专注于嵌入式 & Linux

以Kernel为中心，坚持学习各种资源建设。

Linux设备驱动剖析之SPI（三）

572至574行，分配内存，注意对象的类型是struct spidev_data，看下它在drivers/spi/spidev.c中的定义：

```
00000075 struct spidev_data {
00000076     dev_t           devt;
00000077     spinlock_t      spi_lock;
00000078     struct spi_device   *spi;
00000079     struct list_head    device_entry;
00000080
00000081     /* buffer is NULL unless this device is open (users > 0) */
00000082     struct mutex        buf_lock;
00000083     unsigned        users;
00000084     u8          *buffer;
00000085 };
```

76行，设备号。79行，设备链表，所有采用此驱动的设备将连成一个链表。83行，计数，也即是此设备被open的次数。

回到spidev_probe函数，577至586行，一些锁和链表的初始化。588行，从名字上就可以知道，就是找到第一个为0的位，第一个参数minors的定义：

```
00000054 #define N_SPI_MINORS            32    /* ... up to 256 */
00000055
00000056 static DECLARE_BITMAP(minors, N_SPI_MINORS);
```

DECLARE_BITMAP是一个宏，定义如下：

```
#define DECLARE_BITMAP(name,bits) \
    unsigned long name[BITS_TO_LONGS(bits)]
```

将宏展开后是这样的，unsigned long minors[1]，其实就是定义一个只有一个元素的无符号长整形数组miniors。

590至593行，如果找到了非0位，就将它作为次设备号与之前注册的主设备号生成设备号。

595至598行，创建设备，并生成设备节点，设备节点在/dev目录下，名字的形式为“spidevx.x”。

603至608行，创建设备成功后，将相应的位置1，表示该次设备号已经被使用，同时将该设备加入到设备链表。

611至614行，将设备的私有数据指针指向该设备。

至此，SPI设备驱动的初始化过程也说完了。下面就以应用程序的操作顺序来说，假设是从open-->write这个过程。下面先看驱动中open函数的实现，同样在drivers/spi/spidev.c：

```
00000477 static int spidev_open(struct inode *inode, struct file *filp)
00000478 {
00000479     struct spidev_data    *spidev;
00000480     int             status = -ENXIO;
00000481
00000482     mutex_lock(&device_list_lock);
00000483
00000484
00000485     list_for_each_entry(spidev, &device_list, device_entry) {
00000486         if (spidev->devt == inode->i_rdev) {
00000487             status = 0;
00000488             break;
00000489         }
00000490     }
00000491     if (status == 0) {
00000492         if (!spidev->buffer) {
00000493             spidev->buffer = kmalloc(bufsiz, GFP_KERNEL);
00000494             if (!spidev->buffer) {
00000495                 dev_dbg(&spidev->spi->dev, "open/ENOMEM\n");
00000496                 status = -ENOMEM;
00000497             }
```

```
00000498         }
00000499         if (status == 0) {
00000500             spidev->users++;
00000501             filp->private_data = spidev;
00000502             nonseekable_open(inode, filp);
00000503         }
00000504     } else
00000505         pr_debug("spidev: nothing for minor %d\n", iminor(inode));
00000506
00000507     mutex_unlock(&device_list_lock);
00000508     return status;
00000509 }
```

485至490行，遍历设备链表，每找到一个设备就将它的设备号与打开文件的设备号进行比较，相等的话表示查找成功。

491至505行，查找成功后就分配读写数据内存，使用计数加1，设置文件私有数据指针指向查找到的设备，以后在驱动的write、read函数里就可以把它取出来。

接下来是write函数的定义：

```
00000190 static ssize_t
00000191 spidev_write(struct file *filp, const char __user *buf,
00000192         size_t count, loff_t *f_pos)
00000193 {
00000194     struct spidev_data    *spidev;
00000195     ssize_t            status = 0;
00000196     unsigned long        missing;
00000197
00000198     /* chipselect only toggles at start or end of operation */
00000199     if (count > bufsiz)
00000200         return -EMSGSIZE;
00000201
00000202     spidev = filp->private_data;
00000203
00000204     mutex_lock(&spidev->buf_lock);
00000205     missing = copy_from_user(spidev->buffer, buf, count);
00000206     if (missing == 0) {
00000207         status = spidev_sync_write(spidev, count);
00000208     } else
00000209         status = -EFAULT;
00000210     mutex_unlock(&spidev->buf_lock);
00000211
00000212     return status;
00000213 }
```

199至200行，应用程序写入的数据不能大于驱动中缓冲区的大小，默认为4096个字节。

202行，指向文件的私有数据。

205行，拷贝用户空间的数据到内核空间。

207行，spidev_sync_write的定义：

```
00000130 static inline ssize_t
00000131 spidev_sync_write(struct spidev_data *spidev, size_t len)
00000132 {
00000133     struct spi_transfer    t = {
00000134             .tx_buf        = spidev->buffer,
00000135             .len        = len,
00000136         };
00000137     struct spi_message    m;
00000138
00000139     spi_message_init(&m);
00000140     spi_message_add_tail(&t, &m);
00000141     return spidev_sync(spidev, &m);
00000142 }
```

133行，struct spi_transfer的定义在include/linux/spi/spi.h：

```
00000427 struct spi_transfer {
00000428     /* it's ok if tx_buf == rx_buf (right?)
00000429      * for MicroWire, one buffer must be null
00000430      * buffers must work with dma_*map_single() calls, unless
00000431      *   spi_message.is_dma_mapped reports a pre-existing mapping
00000432      */
00000433     const void    *tx_buf;
00000434     void        *rx_buf;
00000435     unsigned    len;
00000436
00000437     dma_addr_t    tx_dma;
00000438     dma_addr_t    rx_dma;
00000439
00000440     unsigned    cs_change:1;
00000441     u8        bits_per_word;
00000442     u16        delay_usecs;
00000443     u32        speed_hz;
00000444
00000445     struct list_head transfer_list;
00000446 };
```

433至435行，发送、接收缓冲区和长度。437和438行，发送和接收的DMA地址。

440行，传输完成后是否改变片选信号。

441行，如果为0则使用驱动的默认值。

442行，传输完成后等待多长时间（毫秒）再改变片选信号。

443行，将多个传输连成一个链表。

回到spidev_sync_write函数的137行，在spi.h中定义的struct spi_message：

```
00000476 struct spi_message {
00000477     struct list_head    transfers;
00000478
00000479     struct spi_device    *spi;
00000480
00000481     unsigned        is_dma_mapped:1;
00000482
00000483     /* REVISIT:  we might want a flag affecting the behavior of the
00000484      * last transfer ... allowing things like "read 16 bit length L"
00000485      * immediately followed by "read L bytes".  Basically imposing
00000486      * a specific message scheduling algorithm.
00000487      *
00000488      * Some controller drivers (message-at-a-time queue processing)
00000489      * could provide that as their default scheduling algorithm.  But
00000490      * others (with multi-message pipelines) could need a flag to
00000491      * tell them about such special cases.
00000492      */
00000493
00000494     /* completion is reported through a callback */
00000495     void            (*complete)(void *context);
00000496     void            *context;
00000497     unsigned        actual_length;
00000498     int            status;
00000499
00000500     /* for optional use by whatever driver currently owns the
00000501      * spi_message ...  between calls to spi_async and then later
00000502      * complete(), that's the spi_master controller driver.
00000503      */
00000504     struct list_head    queue;
00000505     void            *state;
00000506 };
```

477行，一个message可能包含多个transfer，因此用链表将这些transfer连起来。

479行，这次message所使用的spi设备。

481行，是否采用DMA的标志。

495行，传输完成后的回调函数指针。496行，回调函数的参数。

497行，这次message成功传输的字节数。

504和505行，当前驱动拥有的message。

回到spidev_sync_write函数，139行，spi.h中的内联函数spi_message_init：

```
00000508 static inline void spi_message_init(struct spi_message *m)
00000509 {
00000510     memset(m, 0, sizeof *m);
00000511     INIT_LIST_HEAD(&m->transfers);
00000512 }
```

很简单，清0内存和初始化message的transfer链表。

140行，spi_message_add_tail也是spi.h中的内联函数：

```
00000514 static inline void
00000515 spi_message_add_tail(struct spi_transfer *t, struct spi_message *m)
00000516 {
00000517     list_add_tail(&t->transfer_list, &m->transfers);
00000518 }
```

将transfer加入到链表尾。

141行，spidev_sync函数是在drivers/spi/spidev.c中定义的：

```
00000105 static ssize_t
00000106 spidev_sync(struct spidev_data *spidev, struct spi_message *message)
00000107 {
00000108     DECLARE_COMPLETION_ONSTACK(done);
00000109     int status;
00000110
00000111     message->complete = spidev_complete;
00000112     message->context = &done;
```

```
00000113
00000114     spin_lock_irq(&spidev->spi_lock);
00000115     if (spidev->spi == NULL)
00000116         status = -ESHUTDOWN;
00000117     else
00000118         status = spi_async(spidev->spi, message);
00000119     spin_unlock_irq(&spidev->spi_lock);
00000120
00000121     if (status == 0) {
00000122         wait_for_completion(&done);
00000123         status = message->status;
00000124         if (status == 0)
00000125             status = message->actual_length;
00000126     }
00000127     return status;
00000128 }
```

108行，定义并初始化一个完成量，完成量是Linux的一种同步机制。

111行，spidev_complete函数里就用来唤醒等待completion，定义如下：

```
00000100 static void spidev_complete(void *arg)
00000101 {
00000102     complete(arg);
00000103 }
```

112行，作为spidev_complete函数的参数。

118行，调用drivers/spi/spi.c里的spi_async函数，从函数名知道，这是异步实现的。为什么是异步的？往下看就知道了。

```
00000737 int spi_async(struct spi_device *spi, struct spi_message *message)
00000738 {
00000739     struct spi_master *master = spi->master;
00000740     int ret;
00000741     unsigned long flags;
00000742
00000743     spin_lock_irqsave(&master->bus_lock_spinlock, flags);
00000744
00000745     if (master->bus_lock_flag)
00000746         ret = -EBUSY;
00000747     else
00000748         ret = __spi_async(spi, message);
00000749
00000750     spin_unlock_irqrestore(&master->bus_lock_spinlock, flags);
00000751
00000752     return ret;
00000753 }
```

745行，如果master所在的总线被锁住了，那么就返回忙。

748行，看__spi_async函数的定义：

```
00000679 static int __spi_async(struct spi_device *spi, struct spi_message *message)
00000680 {
00000681     struct spi_master *master = spi->master;
00000682
00000683     /* Half-duplex links include original MicroWire, and ones with
00000684      * only one data pin like SPI_3WIRE (switches direction) or where
00000685      * either MOSI or MISO is missing.  They can also be caused by
00000686      * software limitations.
00000687      */
00000688     if ((master->flags & SPI_MASTER_HALF_DUPLEX)
00000689             || (spi->mode & SPI_3WIRE)) {
00000690         struct spi_transfer *xfer;
00000691         unsigned flags = master->flags;
00000692
00000693         list_for_each_entry(xfer, &message->transfers, transfer_list) {
00000694             if (xfer->rx_buf && xfer->tx_buf)
00000695                 return -EINVAL;
00000696             if ((flags & SPI_MASTER_NO_TX) && xfer->tx_buf)
00000697                 return -EINVAL;
00000698             if ((flags & SPI_MASTER_NO_RX) && xfer->rx_buf)
00000699                 return -EINVAL;
00000700         }
00000701     }
00000702
00000703     message->spi = spi;
00000704     message->status = -EINPROGRESS;
00000705     return master->transfer(spi, message);
00000706 }
```

688至701行，如果master设置了SPI_MASTER_HALF_DUPLEX标志，或者spi设备使用的是3线模式，那么就对message里的每一个transfer的发送和接收buf做一些检查。

705行，调用的是具体的SPI控制器驱动里的函数，这里是drivers/spi/spi_s3c64xx.c里的s3c64xx_spi_transfer函数：

```
00000763 static int s3c64xx_spi_transfer(struct spi_device *spi,
00000764                             struct spi_message *msg)
00000765 {
```

```
00000766     struct s3c64xx_spi_driver_data *sdd;
00000767     unsigned long flags;
00000768
00000769     sdd = spi_master_get_devdata(spi->master);
00000770
00000771     spin_lock_irqsave(&sdd->lock, flags);
00000772
00000773     if (sdd->state & SUSPND) {
00000774         spin_unlock_irqrestore(&sdd->lock, flags);
00000775         return -ESHUTDOWN;
00000776     }
00000777
00000778     msg->status = -EINPROGRESS;
00000779     msg->actual_length = 0;
00000780
00000781     list_add_tail(&msg->queue, &sdd->queue);
00000782
00000783     queue_work(sdd->workqueue, &sdd->work);
00000784
00000785     spin_unlock_irqrestore(&sdd->lock, flags);
00000786
00000787     return 0;
00000788 }
```

标签: Linux驱动

好文要顶 关注我 收藏该文

lknlfy
关注 - 0
粉丝 - 188
+加关注
0
0
« 上一篇：Linux设备驱动剖析之SPI（二）
» 下一篇：Linux设备驱动剖析之SPI（四）
posted @ 2013-08-17 20:13 lknlfy 阅读(1802) 评论(0) 编辑 收藏

刷新评论刷新页面返回顶部
注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新IT新闻:
· Uber投资人的“警示信”：十年牛市或见顶，创业者准备提前过冬吧
· 硅谷巨头纷纷推防手机上瘾功能 但被指力度不足
· 微信“二次实名认证”实为骗局 警方：谨防上当
· 老干爹、阿里爸爸...大企业为何要“山寨”自家商标
· 夏普开始生产智能手机OLED显示屏 帮助苹果摆脱对三星的依赖
» 更多新闻...
最新知识库文章:
· 如何提高一个研发团队的“代码速度”？
· 成为一个有目标的学习者
· 历史转折中的“杭派工程师”
· 如何提高代码质量？
· 在腾讯的八年，我的职业思考
» 更多知识库文章...

公告

<2018年8月>
日一二三四五六
293031 1 2 3 4
5 6 7 8 9 1011
12131415161718
19202122232425
262728293031 1
2 3 4 5 6 7 8

搜索

找找看
谷歌搜索

常用链接

- 我的随笔
- 我的评论
- 我的参与
- 最新评论
- 我的标签
- 更多链接

我的标签

- Android开发(24)
- Linux驱动(15)
- ARM-Linux学习(7)
- Linux应用(7)
- BootLoader学习(7)
- Cubieboard2学习(7)
- ARM裸机开发(5)
- C语言(4)
- USB学习(3)
- 数据结构与算法(3)
- 更多

随笔档案(94)

- 2014年3月 (7)
- 2013年10月 (1)
- 2013年8月 (12)
- 2013年7月 (3)
- 2013年6月 (2)
- 2013年5月 (4)
- 2013年4月 (2)
- 2013年1月 (7)
- 2012年11月 (2)
- 2012年10月 (3)
- 2012年9月 (1)
- 2012年8月 (6)
- 2012年7月 (3)
- 2012年6月 (2)
- 2012年5月 (7)
- 2012年4月 (3)
- 2012年3月 (16)
- 2012年2月 (13)

积分与排名

- 积分 - 135080
- 排名 - 2392

最新评论

- 1. Re:Linux设备驱动剖析之Input（三）
- 很美
- --haoxing990
- 2. Re:Qt下libusb-win32的使用方法
- 我觉得所有问题都是都和驱动有关：
  1每次运行后显示程序异常，是因为没有安装驱动
  2访问不了鼠标等是因为只能访问安装了inf-wizard.exe生成的驱动程序的USB设备
- --蕾小蕾
- 3. Re:Qt下libusb-win32的使用方法
- 另外运行 inf-wizard.exe 时，需要管理员权限，否则可能会出现“System Policy has been modified to reject unsigned drivers”的错误
- --Beny
- 4. Re:Qt下libusb-win32的使用方法
- @史毅磊刚刚发现问题所在了，原来要先使用 libusb自带的inf-wizard.exe 工具先给你的usb安装驱动，再运行就没问题了。楼主文章有说了，但没注意，或许楼主可强调一下，引起注意@lknl.....
- --Beny
- 5. Re:Qt下libusb-win32的使用方法
- @史毅磊我也遇到同样问题，不知道你解决了没有？...
- --Beny

阅读排行榜

- 1. Android NDK开发（2）----- JNI多线程(20546)
- 2. Android应用开发提高篇（4）-----Socket编程（多线程、双向通信）(13240)
- 3. Android NDK开发（1）----- Java与C互相调用实例详解(9812)
- 4. Linux内存映射（mmap）(9257)
- 5. Android应用开发基础篇（16）-----ScaleGestureDetector（缩放手势检测）(8062)

评论排行榜

- 1. Barebox for Tiny6410(网卡驱动移植)(16)
- 2. Android NDK开发（1）----- Java与C互相调用实例详解(13)
- 3. Linux内存映射（mmap）(11)
- 4. Android应用开发提高篇（2）-----文本朗读TTS（TextToSpeech）(10)
- 5. Android应用开发基础篇（12）-----Socket通信(9)

推荐排行榜

- 1. Android NDK开发（1）----- Java与C互相调用实例详解(7)
- 2. 使用FFmpeg捕获一帧摄像头图像(3)
- 3. 从MACHINE_START开始(3)
- 4. Android应用开发提高篇（6）-----FaceDetector（人脸检测）(2)
- 5. Android应用开发基础篇（4）-----TabHost（选项卡）(2)