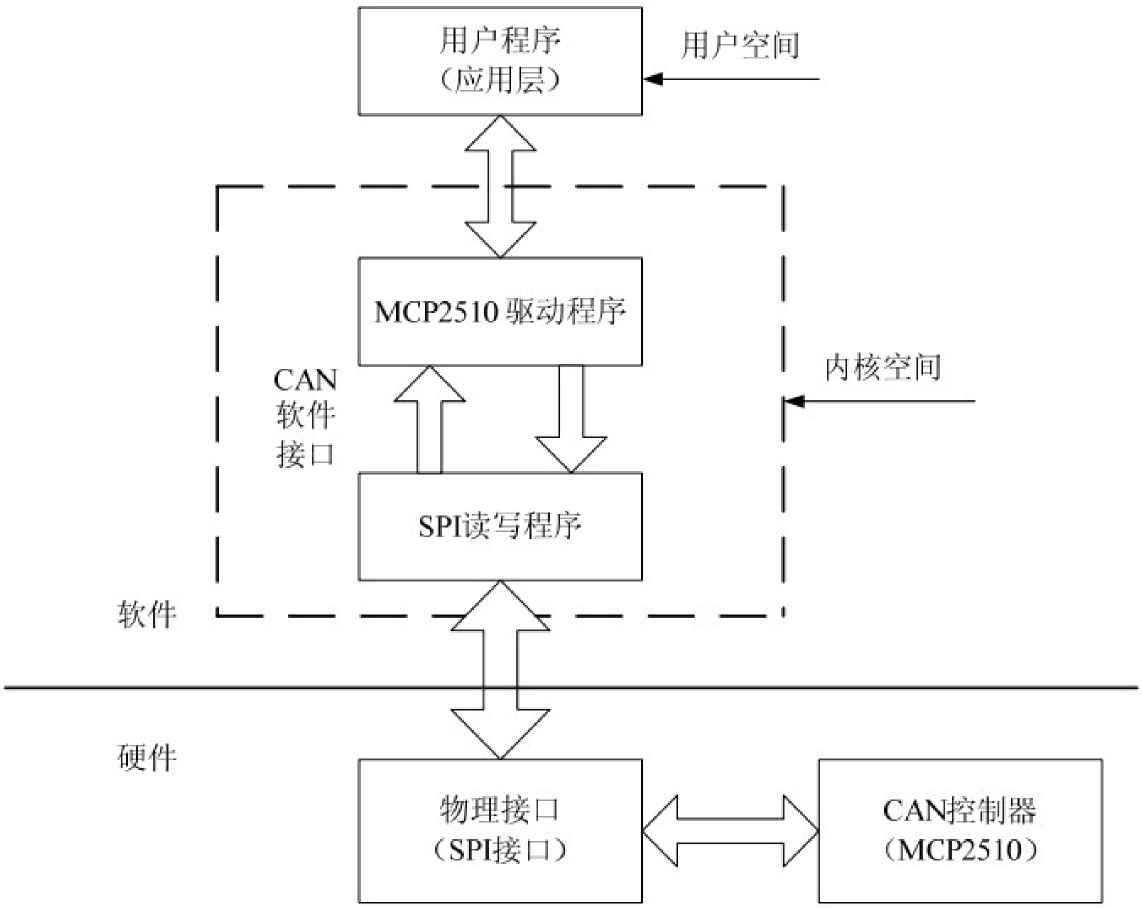
分类： linux驱动编程 2011-06-29 09:53 3321人阅读 评论(0) 收藏 举报

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/article/details/6574224

这篇文档主要介绍spi数据传输过程。

当应用层要向设备传输数据的时候，会通过ioctl向设备驱动发送传输数据的命令。如图，向SPI从设备发送读写命令，实际的读写操作还是调用了主机控制器驱动的数据传输函数。transfer函数用于spi的IO传输。但是，transfer函数一般不会执行真正的传输操作，而是把要传输的内容放到一个队列里，然后调用一种类似底半部的机制进行真正的传输。这是因为,spi总线一般会连多个spi设备，而spi设备间的访问可能会并发。如果直接在transfer函数中实现传输，那么会产生竞态，spi设备互相间会干扰。所以，真正的spi传输与具体的spi控制器的实现有关，spi的框架代码中没有涉及。像spi设备的片选，根据具体设备进行时钟调整等等都在实现传输的代码中被调用。spi的传输命令都是通过结构spi_message定义，设备程序调用transfer函数将spi_message交给spi总线驱动，总线驱动再将message传到底半部排队，实现串行化传输。



在spidev.c中实现了file_operations：

```
static struct file_operations spidev_fops = {
.owner = THIS_MODULE,
.write = spidev_write,
.read = spidev_read,
.unlocked_ioctl = spidev_ioctl,
.open = spidev_open,
.release = spidev_release,
};这里看spidev_ioctl的实现：
static long
spidev_ioctl(struct file *filp, unsigned int cmd, unsigned long
arg)
{
int err = 0;
int retval = 0;
struct spidev_data *spidev;
struct spi_device *spi;
u32 tmp;
unsigned n_ioc;
struct spi_ioc_transfer *ioc;
```

```
/*查看这个命令的幻数字段是否为'k'*/
if (_IOC_TYPE(cmd) != SPI_IOC_MAGIC)
return -ENOTTY;

/*如果方向是用户空间从内核读，即内核向用户空间写，则检查用户空间的地址是否有效*/
if (_IOC_DIR(cmd) & _IOC_READ)
err = !access_ok(VERIFY_WRITE,
(void __user *)arg, _IOC_SIZE(cmd));
/*如果方向是用户空间向内核写，即内核读用户空间，则检查用户空间的地址是否有效*/
if (err == 0 && _IOC_DIR(cmd) & _IOC_WRITE)
err = !access_ok(VERIFY_READ,
(void __user *)arg, _IOC_SIZE(cmd));
if (err)
return -EFAULT;

/* guard against device removal before, or while,
* we issue this ioctl.
*/
spidev = filp->private_data;
spin_lock_irq(&spidev->spi_lock);
spi = spi_dev_get(spidev->spi);
spin_unlock_irq(&spidev->spi_lock);

if (spi == NULL)
return -ESHUTDOWN;

mutex_lock(&spidev->buf_lock);

switch (cmd) {
/* read requests */
case SPI_IOC_RD_MODE:
/*因为已经进行了地址是否有效的检查，所以这里使用
__put_user,__get_user,__copy_from_user可以节省几个时钟周期呢*/
retval = __put_user(spi->mode & SPI_MODE_MASK,
(__u8 __user *)arg);
break;
case SPI_IOC_RD_LSB_FIRST:
retval = __put_user((spi->mode & SPI_LSB_FIRST) ? 1 : 0,
(__u8 __user *)arg);
break;
case SPI_IOC_RD_BITS_PER_WORD:
```

```
retval = __put_user(spi->bits_per_word, (__u8 __user *)arg);
break;
case SPI_IOC_RD_MAX_SPEED_HZ:
retval = __put_user(spi->max_speed_hz, (__u32 __user *)arg);
break;

/*设置SPI模式*/
case SPI_IOC_WR_MODE:
retval = __get_user(tmp, (u8 __user *)arg);
if (retval == 0) {
/*先将之前的模式保存起来，一旦设置失败进行回复*/
u8 save = spi->mode;

if (tmp & ~SPI_MODE_MASK) {
retval = -EINVAL;
break;
}

tmp |= spi->mode & ~SPI_MODE_MASK;
spi->mode = (u8)tmp;
retval = spi_setup(spi);
if (retval < 0)
spi->mode = save;
else
dev_dbg(&spi->dev, "spi mode %02x\n", tmp);
}
break;
case SPI_IOC_WR_LSB_FIRST:
retval = __get_user(tmp, (__u8 __user *)arg);
if (retval == 0) {
u8 save = spi->mode;

if (tmp)
spi->mode |= SPI_LSB_FIRST;
else
spi->mode &= ~SPI_LSB_FIRST;
retval = spi_setup(spi);
if (retval < 0)
spi->mode = save;
else
dev_dbg(&spi->dev, "%csb first\n",
tmp ? 'l' : 'm');
}
break;
```

```
case SPI_IOC_WR_BITS_PER_WORD:
retval = __get_user(tmp, (__u8 __user *)arg);
if (retval == 0) {
u8 save = spi->bits_per_word;

spi->bits_per_word = tmp;
retval = spi_setup(spi);
if (retval < 0)
spi->bits_per_word = save;
else
dev_dbg(&spi->dev, "%d bits per word\n", tmp);
}
break;
case SPI_IOC_WR_MAX_SPEED_HZ:
retval = __get_user(tmp, (__u32 __user *)arg);
if (retval == 0) {
u32 save = spi->max_speed_hz;

spi->max_speed_hz = tmp;
retval = spi_setup(spi);
if (retval < 0)
spi->max_speed_hz = save;
else
dev_dbg(&spi->dev, "%d Hz (max)\n", tmp);
}
break;

default:
/* segmented and/or full-duplex I/O request */
if (_IOC_NR(cmd) != _IOC_NR(SPI_IOC_MESSAGE(0))
|| _IOC_DIR(cmd) != _IOC_WRITE) {
retval = -ENOTTY;
break;
}
/*得到用户空间数据的大小*/
tmp = _IOC_SIZE(cmd);
/*如果这些数据不能分成spi_ioc_transfer的整数倍，则不能进行传输，spi_io_transfer是对spi_transfer的映射*/
if ((tmp % sizeof(struct spi_ioc_transfer)) != 0) {
retval = -EINVAL;
break;
}
/*计算出能分多少个spi_ioc_transfer*/
n_ioc = tmp / sizeof(struct spi_ioc_transfer);
```

```
if (n_ioc == 0)
break;

/*在内核中分配装载这些数据的内存空间*/
ioc = kmalloc(tmp, GFP_KERNEL);
if (!ioc) {
retval = -ENOMEM;
break;
}
/*把用户空间的数据拷贝过来*/
if (__copy_from_user(ioc, (void __user *)arg, tmp)) {
kfree(ioc);
retval = -EFAULT;
break;
}

/*进行数据传输*/
retval = spidev_message(spidev, ioc, n_ioc);
kfree(ioc);
break;
}

mutex_unlock(&spidev->buf_lock);
spi_dev_put(spi);
return retval;
}
```

下面跟踪spidev_message看看： static int spidev_message(struct spidev_data *spidev,

```
struct spi_ioc_transfer *u_xfers, unsigned n_xfers)
{
struct spi_message msg;
struct spi_transfer *k_xfers;
struct spi_transfer *k_tmp;
struct spi_ioc_transfer *u_tmp;
unsigned n, total;
u8 *buf;
int status = -EFAULT;
/*初始化spi_message的tranfers链表头*/
spi_message_init(&msg);
/*分配n个spi_transfer的内存空间，一个spi_message由多个数据段spi_message组成*/
k_xfers = kcalloc(n_xfers, sizeof(*k_tmp), GFP_KERNEL);
if (k_xfers == NULL)
```

```
return -ENOMEM;

buf = spidev->buffer;
total = 0;
/*这个for循环的主要任务是将所有的spi_transfer组装成一个
spi_message*/
for (n = n_xfers, k_tmp = k_xfers, u_tmp = u_xfers;
n;
n--, k_tmp++, u_tmp++) {
/*u_tmp是从用户空间传下来的spi_ioc_message的大小，spi_ioc_message是
对spi_message的映射*/
k_tmp->len = u_tmp->len;
/*统计要传输数据的总量*/
total += k_tmp->len;
if (total > bufsiz) {
status = -EMSGSIZE;
goto done;
}
/*spi_transfer是一个读写的buffer对，如果是要接收则把buffer给接收的
rx_buf*/
if (u_tmp->rx_buf) {
k_tmp->rx_buf = buf;
if (!access_ok(VERIFY_WRITE, (u8 __user *)
(uintptr_t) u_tmp->rx_buf,
u_tmp->len))
goto done;
}
/*如果要传输，这个buffer给tx_buf使用，从用户空间拷过来要传输的数据
*/
if (u_tmp->tx_buf) {
k_tmp->tx_buf = buf;
if (copy_from_user(buf, (const u8 __user *)
(uintptr_t) u_tmp->tx_buf,
u_tmp->len))
goto done;
}
/*指向下一段内存*/
buf += k_tmp->len;
/*最后一个transfer传输完毕是否会影响片选*/
k_tmp->cs_change = !!u_tmp->cs_change;
/*每字长的字节数*/
k_tmp->bits_per_word = u_tmp->bits_per_word;
/*一段数据传输完需要一定的时间等待*/
k_tmp->delay_usecs = u_tmp->delay_usecs;
```

```
/*初始化传输速度*/
k_tmp->speed_hz = u_tmp->speed_hz;
/*将spi_transfer通过它的transfer_list字段挂到spi_message的transfer
队列上*/
spi_message_add_tail(k_tmp, &msg);
}
/*调用底层的传输函数*/
status = spidev_sync(spidev, &msg);
if (status < 0)
goto done;

/* copy any rx data out of bounce buffer */
buf = spidev->buffer;
/*把传输数据拷贝到用户空间打印出来，可以查看是否传输成功*/
for (n = n_xfers, u_tmp = u_xfers; n; n--, u_tmp++) {
if (u_tmp->rx_buf) {
if (__copy_to_user((u8 __user *)
(uintptr_t) u_tmp->rx_buf, buf,
u_tmp->len)) {
status = -EFAULT;
goto done;
}
}
buf += u_tmp->len;
}
status = total;

done:
kfree(k_xfers);
return status;
}
```

看spidev_sync的实现：

```
static ssize_t
spidev_sync(struct spidev_data *spidev, struct spi_message
*message)
{
/*声明并初始化一个完成量*/
DECLARE_COMPLETION_ONSTACK(done);
int status;
/*指定spi_message使用的唤醒完成量函数*/
message->complete = spidev_complete;
message->context = &done;
```

```
spin_lock_irq(&spidev->spi_lock);
if (spidev->spi == NULL)
status = -ESHUTDOWN;
else
/*调用spi核心中的函数进行数据传输*/
status = spi_async(spidev->spi, message);
spin_unlock_irq(&spidev->spi_lock);

if (status == 0) {
/*等待完成量被唤醒*/
wait_for_completion(&done);
status = message->status;
if (status == 0)
status = message->actual_length;
}
return status;
}
```

spi_async在spi.h中定义的：

```
static inline int
spi_async(struct spi_device *spi, struct spi_message *message)
{
message->spi = spi;
return spi->master->transfer(spi, message);
}
```

这里的master->transfer是在spi_bitbang_start中进行赋值的:

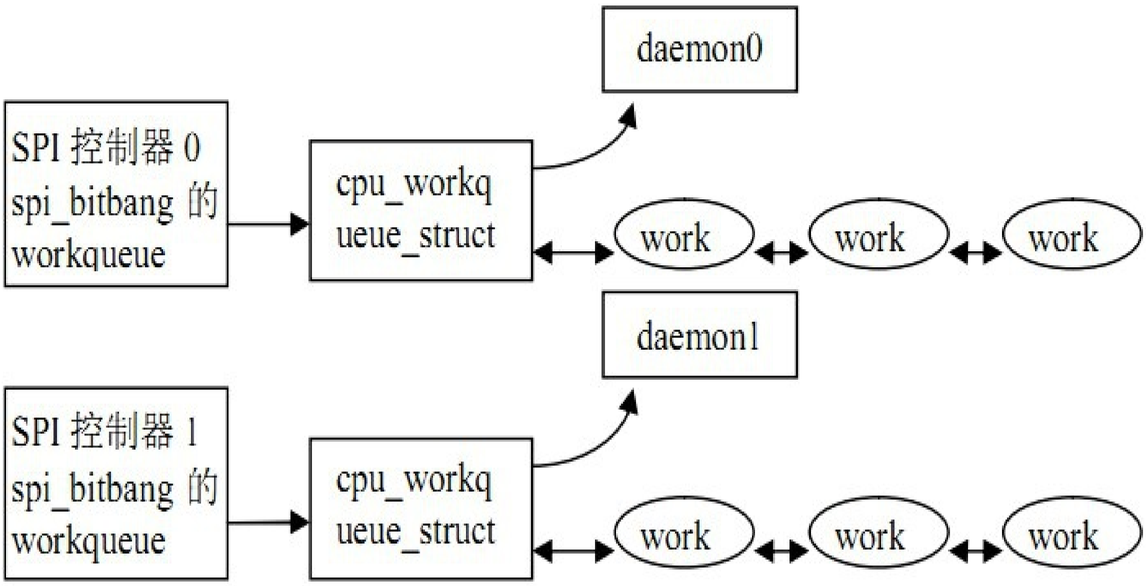
```
bitbang->master->transfer= spi_bitbang_transfer;
```

看spi_bitbang_transfer的实现： int spi_bitbang_transfer(struct spi_device *spi, struct spi_message *m)

```
{
struct spi_bitbang *bitbang;
unsigned long flags;
int status = 0;

m->actual_length = 0;
m->status = -EINPROGRESS;
/*在spi_alloc_master函数中调用spi_master_set_devdata把struct s3c24xx_spi结构存放起来，而struct spi_bitbang正是struct s3c24xx_spi结构所包含的第一个结构*/
bitbang = spi_master_get_devdata(spi->master);

spin_lock_irqsave(&bitbang->lock, flags);
```

```
if (!spi->max_speed_hz)
status = -ENETDOWN;
else {
/*把message加入到bitbang的等待队列中*/
list_add_tail(&m->queue, &bitbang->queue);
/*把bitbang-work加入bitbang->workqueue中，调度运行*/
queue_work(bitbang->workqueue, &bitbang->work);
}
spin_unlock_irqrestore(&bitbang->lock, flags);

return status;
}
EXPORT_SYMBOL_GPL(spi_bitbang_transfer);
```

分析工作队列的处理函数：




```
static void bitbang_work(struct work_struct *work)
{
struct spi_bitbang *bitbang =
container_of(work, struct spi_bitbang, work);
unsigned long flags;

spin_lock_irqsave(&bitbang->lock, flags);
/*设置成忙状态*/
bitbang->busy = 1;
/*对bitqueue中的每一个spi_message进行处理*/
while (!list_empty(&bitbang->queue)) {
struct spi_message *m;
struct spi_device *spi;
unsigned nsecs;
```

```
struct spi_transfer *t = NULL;
unsigned tmp;
unsigned cs_change;
int status;
int (*setup_transfer)(struct spi_device *,
struct spi_transfer *);

m = container_of(bitbang->queue.next, struct spi_message,
queue);
/*从队列中驱动这个spi_message*/
list_del_init(&m->queue);
spin_unlock_irqrestore(&bitbang->lock, flags);

nsecs = 100;

spi = m->spi;
tmp = 0;
cs_change = 1;
status = 0;
setup_transfer = NULL;
/*对spi_message的transfers上的每个spi_transfer进行处理*/
list_for_each_entry (t, &m->transfers, transfer_list) {
。。。。。。。。。。。。。。。。。
if (t->len) {
if (!m->is_dma_mapped)
t->rx_dma = t->tx_dma = 0;
/*调用bitbang->txrx_bufs进行数据的传输，bitbang->txrx_bufs =
s3c24xx_spi_txrx;这个在s3c24xx_spi_probe中进行赋值的*/
status = bitbang->txrx_bufs(spi, t);
}
。。。。。。。。。。。。。。。。

m->status = status;
/*传输完成，唤醒刚才的那个完成变量*/
m->complete(m->context);

/* restore speed and wordsize */
if (setup_transfer)
setup_transfer(spi, NULL);
if (!(status == 0 && cs_change)) {
ndelay(nsecs);
bitbang->chipselect(spi, BITBANG_CS_INACTIVE);
ndelay(nsecs);
}
```

```
spin_lock_irqsave(&bitbang->lock, flags);
}
bitbang->busy = 0;
spin_unlock_irqrestore(&bitbang->lock, flags);
}
```
这个工作队列的处理函数中调用了spi controller driver中的传输函数：

```
static int s3c24xx_spi_txrx(struct spi_device *spi, struct
spi_transfer *t)
{
struct s3c24xx_spi *hw = to_hw(spi);

dev_dbg(&spi->dev, "txrx: tx %p, rx %p, len %d\n",
t->tx_buf, t->rx_buf, t->len);

hw->tx = t->tx_buf; //发送指针
hw->rx = t->rx_buf; //接收指针
hw->len = t->len; //需要发送/接收的数目
hw->count = 0; //存放实际spi传输的数据数目
/*初始化了完成量*/
init_completion(&hw->done);

/*
*只需发送第一个字节(如果发送为空，则发送0xff)，中断中就会自动发送完
其他字节(并接受数据)
*直到所有数据发送完毕且所有数据接收完毕才返回
*/
writeb(hw_txbyte(hw, 0), hw->regs + S3C2410_SPTDAT);
/*等待完成量被唤醒*/
wait_for_completion(&hw->done);
return hw->count;
}
static inline unsigned int hw_txbyte(struct s3c24xx_spi *hw, int
count)
{
return hw->tx ? hw->tx[count] : 0xff;
//如果还有数据没接收完且要发送的数据经已发送完毕，发送空数据0xFF
}
```
下面来分析中断函数：
```
static irqreturn_t s3c24xx_spi_irq(int irq, void *dev)
{
struct s3c24xx_spi *hw = dev;
/*读取spi的状态寄存器*/
unsigned int spsta = readb(hw->regs + S3C2410_SPSTA);
```

```
unsigned int count = hw->count;
/*检测冲突*/
if (spsta & S3C2410_SPSTA_DCOL) {
dev_dbg(hw->dev, "data-collision\n");
/*唤醒完成量*/
complete(&hw->done);
goto irq_done;
}
/*设备忙*/
if (!(spsta & S3C2410_SPSTA_READY)) {
dev_dbg(hw->dev, "spi not ready for tx?\n");
/*唤醒完成量*/
complete(&hw->done);
goto irq_done;
}

hw->count++;
/*接收数据*/
if (hw->rx)
hw->rx[count] = readb(hw->regs + S3C2410_SPRDAT);

count++;
/*如果count小于需要发送或接收数据的数目，发送其他数据*/
if (count < hw->len)
writeb(hw_txbyte(hw, count), hw->regs + S3C2410_SPTDAT);
else
/*唤醒完成量，通知s3c24xx_spi_txrx函数*/
complete(&hw->done);

irq_done:
return IRQ_HANDLED;
}
```

至此spi数据传输过程完成，如果不想为自己的SPI设备写驱动，那么可以用Linux自带的spidev.c提供的驱动程序，只要在登记时，把设备名设置成spidev就可以了。spidev.c会在device目录下自动为每一个匹配的SPI设备创建设备节点，节点名"spi%d"。之后，用户程序可以通过字符型设备的通用接口控制SPI设备。需要注意的是，spidev创建的设备在设备模型中属于虚拟设备，他的class是spidev_class，他的父设备是在boardinfo中定义的spi设备。