专注于嵌入式 & Linux

以Kernel为中心，坚持学习各种资源建设。

< 2018年8月 >

日	一	二	三	四	五	六
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	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)

博客园 首页 新随笔 联系 管理 订阅 XML 随笔- 94 文章- 0 评论- 179

Linux设备驱动剖析之SPI（一）

写在前面

初次接触SPI是因为几年前玩单片机的时候，由于普通的51单片机没有SPI控制器，所以只好用IO口去模拟。最近一次接触SPI是大三时参加的校内选拔赛，当时需要用2440去控制nrf24L01，从而实现数据的无线传输。nrf24L01是一种典型的SPI接口的2.4GHz无线收发器，当时搞了很久，由于时间比较紧，而且当时根本不熟悉Linux的SPI子系统，最后虽然采用IO口模拟SPI的方式勉强实现了，但是这根本就不符合Linux驱动的编程规范，或者说是在破坏Linux、污染kernel。

根据我个人所知道的，Linux SPI一直是处于被“忽略”的角色，市场上大部分板子在板级文件里都没有关于SPI的相关代码，而大部分讲驱动的书籍也没有专门的一章来讲述关于Linux SPI方面的内容，与IIC相比，SPI就是一个不被重视的“家伙”，为什么？我也不知道。为了帮SPI抱打不平，我决定基于Linux-2.6.36，说说Linux中SPI子系统。

先给出Linux SPI子系统的体系结构图：

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

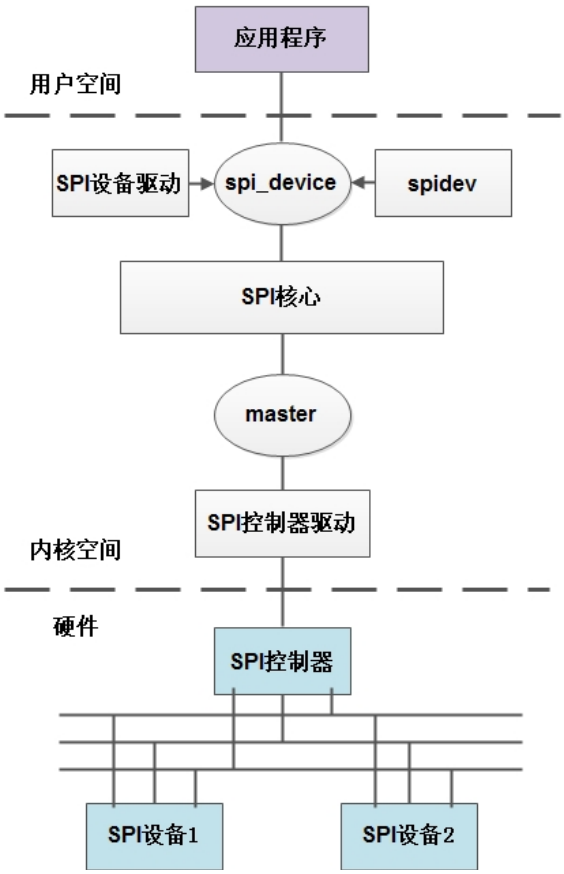
1. Re:Linux设备驱动剖析之Input（三）
很美
--haoxing990
2. Re:Qt下libusb-win32的使用方法
我觉得所有问题都是都和驱动有关：
1每次运行后显示程序异常，是因为没有安装驱动
2访问不了鼠标等是因为只能访问安装了inf-wizard.exe生成的驱动程序的USB设备
--蕾小蕾
3. Re:Qt下libusb-win32的使用方法
另外运行 inf-wizard.exe 时，需要管理员权限，否则可能会出现"System Policy has been modified to reject unsigned drivers"的错误
--Beny
4. Re:Qt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)



SPI子系统体系结构

下面开始分析SPI子系统。

Linux中SPI子系统的初始化是从drivers/spi/spi.c文件中的spi_init函数开始的，看看它的定义：

```
00001025 static int __init spi_init(void)
00001026 {
00001027     int    status;
00001028
00001029     buf = kmalloc(SPI_BUFSIZ, GFP_KERNEL);
00001030     if (!buf) {
00001031         status = -ENOMEM;
00001032         goto err0;
00001033     }
00001034
00001035     status = bus_register(&spi_bus_type);
00001036     if (status < 0)
00001037         goto err1;
00001038
00001039     status = class_register(&spi_master_class);
00001040     if (status < 0)
00001041         goto err2;
00001042     return 0;
00001043
00001044 err2:
00001045     bus_unregister(&spi_bus_type);
00001046 err1:
00001047     kfree(buf);
00001048     buf = NULL;
00001049 err0:
00001050     return status;
00001051 }
```

1029行，分配spi buf内存，其中buf和SPI_BUFSIZ都在spi.c文件中定义：

```
00000945 #define    SPI_BUFSIZ    max(32,SMP_CACHE_BYTES)
00000946
00000947 static u8    *buf;
```

1035行，注册spi总线，同样是在spi.c文件中：

```
00000145 struct bus_type spi_bus_type = {
00000146     .name        = "spi",
00000147     .dev_attrs    = spi_dev_attrs,
00000148     .match        = spi_match_device,
00000149     .uevent        = spi_uevent,
00000150     .suspend        = spi_suspend,
00000151     .resume        = spi_resume,
00000152 };
```

146行，总线的名字就叫spi。

148行，比较重要的，spi_match_device是spi总线上匹配设备和设备驱动的函数，同样是在spi.c文件中：

```
00000085 static int spi_match_device(struct device *dev, struct device_driver *drv)
00000086 {
00000087     const struct spi_device    *spi = to_spi_device(dev);
00000088     const struct spi_driver    *sdrv = to_spi_driver(drv);
00000089
00000090     /* Attempt an OF style match */
00000091     if (of_driver_match_device(dev, drv))
00000092         return 1;
00000093
00000094     if (sdrv->id_table)
00000095         return !!spi_match_id(sdrv->id_table, spi);
00000096
00000097     return strcmp(spi->modalias, drv->name) == 0;
00000098 }
```

写过驱动的都应该知道platform总线有struct platform_device和struct platform_driver，到了SPI总线，当然也有对应的struct spi_device和struct spi_driver，如87、88行所示。87行，关于struct spi_device的定义是在include/linux/spi/spi.h中：

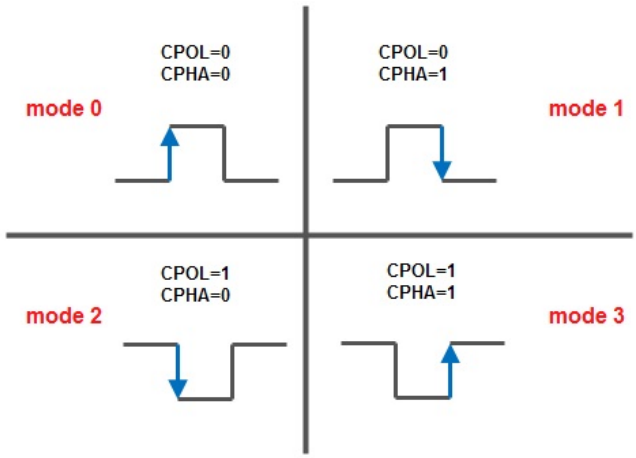
```
00000069 struct spi_device {
00000070     struct device        dev;
00000071     struct spi_master    *master;
00000072     u32            max_speed_hz;
00000073     u8            chip_select;
00000074     u8            mode;
00000075 #define    SPI_CPHA    0x01            /* clock phase */
00000076 #define    SPI_CPOL    0x02            /* clock polarity */
00000077 #define    SPI_MODE_0    (0|0)            /* (original MicroWire) */
00000078 #define    SPI_MODE_1    (0|SPI_CPHA)
00000079 #define    SPI_MODE_2    (SPI_CPOL|0)
00000080 #define    SPI_MODE_3    (SPI_CPOL|SPI_CPHA)
00000081 #define    SPI_CS_HIGH    0x04            /* chipselect active high? */
00000082 #define    SPI_LSB_FIRST    0x08            /* per-word bits-on-wire */
00000083 #define    SPI_3WIRE    0x10            /* SI/SO signals shared */
00000084 #define    SPI_LOOP    0x20            /* loopback mode */
00000085 #define    SPI_NO_CS    0x40            /* 1 dev/bus, no chipselect */
00000086 #define    SPI_READY    0x80            /* slave pulls low to pause */
00000087     u8            bits_per_word;
00000088     int            irq;
00000089     void            *controller_state;
00000090     void            *controller_data;
00000091     char            modalias[SPI_NAME_SIZE];
00000092
00000093     /*
00000094      * likely need more hooks for more protocol options affecting how
00000095      * the controller talks to each chip, like:
00000096      *  - memory packing (12 bit samples into low bits, others zeroed)
00000097      *  - priority
00000098      *  - drop chipselect after each word
00000099      *  - chipselect delays
00000100      *  - ...
00000101      */
00000102 };
```

70行，dev，嵌入到设备模型中用的。

71行，master，spi设备的更高层描述，每一个spi控制器就对应一个master，一个spi设备必须对应一个master，master下可以有多个spi设备。

72，73行没什么好说的，从变量的名字就可以明白。

74行，mode，针对时钟相位CPHA（0或1）和时钟极性CPOL（0或1）的不同组合，将spi分成四种模式，就是77至80行这四种。CPOL表示当空闲时（没有进行数据传输）时钟信号的电平，CPOL=0表示低电平，CPOL=1表示高电平。每个时钟周期都有两次电平的跳变，上升沿和下降沿，CPHA就表示在每个时钟周期里是第一个边沿采样数据还是第二个边沿采样数据，具体第一个边沿或者第二个边沿是上升沿还是下降沿则由CPOL决定。看下面这张图就明白了。蓝色箭头表示对数据进行采样。



87行，如果传输是以字节为单位的话就设置为8，相应地，如果是以2个字节为单位则设置为16。

88行，中断号。89行，没有使用，在后面会看到这个值会被设置为NULL。90行，后面讲到具体的控制器再说。91行，很重要，一般来说设备与驱动能否匹配起来就要看它，注意，这里只是说一般，因为还有另外两种匹配的方法，不过大部分设备和驱动能否匹配起来都是根据名字是否相等，当然像USB驱动就不是根据名字来匹配的，而是根据与id table。struct spi_driver的定义也是在include/linux/spi/spi.h：

```
00000175 struct spi_driver {
00000176     const struct spi_device_id *id_table;
00000177     int            (*probe)(struct spi_device *spi);
00000178     int            (*remove)(struct spi_device *spi);
00000179     void            (*shutdown)(struct spi_device *spi);
00000180     int            (*suspend)(struct spi_device *spi, pm_message_t mesg);
00000181     int            (*resume)(struct spi_device *spi);
00000182     struct device_driver    driver;
00000183 };
```

182行，driver就是在设备模型中使用的那个device_driver，其他都是一些函数指针的定义，挺熟悉的了，就不多说了。

回到spi_match_device函数，91行和95行就是设备和驱动匹配的另外两种方法，因为后文要讲的spi驱动使用的是第三种方法，因此这里就不讨论这两种方法了。97行，根据设备名和驱动名是否相等进行匹配，相等则返回1，表示匹配成功，此时驱动里的probe函数将会被调用，这也是我们最希望看到的，返回0则表示匹配失败。

我们知道，对于具体的平台，nand、iic、frame buffer等这些都是平台设备，spi当然也一样是平台设备，对于平台设备，大部分情况下是先注册设备再注册驱动。因此下面就以tiny6410为具体平台，按照这种先后顺序来讲述。

S3c6410有两个SPI控制器，以SPI0为例就可以了。首先看s3c6410中关于SPI0控制器的描述，在arch/arm/mach-s3c64xx/dev-spi.c文件中：

```
00000101 struct platform_device s3c64xx_device_spi0 = {
00000102     .name          = "s3c64xx-spi",
00000103     .id          = 0,
00000104     .num_resources      = ARRAY_SIZE(s3c64xx_spi0_resource),
00000105     .resource      = s3c64xx_spi0_resource,
00000106     .dev = {
00000107         .dma_mask        = &spi_dmamask,
00000108         .coherent_dma_mask    = DMA_BIT_MASK(32),
00000109         .platform_data = &s3c64xx_spi0_pdata,
00000110     },
00000111 };
```

102行，驱动能否与这个设备匹配，就看这个名字了，因此对应的驱动名字必须与之一样。103行，SPI控制器的ID，SPI0控制器就为0，SPI1控制器就为1。

104和105行是关于IO口资源、DMA资源和中断资源的。107、108行是关于DMA的，不说了。109行，给驱动用的，在驱动那里再说。在板初始化函数mini6410_machine_init里调用platform_add_devices函数就可以将SPI0设备注册到platform总线上。

S3c6410的SPI控制器驱动在drivers/spi/spi_s3c64xx.c文件里。初始化函数：

```
00001183 static int __init s3c64xx_spi_init(void)
00001184 {
00001185     return platform_driver_probe(&s3c64xx_spi_driver, s3c64xx_spi_probe);
00001186 }
```

1185行，s3c64xx_spi_driver是struct platform_driver的实例，也在spi_s3c64xx.c文件中定义：

```
00001172 static struct platform_driver s3c64xx_spi_driver = {
00001173     .driver = {
00001174         .name    = "s3c64xx-spi",
00001175         .owner = THIS_MODULE,
00001176     },
00001177     .remove = s3c64xx_spi_remove,
00001178     .suspend = s3c64xx_spi_suspend,
00001179     .resume = s3c64xx_spi_resume,
00001180 };
```

1174行，看到了没？和之前在s3c64xx_device_spi0里定义的名字是一样的，这样它们就可以匹配起来，1185行的s3c64xx_spi_probe驱动探测函数就会被调用，看下它的定义：

```
00000911 static int __init s3c64xx_spi_probe(struct platform_device *pdev)
00000912 {
00000913     struct resource    *mem_res, *dmatx_res, *dmarx_res;
00000914     struct s3c64xx_spi_driver_data *sdd;
00000915     struct s3c64xx_spi_info *sci;
00000916     struct spi_master *master;
00000917     int ret;
00000918
00000919     if (pdev->id < 0) {
00000920         dev_err(&pdev->dev,
00000921                 "Invalid platform device id-%d\n", pdev->id);
00000922         return -ENODEV;
00000923     }
00000924
00000925     if (pdev->dev.platform_data == NULL) {
00000926         dev_err(&pdev->dev, "platform_data missing!\n");
00000927         return -ENODEV;
00000928     }
00000929
00000930     sci = pdev->dev.platform_data;
00000931     if (!sci->src_clk_name) {
00000932         dev_err(&pdev->dev,
00000933             "Board init must call s3c64xx_spi_set_info()\n");
00000934         return -EINVAL;
00000935     }
00000936
00000937     /* Check for availability of necessary resource */
00000938
00000939     dmatx_res = platform_get_resource(pdev, IORESOURCE_DMA, 0);
```

```
00000940     if (dmatx_res == NULL) {
00000941         dev_err(&pdev->dev, "Unable to get SPI-Tx dma resource\n");
00000942         return -ENXIO;
00000943     }
00000944
00000945     dmarx_res = platform_get_resource(pdev, IORESOURCE_DMA, 1);
00000946     if (dmarx_res == NULL) {
00000947         dev_err(&pdev->dev, "Unable to get SPI-Rx dma resource\n");
00000948         return -ENXIO;
00000949     }
00000950
00000951     mem_res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
00000952     if (mem_res == NULL) {
00000953         dev_err(&pdev->dev, "Unable to get SPI MEM resource\n");
00000954         return -ENXIO;
00000955     }
00000956
00000957     master = spi_alloc_master(&pdev->dev,
00000958                 sizeof(struct s3c64xx_spi_driver_data));
00000959     if (master == NULL) {
00000960         dev_err(&pdev->dev, "Unable to allocate SPI Master\n");
00000961         return -ENOMEM;
00000962     }
00000963
00000964     platform_set_drvdata(pdev, master);
00000965
00000966     sdd = spi_master_get_devdata(master);
00000967     sdd->master = master;
00000968     sdd->cntrlr_info = sci;
00000969     sdd->pdev = pdev;
00000970     sdd->sfr_start = mem_res->start;
00000971     sdd->tx_dmach = dmatx_res->start;
00000972     sdd->rx_dmach = dmarx_res->start;
00000973
00000974     sdd->cur_bpw = 8;
00000975
00000976     master->bus_num = pdev->id;
00000977     master->setup = s3c64xx_spi_setup;
00000978     master->transfer = s3c64xx_spi_transfer;
00000979     master->num_chipselect = sci->num_cs;
00000980     master->dma_alignment = 8;
00000981     /* the spi->mode bits understood by this driver: */
00000982
00000983     master->mode_bits = SPI_CPOL | SPI_CPHA | SPI_CS_HIGH;
00000984
00000985     if (request_mem_region(mem_res->start,
00000986             resource_size(mem_res), pdev->name) == NULL) {
00000987         dev_err(&pdev->dev, "Req mem region failed\n");
00000988         ret = -ENXIO;
00000989         goto err0;
00000990     }
00000991
00000992     sdd->regs = ioremap(mem_res->start, resource_size(mem_res));
00000993     if (sdd->regs == NULL) {
00000994         dev_err(&pdev->dev, "Unable to remap IO\n");
00000995         ret = -ENXIO;
00000996         goto err1;
00000997     }
00000998
00000999     if (sci->cfg_gpio == NULL || sci->cfg_gpio(pdev)) {
00001000         dev_err(&pdev->dev, "Unable to config gpio\n");
00001001         ret = -EBUSY;
00001002         goto err2;
00001003     }
00001004
00001005     /* Setup clocks */
00001006     sdd->clk = clk_get(&pdev->dev, "spi");
00001007     if (IS_ERR(sdd->clk)) {
00001008         dev_err(&pdev->dev, "Unable to acquire clock 'spi'\n");
00001009         ret = PTR_ERR(sdd->clk);
00001010         goto err3;
00001011     }
00001012
00001013     if (clk_enable(sdd->clk)) {
00001014         dev_err(&pdev->dev, "Couldn't enable clock 'spi'\n");
00001015         ret = -EBUSY;
00001016         goto err4;
00001017     }
00001018
```

```
00001019     sdd->src_clk = clk_get(&pdev->dev, sci->src_clk_name);
00001020     if (IS_ERR(sdd->src_clk)) {
00001021         dev_err(&pdev->dev,
00001022             "Unable to acquire clock '%s'\n", sci->src_clk_name);
00001023         ret = PTR_ERR(sdd->src_clk);
00001024         goto err5;
00001025     }
00001026
00001027     if (clk_enable(sdd->src_clk)) {
00001028         dev_err(&pdev->dev, "Couldn't enable clock '%s'\n",
00001029                             sci->src_clk_name);
00001030         ret = -EBUSY;
00001031         goto err6;
00001032     }
00001033
00001034     sdd->workqueue = create_singlethread_workqueue(
00001035                         dev_name(master->dev.parent));
00001036     if (sdd->workqueue == NULL) {
00001037         dev_err(&pdev->dev, "Unable to create workqueue\n");
00001038         ret = -ENOMEM;
00001039         goto err7;
00001040     }
00001041
00001042     /* Setup Deufult Mode */
00001043     s3c64xx_spi_hwinit(sdd, pdev->id);
00001044
00001045     spin_lock_init(&sdd->lock);
00001046     init_completion(&sdd->xfer_completion);
00001047     INIT_WORK(&sdd->work, s3c64xx_spi_work);
00001048     INIT_LIST_HEAD(&sdd->queue);
00001049
00001050     if (spi_register_master(master)) {
00001051         dev_err(&pdev->dev, "cannot register SPI master\n");
00001052         ret = -EBUSY;
00001053         goto err8;
00001054     }
00001055
00001056     dev_dbg(&pdev->dev, "Samsung SoC SPI Driver loaded for Bus SPI-%d "
00001057                         "with %d Slaves attached\n",
00001058                         pdev->id, master->num_chipselect);
00001059     dev_dbg(&pdev->dev, "\tIOmem=[0x%x-0x%x]\tDMA=[Rx-%d, Tx-%d]\n",
00001060                         mem_res->end, mem_res->start,
00001061                         sdd->rx_dmach, sdd->tx_dmach);
00001062
00001063     return 0;
00001064
00001065 err8:
00001066     destroy_workqueue(sdd->workqueue);
00001067 err7:
00001068     clk_disable(sdd->src_clk);
00001069 err6:
00001070     clk_put(sdd->src_clk);
00001071 err5:
00001072     clk_disable(sdd->clk);
00001073 err4:
00001074     clk_put(sdd->clk);
00001075 err3:
00001076 err2:
00001077     iounmap((void *) sdd->regs);
00001078 err1:
00001079     release_mem_region(mem_res->start, resource_size(mem_res));
00001080 err0:
00001081     platform_set_drvdata(pdev, NULL);
00001082     spi_master_put(master);
00001083
00001084     return ret;
00001085 }
```

函数很长，但做的东西却很简单。919至923行，SPI控制器的ID是从0开始的，小于0的话，没门，出错。

925至928行，必须要有platform_data，否则出错。930行，如果platform_data存在的话就把它取出来。

931至935行，如果src_clk_name为0，则表示在板初始化函数里没有调用s3c64xx_spi_set_info函数。

939至955行，获取在设备里定义的IO口和DMA资源。

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

1 0

« 上一篇：Qt下libusb-win32的使用(二)批量读写操作
» 下一篇：Linux设备驱动剖析之SPI（二）

posted @ 2013-08-17 19:45 lknlfy 阅读(4643) 评论(0) 编辑 收藏