# 5. Linux SPI总线和设备驱动架构

http://blog.csdn.net/DroidPhone/article/details/23367051

## 5.1 SPI 概述

SPI是"Serial Peripheral Interface" 的缩写，是一种四线制的同步串行通信接口，用来连接微控制器、传感器、存储设备，SPI设备分为主设备和从设备两种，用于通信和控制的四根线分别是：

CS    片选信号

SCK  时钟信号

MISO  主设备的数据输入、从设备的数据输出脚

MOSI  主设备的数据输出、从设备的数据输入脚

因为在大多数情况下，CPU或SOC一侧通常都是工作在主设备模式，所以，目前的Linux内核版本中，只实现了主模式的驱动框架。

### 5.1.1硬件结构

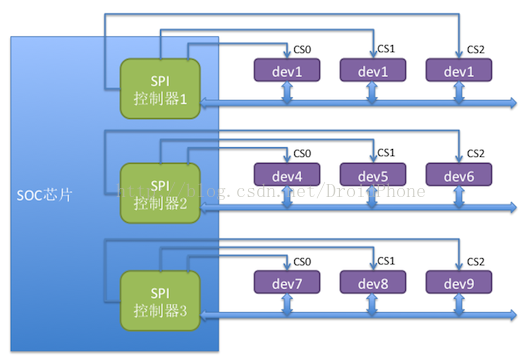


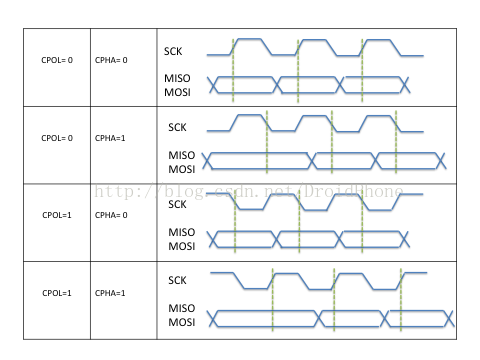
 图1.1    SPI硬件结构图

主设备对应SOC芯片中的SPI控制器，通常，一个SOC中可能存在多个SPI控制器，像上面的例子所示，SOC芯片中有3个SPI控制器。每个控制器下可以连接多个SPI从设备，每个从设备有各自独立的CS引脚。每个从设备共享另外3个信号引脚：SCK、MISO、MOSI。任何时刻，只有一个CS引脚处于有效状态，与该有效CS引脚连接的设备此时可以与主设备（SPI控制器）通信，其它的从设备处于等待状态，并且它们的3个引脚必须处于高阻状态。

通常，负责发出时钟信号的设备我们称之为主设备，另一方则作为从设备.

### 5.1.1.1工作时序

按照时钟信号和数据信号之间的相位关系，SPI有4种工作时序模式：

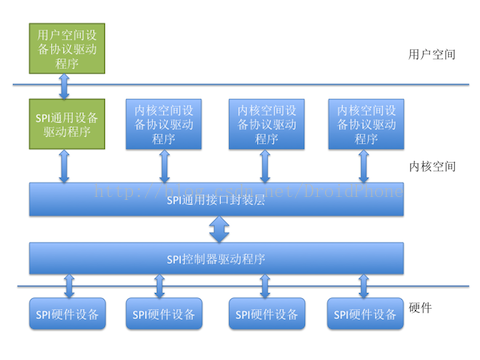


用CPOL表示时钟信号的初始电平的状态，CPOL为0表示时钟信号初始状态为低电平，为1表示时钟信号的初始电平是高电平。另外，我们用CPHA来表示在那个时钟沿采样数据，CPHA为0表示在首个时钟变化沿采样数据，而CPHA为1则表示要在第二个时钟变化沿来采样数据。内核用CPOL和CPHA的组合来表示当前SPI需要的工作模式：

* CPOL＝0，CPHA＝1        模式0
* CPOL＝0，CPHA＝1        模式1
* CPOL＝1，CPHA＝0        模式2
* CPOL＝1，CPHA＝1        模式3

## 5.1.2软件架构

在内核的SPI驱动的软件架构中，进行了合理的分层和抽象，如下图所示：



### 5.1.2.1 SPI控制器驱动程序

SPI控制器不用关心设备的具体功能，它只负责把上层协议驱动准备好的数据按SPI总线的时序要求发送给SPI设备，同时把从设备收到的数据返回给上层的协议驱动，因此，内核把SPI控制器的驱动程序独立出来。SPI控制器驱动负责控制具体的控制器硬件，诸如DMA和中断操作等等，因为多个上层的协议驱动可能会通过控制器请求数据传输操作，所以，SPI控制器驱动同时也要负责对这些请求进行队列管理，保证先进先出的原则。

### 5.1.2.2 SPI通用接口封装层

为了简化SPI驱动程序的编程工作，同时也为了降低协议驱动程序和控制器驱动程序的耦合程度，内核把控制器驱动和协议驱动的一些通用操作封装成标准的接口，加上一些通用的逻辑处理操作，组成了SPI通用接口封装层。这样的好处是，对于控制器驱动程序，只要实现标准的接口回调API，并把它注册到通用接口层即可，无需直接和协议层驱动程序进行交互。而对于协议层驱动来说，只需通过通用接口层提供的API即可完成设备和驱动的注册，并通过通用接口层的API完成数据的传输，无需关注SPI控制器驱动的实现细节。

### 5.1.2.3 [SPI协议驱动程序](http://blog.csdn.net/droidphone/article/details/23367051/#t5)

上面我们提到，控制器驱动程序并不清楚和关注设备的具体功能，SPI设备的具体功能是由SPI协议驱动程序完成的，SPI协议驱动程序了解设备的功能和通信数据的协议格式。向下，协议驱动通过通用接口层和控制器交换数据，向上，协议驱动通常会根据设备具体的功能和内核的其它子系统进行交互，例如，和MTD层交互以便把SPI接口的存储设备实现为某个文件系统，和TTY子系统交互把SPI设备实现为一个TTY设备，和网络子系统交互以便把一个SPI设备实现为一个网络设备，等等。当然，如果是一个专有的SPI设备，我们也可以按设备的协议要求，实现自己的专有协议驱动。

### 5.1.2.4 [SPI通用设备驱动程序](http://blog.csdn.net/droidphone/article/details/23367051/#t6)

有时候，考虑到连接在SPI控制器上的设备的可变性，在内核没有配备相应的协议驱动程序，对于这种情况，内核为我们准备了通用的SPI设备驱动程序，该通用设备驱动程序向用户空间提供了控制SPI控制的控制接口，具体的协议控制和数据传输工作交由用户空间根据具体的设备来完成，在这种方式中，只能采用同步的方式和SPI设备进行通信，所以通常用于一些数据量较少的简单SPI设备。

## 5.2 [SPI通用接口层](http://blog.csdn.net/droidphone/article/details/23932447)

SPI通用接口层用于把具体SPI设备的协议驱动和SPI控制器驱动联接在一起，通用接口层除了为协议驱动和控制器驱动提供一系列的标准接口API，同时还为这些接口API定义了相应的数据结构，这些数据结构一部分是SPI设备、SPI协议驱动和SPI控制器的数据抽象，一部分是为了协助数据传输而定义的数据结构。另外，通用接口层还负责SPI系统与Linux设备模型相关的初始化工作。本章的我们就通过这些数据结构和API的讨论来对整个通用接口层进行深入的了解。

SPI通用接口层的代码集中在：/drivers/spi/spi.c中。

### 5.2.1 SPI设备模型的初始化

通常地，根据linux设备模型的组织方式，各种设备会挂在合适的总线上，设备驱动和设备通过总线互相进行匹配，使得设备能够找到正确的驱动程序进行控制和驱动。同时，性质相似的设备可以归为某一个类的设备，它们具有某些共同的设备属性，在设备模型上就是所谓的class。SPI设备也不例外，它们也遵循linux的设备模型的规则：

**struct** bus\_type spi\_bus\_type = {

        .name           = "spi",

        .dev\_attrs      = spi\_dev\_attrs,

        .match          = spi\_match\_device,

        .uevent         = spi\_uevent,

        .pm             = &spi\_pm,

};

**static** **struct** **class** spi\_master\_class = {

        .name           = "spi\_master",

        .owner          = THIS\_MODULE,

        .dev\_release    = spi\_master\_release,

};

**static** **int** \_\_init spi\_init(**void**)

{

**int**     status;

        buf = kmalloc(SPI\_BUFSIZ, GFP\_KERNEL);

        ......

        status = bus\_register(&spi\_bus\_type);

        ......

        status = class\_register(&spi\_master\_class);

        ......

**return** 0;

        ......

}

postcore\_initcall(spi\_init);

可见，在初始化阶段，spi\_init函数向系统注册了一个名为spi的总线类型，同时也为SPI控制器注册了一个名为spi\_master的设备类。这样，以后在sysfs中就会出现以下两个文件节点：

sys/bus/spi

sys/class/spi\_master

代表spi总线的spi\_bus\_type结构的match字段指向了spi\_match\_device函数，该函数用于匹配spi总线上的设备和驱动，具体的代码这里就不贴了，各位可以自行查看内核的代码树。

### 5.2.2 [spi\_master结构](http://blog.csdn.net/droidphone/article/details/23932447#t1)

SPI控制器负责按照设定的物理信号格式在主控和spi设备之间交换数据，SPI控制器数据是如何被传输的，而不关心数据的内容。SPI通用接口层用spi\_master结构来表示一个spi控制器，我们看看它的主要字段的意义：

|  |  |
| --- | --- |
| 字段名称 | 描述 |
| struct device   dev | spi控制器对应的device结构 |
| struct list\_head list | 系统可能有多个控制器，用该链表链接在一个全局链表变量上 |
| s16             bus\_num | 该控制器对应的spi总线编号，从0开始，通常由板级代码设定 |
| u16             num\_chipselect | 连接到该spi控制器的片选信号的个数 |
| u16             mode\_bits | 工作模式，由驱动解释该模式的意义 |
| u32             min\_speed\_hz | 最低工作时钟 |
| u32             max\_speed\_hz | 最高工作时钟 |
| u16             flags | 用于设定某些限制条件的标志位 |
| int             (\*setup)(struct spi\_device \*spi) | 回调函数，用于设置某个spi设备在该控制器上的工作参数 |
| int             (\*transfer)(......） | 回调函数，用于把包含数据信息的mesg结构加入控制器的消息链表中 |
| void           (\*cleanup)(struct spi\_device \*spi) | 回调函数，当spi\_master被释放时，该函数被调用 |
| struct kthread\_worker   kworker | 用于管理数据传输消息队列的工作队列线程 |
| struct kthread\_work     pump\_messages | 具体实现数据传输队列的工作队列 |
| struct list\_head        queue | 该控制器的消息队列，所有等待传输的消息队列挂在该链表下 |
| struct spi\_message      \*cur\_msg | 当前正带处理的消息队列 |
| int (\*prepare\_transfer\_hardware)(......) | 回调函数，正式发起传送前会被调用，用于准备硬件资源 |
| int (\*transfer\_one\_message)(......) | 单个消息的原子传送回调函数，队列中的每个消息都会调用一次该回调来完成传输工作 |
| int (\*unprepare\_transfer\_hardware)(......) | 清理回调函数 |
| int                     \*cs\_gpios | 片选信号所用到的gpio |

spi\_master结构通常由控制器驱动定义，然后通过以下通用接口层的API注册到系统中：

int spi\_register\_master(struct spi\_master \*master);

### 5.2.3 [spi\_device结构](http://blog.csdn.net/droidphone/article/details/23932447#t2)

SPI通用接口层用spi\_device结构来表示一个spi设备，它的各个字段的意义如下：

|  |  |
| --- | --- |
| struct device           dev | 代表该spi设备的device结构 |
| struct spi\_master       \*master | 指向该spi设备所使用的控制器 |
| u32     max\_speed\_hz | 该设备的最大工作时钟频率 |
| u8      chip\_select | 在控制器中的片选引脚编号索引 |
| u16     mode | 设备的工作模式，包括时钟格式，片选信号的有效电平等等 |
| u8      bits\_per\_word | 设备每个单位数据所需要的比特数 |
| int     irq | 设备使用的irq编号 |
| char    modalias[SPI\_NAME\_SIZE] | 该设备的名字，用于spi总线和驱动进行配对 |
| int     cs\_gpio | 片选信号的gpio编号，通常不用我们自己设置，接口层会根据上面的chip\_select字段在spi\_master结构中进行查找并赋值 |

要完成向系统增加并注册一个SPI设备，我们还需要另一个数据结构：

**struct** spi\_board\_info {

**char**            modalias[SPI\_NAME\_SIZE];

**const** **void**      \*platform\_data;

**void**            \*controller\_data;

**int**             irq;

        u32             max\_speed\_hz;

        u16             bus\_num;

        u16             chip\_select;

        u16             mode;

};

spi\_board\_info结构大部分字段和spi\_device结构相对应，bus\_num字段则用来指定所属的控制器编号，通过spi\_board\_info结构，我们可以有两种方式向系统增加spi设备。第一种方式是在SPI控制器驱动已经被加载后，我们使用通用接口层提供的如下API来完成：

struct spi\_device \*spi\_new\_device(struct spi\_master \*master, struct spi\_board\_info \*chip);

第二种方式是在板子的初始化代码中，定义一个spi\_board\_info数组，然后通过以下API注册spi\_board\_info：

int spi\_register\_board\_info(struct spi\_board\_info const \*info, unsigned n);

上面这个API会把每个spi\_board\_info挂在全局链表变量board\_list上，并且遍历已经在系统中注册了的控制器，匹配上相应的控制器并取得它们的spi\_master结构指针，最终也会通过spi\_new\_device函数添加SPI设备。因为spi\_register\_board\_info可以在板子的初始化代码中调用，可能这时控制器驱动尚未加载，此刻无法取得相应的spi\_master指针，不过不要担心，控制器驱动被加载时，一定会调用spi\_register\_master函数来注册spi\_master结构，而spi\_register\_master函数会反过来遍历全局链表board\_list上的spi\_board\_info，然后通过spi\_new\_device函数添加SPI设备。

int \_\_init

spi\_register\_board\_info(struct spi\_board\_info const \*info, unsigned n)

{

    struct boardinfo \*bi;

    int i;

    bi = kzalloc(n \* sizeof(\*bi), GFP\_KERNEL);

    if (!bi)

        return -ENOMEM;

    for (i = 0; i < n; i++, bi++, info++) {

        struct spi\_master \*master;

        memcpy(&bi->board\_info, info, sizeof(\*info));

        mutex\_lock(&board\_lock);

        list\_add\_tail(&bi->list, &board\_list);

        list\_for\_each\_entry(master, &spi\_master\_list, list)

            spi\_match\_master\_to\_boardinfo(master, &bi->board\_info);

        mutex\_unlock(&board\_lock);

    }

    return 0;

}

对于此处，n为1，在程序中首先创建相应的内存，在for循环中，将信息保存到内存中，然后插入board\_list链表，接着遍历spi\_master\_list链表，注意此处，由于device\_initcall的优先级高于module\_init，所以此时spi\_master\_list链表为空，那么还不能调用spi\_match\_master\_to\_boardinfo函数创建spi设备，具体的创建设备将在spi总线驱动的探测函数中，使用spi\_register\_master()函数创建设备。

### 5.2.4 [spi\_driver结构](http://blog.csdn.net/droidphone/article/details/23932447#t3)

根据linux的设备模型，有device就必定有driver与之对应，上一节介绍的spi\_device结构中内嵌了device结构字段dev，同样地，代表驱动程序的spi\_driver结构也内嵌了device\_driver结构：

struct spi\_driver {

        const struct spi\_device\_id \*id\_table;

        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;

};

id\_table字段用于指定该驱动可以驱动的设备名称，总线的匹配函数会把id\_table中指定的名字和spi\_device结构中的modalias字段进行比较，两者相符即表示匹配成功，然后出发spi\_driver的probe回调函数被调用，从而完成驱动程序的初始化工作。通用接口层提供如下API来完成spi\_driver的注册：

**int** spi\_register\_driver(**struct** spi\_driver \*sdrv)

{

        sdrv->driver.bus = &spi\_bus\_type;

**if** (sdrv->probe)

                sdrv->driver.probe = spi\_drv\_probe;

**if** (sdrv->remove)

                sdrv->driver.remove = spi\_drv\_remove;

**if** (sdrv->shutdown)

                sdrv->driver.shutdown = spi\_drv\_shutdown;

**return** driver\_register(&sdrv->driver);

}

需要注意的是，这里的spi\_driver结构代表的是具体的SPI协议驱动程序。

### 5.2.5 spi\_message和spi\_transfer结构

要完成和SPI设备的数据传输工作，我们还需要另外两个数据结构：spi\_message和spi\_transfer。spi\_message包含了一个的spi\_transfer结构序列，一旦控制器接收了一个spi\_message，其中的spi\_transfer应该按顺序被发送，并且不能被其它spi\_message打断，所以我们认为spi\_message就是一次SPI数据交换的原子操作。下面我们看看这两个数据结构的定义：

**struct** spi\_message {

**struct** list\_head        transfers;

**struct** spi\_device       \*spi;

        unsigned                is\_dma\_mapped:1;

        /\* completion is reported through a callback \*/

**void**                    (\*complete)(**void** \*context);

**void**                    \*context;

        unsigned                frame\_length;

        unsigned                actual\_length;

**int**                     status;

**struct** list\_head        queue;

**void**                    \*state;

};

链表字段queue用于把该结构挂在代表控制器的spi\_master结构的queue字段上，控制器上可以同时被加入多个spi\_message进行排队。另一个链表字段transfers则用于链接挂在本message下的spi\_tranfer结构。complete回调函数则会在该message下的所有spi\_transfer都被传输完成时被调用，以便通知协议驱动处理接收到的数据以及准备下一批需要发送的数据。我们再来看看spi\_transfer结构：

**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              tx\_nbits;

        u8              rx\_nbits;

        u8              bits\_per\_word;

        u16             delay\_usecs;

        u32             speed\_hz;

**struct** list\_head transfer\_list;

};

首先，transfer\_list链表字段用于把该transfer挂在一个spi\_message结构中，tx\_buf和rx\_buf提供了非dma模式下的数据缓冲区地址，len则是需要传输数据的长度，tx\_dma和rx\_dma则给出了dma模式下的缓冲区地址。原则来讲，spi\_transfer才是传输的最小单位，之所以又引进了spi\_message进行打包，我觉得原因是：有时候希望往spi设备的多个不连续的地址（或寄存器）一次性写入，如果没有spi\_message进行把这样的多个spi\_transfer打包，因为通常真正的数据传送工作是在另一个内核线程（工作队列）中完成的，不打包的后果就是会造成更多的进程切换，效率降低，延迟增加，尤其对于多个不连续地址的小规模数据传送而言就更为明显。

通用接口层为我们提供了一系列用于操作和维护spi\_message和spi\_transfer的API函数，这里也列一下。

用于初始化spi\_message结构：

* void spi\_message\_init(struct spi\_message \*m);

把一个spi\_transfer加入到一个spi\_message中（注意，只是加入，未启动传输过程），和移除一个spi\_transfer：

* void spi\_message\_add\_tail(struct spi\_transfer \*t, struct spi\_message \*m);
* void spi\_transfer\_del(struct spi\_transfer \*t);

以上两个API的组合，初始化一个spi\_message并添加数个spi\_transfer结构：

* void spi\_message\_init\_with\_transfers(struct spi\_message \*m, struct spi\_transfer \*xfers, unsigned int num\_xfers);

分配一个自带数个spi\_transfer机构的spi\_message：

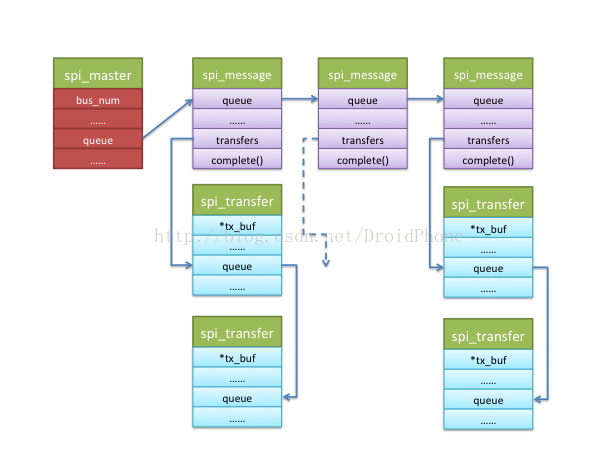
* struct spi\_message \*spi\_message\_alloc(unsigned ntrans, gfp\_t flags);

发起一个spi\_message的传送操作：

异步版本    int spi\_async(struct spi\_device \*spi, struct spi\_message \*message);

同步版本    int spi\_sync(struct spi\_device \*spi, struct spi\_message \*message);

利用以上这些API函数，SPI设备的协议驱动程序就可以完成与某个SPI设备的数据交换工作，同时也可以看到，因为有通用接口层的隔离，控制器驱动对于协议驱动程序来说是透明的，也就是说，协议驱动程序只关心具体需要处理和交换的数据，无需关心控制器是如何传送这些数据的。spi\_master，spi\_message，spi\_transfer这几个数据结构的关系可以用下图来描述：



总结一下，协议驱动发送数据的流程大致是这样的：

1. 定义一个spi\_message结构；
2. 用spi\_message\_init函数初始化spi\_message；
3. 定义一个或数个spi\_transfer结构，初始化并为数据准备缓冲区并赋值给spi\_transfer相应的字段（tx\_buf，rx\_buf等）；
4. 通过spi\_message\_init函数把这些spi\_transfer挂在spi\_message结构下；
5. 如果使用同步方式，调用spi\_sync()，如果使用异步方式，调用spi\_async()；
   1. 另外，通用接口层也为一些简单的数据传输提供了独立的API来完成上述的组合过程：
6. int spi\_write(struct spi\_device \*spi, const void \*buf, size\_t len);    ----    同步方式发送数据。
7. int spi\_read(struct spi\_device \*spi, void \*buf, size\_t len);    ----    同步方式接收数据。
8. int spi\_sync\_transfer(struct spi\_device \*spi, struct spi\_transfer \*xfers, unsigned int num\_xfers);    ----   同步方式，直接传送数个spi\_transfer，接收和发送。
9. int spi\_write\_then\_read(struct spi\_device \*spi, const void \*txbuf, unsigned n\_tx, void \*rxbuf, unsigned n\_rx);    ----    先写后读。
10. ssize\_t spi\_w8r8(struct spi\_device \*spi, u8 cmd);    ----    写8位，然后读8位。
11. ssize\_t spi\_w8r16(struct spi\_device \*spi, u8 cmd);    ----    写8位，然后读16位。

## 5.3 SPI控制器驱动

整个SPI驱动架构可以分为协议驱动、通用接口层和控制器驱动三大部分。其中，控制器驱动负责最底层的数据收发工作，为了完成数据的收发工作，控制器驱动需要完成以下这些功能：

1.    申请必要的硬件资源，例如中断，DMA通道，DMA内存缓冲区等等；

2.    配置SPI控制器的工作模式和参数，使之可以和相应的设备进行正确的数据交换工作；

3.    向通用接口层提供接口，使得上层的协议驱动可以通过通用接口层访问控制器驱动；

4.    配合通用接口层，完成数据消息队列的排队和处理，直到消息队列变空为止；

### 5.3.1 [定义控制器设备](http://blog.csdn.net/droidphone/article/details/24353293#t0)

SPI控制器遵循linux的设备模型框架，所以，一个SPI控制器在代码中对应一个device结构，对于嵌入式系统，我们通常把SPI控制器作为一个平台设备来对待，所以，对于我们来说，只要在板级的代码中为SPI控制器定义一个platform\_device结构即可。下面以Samsung的SOC芯片：S3C6410，做为例子，看看如何定义这个platform\_device。以下的代码来自：/arch/arm/plat-samsung/devs.c中：

**static** **struct** resource s3c64xx\_spi0\_resource[] = {

    [0] = DEFINE\_RES\_MEM(S3C\_PA\_SPI0, SZ\_256),

    [1] = DEFINE\_RES\_DMA(DMACH\_SPI0\_TX),

    [2] = DEFINE\_RES\_DMA(DMACH\_SPI0\_RX),

    [3] = DEFINE\_RES\_IRQ(IRQ\_SPI0),

};

**struct** platform\_device s3c64xx\_device\_spi0 = {

    .name       = "s3c6410-spi",

    .id     = 0,

    .num\_resources  = ARRAY\_SIZE(s3c64xx\_spi0\_resource),

    .resource   = s3c64xx\_spi0\_resource,

    .dev = {

        .dma\_mask       = &samsung\_device\_dma\_mask,

        .coherent\_dma\_mask  = DMA\_BIT\_MASK(32),

    },

};

由此可见，在这个platform\_device中，我们定义了控制器所需的寄存器地址、DMA通道资源和IRQ编号，设备的名字定义为：s3c64xx-spi，这个名字用于后续和相应的控制器驱动相匹配。在machine的初始化代码中，我们需要注册这个代表SPI控制器的平台设备，另外，也会通过s3c64xx\_spi0\_set\_platdata函数设置平台相关的参数供后续的控制器驱动使用：

**static** **struct** platform\_device \*crag6410\_devices[] \_\_initdata = {

        ......

        &s3c64xx\_device\_spi0,

        ......

};

**static** **void** \_\_init xxxx\_machine\_init(**void**)

{

        s3c64xx\_spi0\_set\_platdata(NULL, 0, 2);

        //注册平台设备

        platform\_add\_devices(crag6410\_devices, ARRAY\_SIZE(crag6410\_devices));

}

s3c64xx\_spi0\_set\_platdata函数的定义如下：

**void** \_\_init s3c64xx\_spi0\_set\_platdata(**int** (\*cfg\_gpio)(**void**), **int** src\_clk\_nr,

**int** num\_cs)

{

**struct** s3c64xx\_spi\_info pd;

    ......

    pd.num\_cs = num\_cs;

    pd.src\_clk\_nr = src\_clk\_nr;

    pd.cfg\_gpio = (cfg\_gpio) ? cfg\_gpio : s3c64xx\_spi0\_cfg\_gpio;

        ......

    s3c\_set\_platdata(&pd, **sizeof**(pd), &s3c64xx\_device\_spi0);

}

上述函数主要是指定了控制器使用到的gpio配置、片选引脚个数和时钟配置等信息。这些信息在后面的控制器驱动中要使用到。

### 5.3.2 注册SPI控制器的platform\_driver

上一节中，我们把SPI控制器注册为一个platform\_device，相应地，对应的驱动就应该是一个平台驱动：platform\_driver，它们通过platform bus进行相互匹配。以下的代码来自：/drivers/spi/spi-s3c64xx.c

**static** **struct** platform\_driver s3c64xx\_spi\_driver = {

        .driver = {

                .name   = "s3c64xx-spi",

                .owner = THIS\_MODULE,

                .pm = &s3c64xx\_spi\_pm,

                .of\_match\_table = of\_match\_ptr(s3c64xx\_spi\_dt\_match),

        },

        .remove = s3c64xx\_spi\_remove,

        .id\_table = s3c64xx\_spi\_driver\_ids,

};

MODULE\_ALIAS("platform:s3c64xx-spi");

**static** **int** \_\_init s3c64xx\_spi\_init(**void**)

{

**return** platform\_driver\_probe(&s3c64xx\_spi\_driver, s3c64xx\_spi\_probe);

}

subsys\_initcall(s3c64xx\_spi\_init);

显然，系统初始化阶段（subsys\_initcall阶段），通过s3c64xx\_spi\_init()，注册了一个平台驱动，该驱动的名字正好也是：s3c64xx-spi，自然地，平台总线会把它和上一节定义的platform\_device匹配上，并且触发probe回调被调用（就是s3c64xx\_spi\_probe函数）。当然，这里的匹配是通过id\_table字段完成的：

**static** **struct** platform\_device\_id s3c64xx\_spi\_driver\_ids[] = {

        {

                .name           = "s3c2443-spi",

                .driver\_data    = (kernel\_ulong\_t)&s3c2443\_spi\_port\_config,

        }, {

                .name           = "s3c6410-spi",

                .driver\_data    = (kernel\_ulong\_t)&s3c6410\_spi\_port\_config,

        },

        ......

        { },

};

### 5.3.3 注册spi\_master

在linux设备模型看来，代表SPI控制器的是第一节所定义的platform\_device结构，但是对于SPI通用接口层来说，代表控制器的是spi\_master结构，关于spi\_master结构的描述，请参看第二节。我们知道，设备和驱动匹配上后，驱动的probe回调函数就会被调用，而probe回调函数正是对驱动程序和设备进行初始化的合适时机，本例中，对应的probe回调是：s3c64xx\_spi\_probe：

**static** **int** s3c64xx\_spi\_probe(**struct** platform\_device \*pdev)

{

        ......

        /\* 分配一个spi\_master结构 \*/

        master = spi\_alloc\_master(&pdev->dev,

**sizeof**(**struct** s3c64xx\_spi\_driver\_data));

        ......

        platform\_set\_drvdata(pdev, master);

        ......

        master->dev.of\_node = pdev->dev.of\_node;

        master->bus\_num = sdd->port\_id;

        master->setup = s3c64xx\_spi\_setup;

        master->cleanup = s3c64xx\_spi\_cleanup;

        master->prepare\_transfer\_hardware = s3c64xx\_spi\_prepare\_transfer;

        master->transfer\_one\_message = s3c64xx\_spi\_transfer\_one\_message;

        master->unprepare\_transfer\_hardware = s3c64xx\_spi\_unprepare\_transfer;

        master->num\_chipselect = sci->num\_cs;

        master->dma\_alignment = 8;

        master->bits\_per\_word\_mask = SPI\_BPW\_MASK(32) | SPI\_BPW\_MASK(16) |

                                        SPI\_BPW\_MASK(8);

        /\* the spi->mode bits understood by this driver: \*/

        master->mode\_bits = SPI\_CPOL | SPI\_CPHA | SPI\_CS\_HIGH;

        master->auto\_runtime\_pm = **true**;

        ......

        /\* 向通用接口层注册spi\_master结构 \*/

**if** (spi\_register\_master(master)) {

                dev\_err(&pdev->dev, "cannot register SPI master\n");

                ret = -EBUSY;

**goto** err3;

        }

        ......

}

上述函数，除了完成必要的硬件资源初始化工作以外，最重要的工作就是通过spi\_alloc\_master函数分配了一个spi\_master结构，初始化该结构，最终通过spi\_register\_master函数完成了对控制器的注册工作。从代码中我们也可以看出，spi\_master结构中的几个重要的回调函数已经被赋值，这几个回调函数由通用接口层在合适的时机被调用，以便完成控制器和设备之间的数据交换工作。

### 5.3.4 实现spi\_master结构的回调函数

事实上，SPI控制器驱动程序的主要工作，就是要实现spi\_master结构中的几个回调函数，其它的工作逻辑，均由通用接口层帮我们完成，通用接口层会在适当的时机调用这几个回调函数，这里我只是介绍一下各个回调函数的作用，具体的实现例子，请各位自行阅读代码树中各个平台的例子（代码位于：/drivers/spi/）。

int (\*setup)(struct spi\_device \*spi)

当协议驱动希望修改控制器的工作模式或参数时，会调用通用接口层提供的API：spi\_setup()，该API函数最后会调用setup回调函数来完成设置工作。

int (\*transfer)(struct spi\_device \*spi, struct spi\_message \*mesg)

目前已经可以不用我们自己实现该回调函数，初始化时直接设为NULL即可，目前的通用接口层已经实现了消息队列化，注册spi\_master时，通用接口层会提供实现好的通用函数。现在只有一些老的驱动还在使用该回调方式，新的驱动应该停止使用该回调函数，而是应该使用队列化的transfer\_one\_message回调。需要注意的是，我们只能选择其中一种方式，设置了transfer\_one\_message回调，就不能设置transfer回调，反之亦然。

void (\*cleanup)(struct spi\_device \*spi)

当一个SPI从设备（spi\_device结构）被释放时，该回调函数会被调用，以便释放该从设备所占用的硬件资源。

int (\*prepare\_transfer\_hardware)(struct spi\_master \*master)

int (\*unprepare\_transfer\_hardware)(struct spi\_master \*master)

这两个回调函数用于在发起一个数据传送过程前和后，给控制器驱动一个机会，申请或释放某些必要的硬件资源，例如DMA资源和内存资源等等。

int (\*prepare\_message)(struct spi\_master \*master, struct spi\_message \*message)

int (\*unprepare\_message)(struct spi\_master \*master, struct spi\_message \*message)

这两个回调函数也是用于在发起一个数据传送过程前和后，给控制器驱动一个机会，对message进行必要的预处理或后处理，比如根据message需要交换数据的从设备，设定控制器的正确工作时钟、字长和工作模式等。

int (\*transfer\_one\_message)(struct spi\_master \*master, struct spi\_message \*mesg)

当通用接口层发现master的队列中有消息需要传送时，会调用该回调函数，所以该函数是真正完成一个消息传送的工作函数，当传送工作完成时，应该调用spi\_finalize\_current\_message函数，以便通知通用接口层，发起队列中的下一个消息的传送工作。

## 5.4 SPI数据传输的队列化

我们知道，SPI数据传输可以有两种方式：同步方式和异步方式。所谓同步方式是指数据传输的发起者必须等待本次传输的结束，期间不能做其它事情，用代码来解释就是，调用传输的函数后，直到数据传输完成，函数才会返回。而异步方式则正好相反，数据传输的发起者无需等待传输的结束，数据传输期间还可以做其它事情，用代码来解释就是，调用传输的函数后，函数会立刻返回而不用等待数据传输完成，我们只需设置一个回调函数，传输完成后，该回调函数会被调用以通知发起者数据传送已经完成。同步方式简单易用，很适合处理那些少量数据的单次传输。但是对于数据量大、次数多的传输来说，异步方式就显得更加合适。

对于SPI控制器来说，要支持异步方式必须要考虑以下两种状况：

对于同一个数据传输的发起者，既然异步方式无需等待数据传输完成即可返回，返回后，该发起者可以立刻又发起一个message，而这时上一个message还没有处理完。

 对于另外一个不同的发起者来说，也有可能同时发起一次message传输请求。

队列化正是为了为了解决以上的问题，所谓队列化，是指把等待传输的message放入一个等待队列中，发起一个传输操作，其实就是把对应的message按先后顺序放入一个等待队列中，系统会在不断检测队列中是否有等待传输的message，如果有就不停地调度数据传输内核线程，逐个取出队列中的message进行处理，直到队列变空为止。SPI通用接口层为我们实现了队列化的基本框架。

### 5.4.1 spi\_transfer的队列化

回顾一下通用接口层的介绍，对协议驱动来说，一个spi\_message是一次数据交换的原子请求，而spi\_message由多个spi\_transfer结构组成，这些spi\_transfer通过一个链表组织在一起，我们看看这两个数据结构关于spi\_transfer链表的相关字段：

**struct** spi\_transfer {

        ......

**const** **void**      \*tx\_buf;

**void**            \*rx\_buf;

        ......

**struct** list\_head transfer\_list;

};

**struct** spi\_message {

**struct** list\_head        transfers;

**struct** spi\_device       \*spi;

        ......

**struct** list\_head        queue;

        ......

};

可见，一个spi\_message结构有一个链表头字段：transfers，而每个spi\_transfer结构都包含一个链表头字段：transfer\_list，通过这两个链表头字段，所有属于这次message传输的transfer都会挂在spi\_message.transfers字段下面。我们可以通过以下API向spi\_message结构中添加一个spi\_transfer结构：

static inline void

spi\_message\_add\_tail(struct spi\_transfer \*t, struct spi\_message \*m)

{

        list\_add\_tail(&t->transfer\_list, &m->transfers);

}

通用接口层会以一个message为单位，在工作线程中调用控制器驱动的transfer\_one\_message回调函数来完成spi\_transfer链表的处理和传输工作，关于工作线程，我们留在后面讨论。

### 5.4.2 spi\_message的队列化

一个或者多个协议驱动程序可以同时向控制器驱动申请多个spi\_message请求，这些spi\_message也是以链表的形式被过在表示控制器的spi\_master结构体的queue字段下面：

struct spi\_master {

        struct device   dev;

        ......

        bool                            queued;

        struct kthread\_worker           kworker;

        struct task\_struct              \*kworker\_task;

        struct kthread\_work             pump\_messages;

        spinlock\_t                      queue\_lock;

        struct list\_head                queue;

        struct spi\_message              \*cur\_msg;

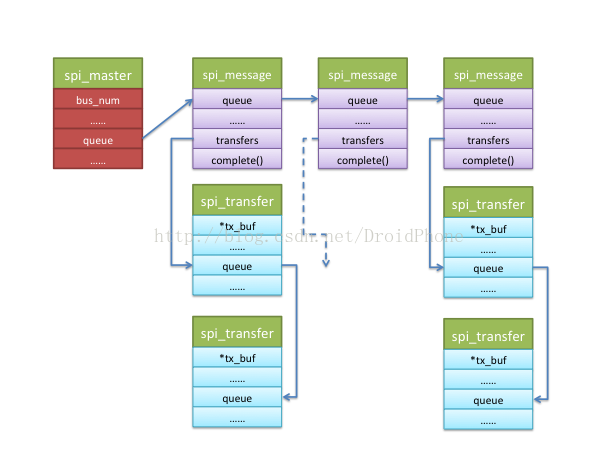
        ......

}

以下的API可以被协议驱动程序用于发起一个message传输操作：

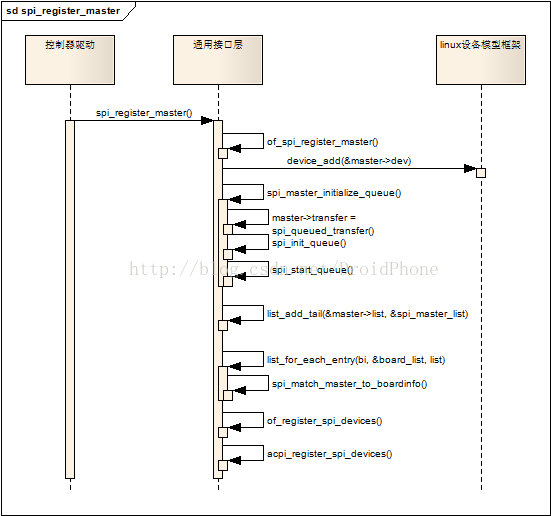
extern int spi\_async(struct spi\_device \*spi, struct spi\_message \*message)

spi\_async函数是发起一个异步传输的API，它会把spi\_message结构挂在spi\_master的queue字段下，然后启动专门为spi传输准备的内核工作线程，由该工作线程来实际处理message的传输工作，因为是异步操作，所以该函数会立刻返回，不会等待传输的完成，这时，协议驱动程序（可能是另一个协议驱动程序）可以再次调用该API，发起另一个message传输请求，结果就是，当工作线程被唤醒时，spi\_master下面可能已经挂了多个待处理的spi\_message结构，工作线程会按先进先出的原则来逐个处理这些message请求，每个message传送完成后，对应spi\_message结构的complete回调函数就会被调用，以通知协议驱动程序准备下一帧数据。这就是spi\_message的队列化。工作线程唤醒时，spi\_master、spi\_message和spi\_transfer之间的关系可以用下图来描述：



### 5.4.3 队列以及工作线程的初始化

通过第三节，SPI控制器驱动在初始化时，会调用通用接口层提供的API：spi\_register\_master，来完成控制器的注册和初始化工作，和队列化相关的字段和工作线程的初始化工作正是在该API中完成的。我先把该API的调用序列图贴出来：



我们当然不希望自己实现一套队列化框架，所以，如果你在实现一个新的SPI控制器驱动，请记住，不要在控制器驱动中实现并赋值spi\_master结构的transfer回调字段！进入spi\_master\_initialize\_queue函数看看：

**static** **int** spi\_master\_initialize\_queue(**struct** spi\_master \*master)

{

        ......

        master->queued = **true**;

        master->transfer = spi\_queued\_transfer;

**if** (!master->transfer\_one\_message)

                master->transfer\_one\_message = spi\_transfer\_one\_message;

        /\* Initialize and start queue \*/

        ret = spi\_init\_queue(master);

        ......

        ret = spi\_start\_queue(master);

        ......

}

该函数把master->transfer回调字段设置为默认的实现函数：spi\_queued\_transfer，如果控制器驱动没有实现transfer\_one\_message回调，用默认的spi\_transfer\_one\_message函数进行赋值。然后分别调用spi\_init\_queue和spi\_start\_queue函数初始化队列并启动工作线程。spi\_init\_queue函数最主要的作用就是建立一个内核工作线程：

**static** **int** spi\_init\_queue(**struct** spi\_master \*master)

{

        ......

        INIT\_LIST\_HEAD(&master->queue);

        ......

        init\_kthread\_worker(&master->kworker);

        master->kworker\_task = kthread\_run(kthread\_worker\_fn,

                                           &master->kworker, "%s",

                                           dev\_name(&master->dev));

        ......

        init\_kthread\_work(&master->pump\_messages, spi\_pump\_messages);

        ......

**return** 0;

}

内核工作线程的工作函数是：spi\_pump\_messages，该函数是整个队列化关键实现函数，我们将会在下一节中讨论该函数。spi\_start\_queue就很简单了，只是唤醒该工作线程而已：

**static** **int** spi\_start\_queue(**struct** spi\_master \*master)

{

        ......

        master->running = **true**;

        master->cur\_msg = NULL;

        ......

        queue\_kthread\_work(&master->kworker, &master->pump\_messages);

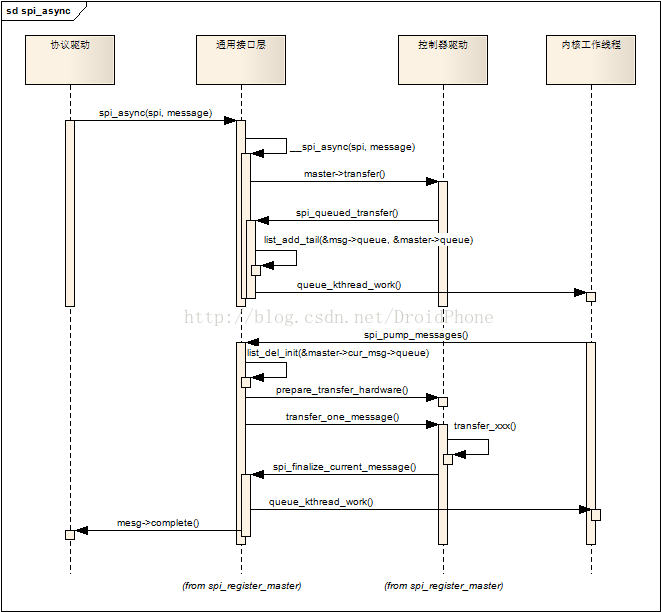
**return** 0;

}

自此，队列化的相关工作已经完成，系统等待message请求被发起，然后在工作线程中处理message的传送工作。

### 5.4.4 队列化的工作机制及过程

当协议驱动程序通过spi\_async发起一个message请求时，队列化和工作线程被激活，触发一些列的操作，最终完成message的传输操作。我们先看看spi\_async函数的调用序列图：



spi\_async会调用控制器驱动的transfer回调，前面一节已经讨论过，transfer回调已经被设置为默认的实现函数：spi\_queued\_transfer，该函数只是简单地把spi\_message结构加入spi\_master的queue链表中，然后唤醒工作线程。工作线程的工作函数是spi\_pump\_messages，它首先把该spi\_message从队列中移除，然后调用控制器驱动的prepare\_transfer\_hardware回调来让控制器驱动准备必要的硬件资源，然后调用控制器驱动的transfer\_one\_message回调函数完成该message的传输工作，控制器驱动的transfer\_one\_message回调函数在完成传输后，必须要调用spi\_finalize\_current\_message函数，通知通用接口层继续处理队列中的下一个message，另外，spi\_finalize\_current\_message函数也会调用该message的complete回调函数，以便通知协议驱动程序准备下一帧数据。

关于控制器驱动的transfer\_one\_message回调函数，我们的控制器驱动可以不用实现该函数，通用接口层已经为我们准备了一个标准的实现函数：spi\_transfer\_one\_message，这样，我们的控制器驱动就只要实现transfer\_one回调来完成实际的传输工作即可，而不用关心何时需压气哦调用spi\_finalize\_current\_message等细节。这里顺便也贴出transfer\_one\_message的代码：

**static** **int** spi\_transfer\_one\_message(**struct** spi\_master \*master,

**struct** spi\_message \*msg)

{

        ......

        spi\_set\_cs(msg->spi, **true**);

        list\_for\_each\_entry(xfer, &msg->transfers, transfer\_list) {

                ......

                reinit\_completion(&master->xfer\_completion);

                ret = master->transfer\_one(master, msg->spi, xfer);

                ......

**if** (ret > 0)

                        wait\_for\_completion(&master->xfer\_completion);

                ......

**if** (xfer->cs\_change) {

**if** (list\_is\_last(&xfer->transfer\_list,

                                         &msg->transfers)) {

                                keep\_cs = **true**;

                        } **else** {

                                cur\_cs = !cur\_cs;

                                spi\_set\_cs(msg->spi, cur\_cs);

                        }

                }

                msg->actual\_length += xfer->len;

        }

out:

**if** (ret != 0 || !keep\_cs)

                spi\_set\_cs(msg->spi, **false**);

        ......

        spi\_finalize\_current\_message(master);

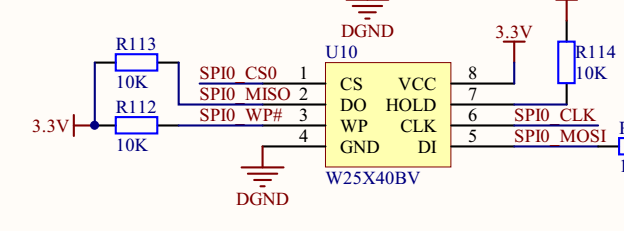
**return** ret;

}

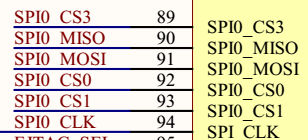
逻辑很清晰，这里就不再解释了。因为很多时候读者使用的内核版本和我写作时使用的版本不一样，经常会有人问有些函数或者结构不一样，所以这里顺便声明一下我使用的内核版本：3.13.0 -rc6。

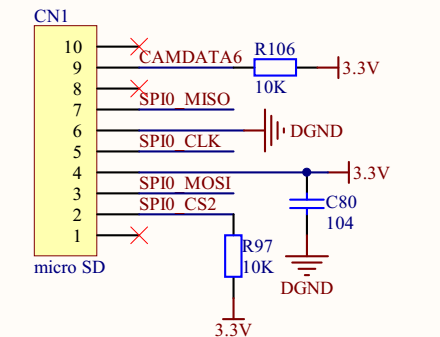
## 5.5 实例分析-驱动编写之SPI设备静态注册spidev.c

CS0放的是W25X40 内容是PMON，不能动。

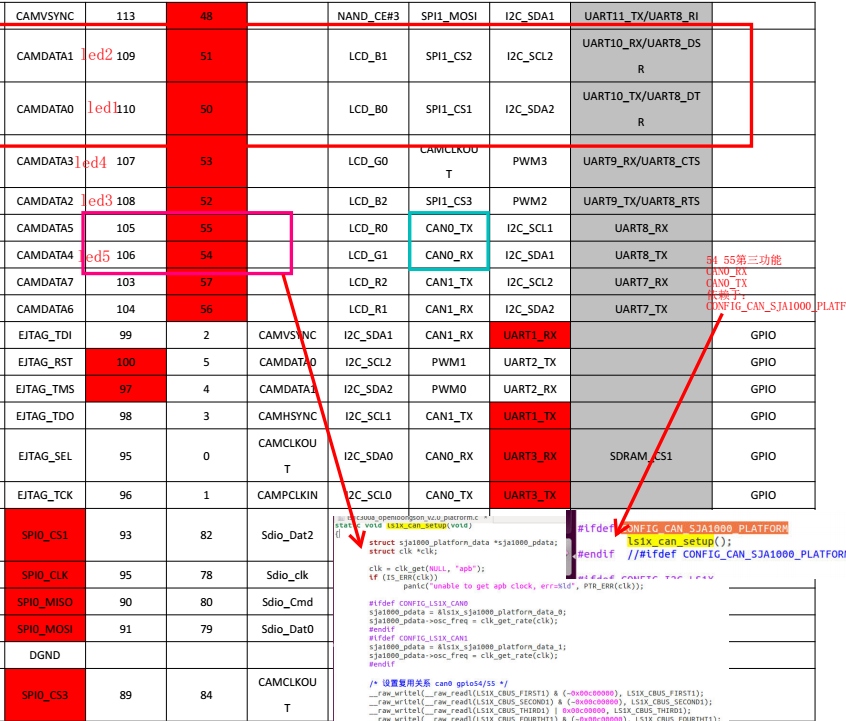


CS2操作SD卡。





CS1和CS3是空余的。



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。要使用spidev.c的驱动，只要在登记设备时，把设备名设置成spidev就可以。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，它的class是spidev\_class。它的父设备是在boardinfo中定义的spi设备。

下面就采用这种方法

1. 先创建一个spi\_board\_info结构描述spi设备信息，调用spi\_register\_board\_info将这个结构添加到board\_list中。

以上一定要在注册spi控制器驱动即spi master前。

平台文件添加 spidev ,使用CS3

{

.modalias = "spidev",

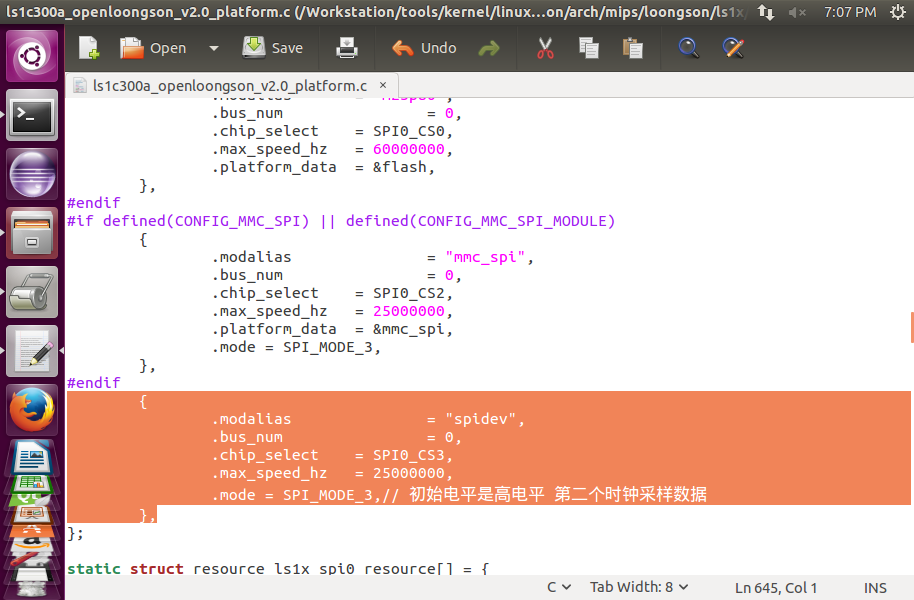
.bus\_num = 0,

.chip\_select = SPI0\_CS3,

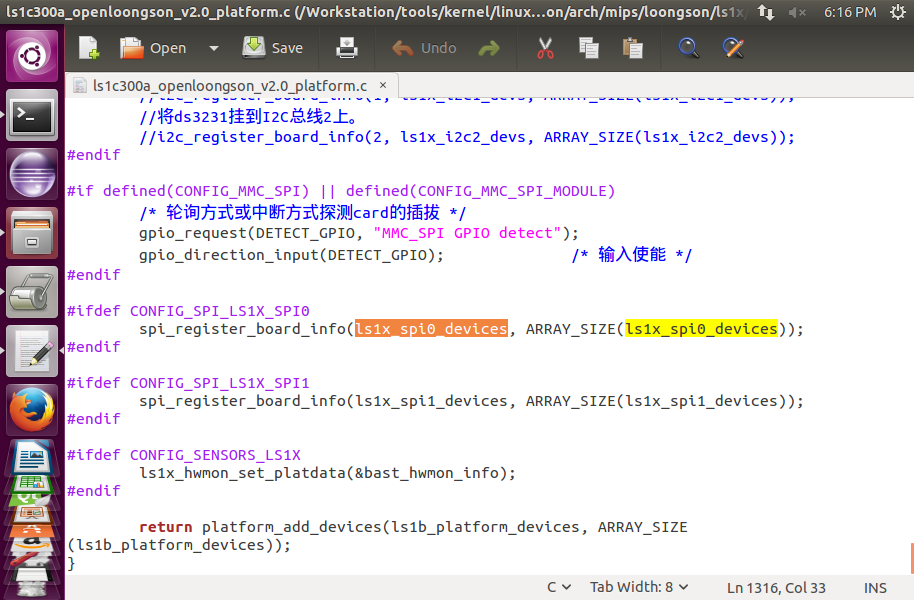
.max\_speed\_hz = 25000000,

.mode = SPI\_MODE\_3,// 初始电平是高电平 第二个时钟采样数据

},



调用下面函数，就把上面一个设备“spidev”登记到/sys/bus/spi下了,



spi\_register\_master注册SPI控制器驱动，此时会调用scan\_boardinfo扫描board\_list，根据spi\_board\_info调用spi\_new\_device生成spi\_device结构，用spi\_add\_device添加设备。

重启后下载到开发板上，ls /sys/class能看到新加的类：

[root@Loongson:/sys/class]#ls

backlight graphics input mtd sound video4linux

bdi hidraw leds net spi\_master video\_output

block hwmon mdio\_bus rtc spidev vtconsole

bsg i2c-adapter mem scsi\_device tty

firmware i2c-dev misc scsi\_disk ubi

gpio ieee80211 mmc\_host scsi\_host vc

[root@Loongson:/sys/class]#

登记的设备：

[root@Loongson:/]#ls /sys/bus/spi/devices

spi0.2 spi0.3

[root@Loongson:/]#

spi0.2是SD卡，spi0.3是spidev

查询设备节点：

[root@Loongson:/]#ls /dev |grep spi

spidev0.3

[root@Loongson:/]#

如果没有设备节点，内核选择自动添加驱动：user mode spi device driver support。

Devices Drivers --->

[\*] SPI support --->

<\*> user mode spi device driver support

## 5.6实例分析-驱动编写之SPI设备动态注册spike.c

参考：https://github.com/scottellis/spike

动态主册SPI设备的步骤如下:

1. 得到管理总线的spi\_master控制器指针(句柄);

2. 为总线分配spi\_device结构;

3. 验证没有其他的设备已经在这条总线bus.cs上注册过了;

4. 使用设备特定的值(speed,datasize,etc)来填充spi\_device;

5. 将新的spi\_device添加到总线;

源码如下：

/\*spike.c\*/

/\*

spike.c

Copyright Scott Ellis, 2010

This program is free software; you can redistribute it and/or modify

it under the terms of the GNU General Public License as published by

the Free Software Foundation; either version 2 of the License, or

(at your option) any later version.

This program is distributed in the hope that it will be useful,

but WITHOUT ANY WARRANTY; without even the implied warranty of

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

GNU General Public License for more details.

You should have received a copy of the GNU General Public License

along with this program; if not, write to the Free Software

Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

spike v0.1 loads and registers a spi driver for a device at the bus/

cable select specified by the constants SPI\_BUS.SPI\_BUS\_CS1

\*/

#include <linux/init.h>

#include <linux/module.h>

#include <linux/fs.h>

#include <linux/device.h>

#include <linux/mutex.h>

#include <linux/slab.h>

#include <linux/cdev.h>

#include <linux/spi/spi.h>

#include <linux/string.h>

#include <asm/uaccess.h>

#define USER\_BUFF\_SIZE 128

#define SPI\_BUS 0

#define SPI\_BUS\_CS1 1

#define SPI\_BUS\_SPEED 1000000

const char this\_driver\_name[] = "spike";

struct spike\_dev {

struct semaphore spi\_sem;

struct semaphore fop\_sem;

dev\_t devt;

struct cdev cdev;

struct class \*class;

struct spi\_device \*spi\_device;

char \*user\_buff;

};

static struct spike\_dev spike\_dev;

static ssize\_t spike\_read(struct file \*filp, char \_\_user \*buff, size\_t count,

loff\_t \*offp)

{

size\_t len;

ssize\_t status = 0;

if (!buff)

return -EFAULT;

if (\*offp > 0)

return 0;

if (down\_interruptible(&spike\_dev.fop\_sem))

return -ERESTARTSYS;

if (!spike\_dev.spi\_device)

strcpy(spike\_dev.user\_buff, "spi\_device is NULL\n");

else if (!spike\_dev.spi\_device->master)

strcpy(spike\_dev.user\_buff, "spi\_device->master is NULL\n");

else

sprintf(spike\_dev.user\_buff, "%s ready on SPI%d.%d\n",

this\_driver\_name,

spike\_dev.spi\_device->master->bus\_num,

spike\_dev.spi\_device->chip\_select);

len = strlen(spike\_dev.user\_buff);

if (len < count)

count = len;

if (copy\_to\_user(buff, spike\_dev.user\_buff, count)) {

printk(KERN\_ALERT "spike\_read(): copy\_to\_user() failed\n");

status = -EFAULT;

} else {

\*offp += count;

status = count;

}

up(&spike\_dev.fop\_sem);

return status;

}

static int spike\_open(struct inode \*inode, struct file \*filp)

{

int status = 0;

if (down\_interruptible(&spike\_dev.fop\_sem))

return -ERESTARTSYS;

if (!spike\_dev.user\_buff) {

spike\_dev.user\_buff = kmalloc(USER\_BUFF\_SIZE, GFP\_KERNEL);

if (!spike\_dev.user\_buff)

status = -ENOMEM;

}

up(&spike\_dev.fop\_sem);

return status;

}

static int spike\_probe(struct spi\_device \*spi\_device)

{

if (down\_interruptible(&spike\_dev.spi\_sem))

return -EBUSY;

spike\_dev.spi\_device = spi\_device;

up(&spike\_dev.spi\_sem);

return 0;

}

static int spike\_remove(struct spi\_device \*spi\_device)

{

if (down\_interruptible(&spike\_dev.spi\_sem))

return -EBUSY;

spike\_dev.spi\_device = NULL;

up(&spike\_dev.spi\_sem);

return 0;

}

static int \_\_init add\_spike\_device\_to\_bus(void)

{

struct spi\_master \*spi\_master;

struct spi\_device \*spi\_device;

struct device \*pdev;

char buff[64];

int status = 0;

spi\_master = spi\_busnum\_to\_master(SPI\_BUS);

if (!spi\_master) {

printk(KERN\_ALERT "spi\_busnum\_to\_master(%d) returned NULL\n",

SPI\_BUS);

printk(KERN\_ALERT "Missing modprobe omap2\_mcspi?\n");

return -1;

}

spi\_device = spi\_alloc\_device(spi\_master);

if (!spi\_device) {

put\_device(&spi\_master->dev);

printk(KERN\_ALERT "spi\_alloc\_device() failed\n");

return -1;

}

spi\_device->chip\_select = SPI\_BUS\_CS1;

/\* Check whether this SPI bus.cs is already claimed \*/

snprintf(buff, sizeof(buff), "%s.%u",

dev\_name(&spi\_device->master->dev),

spi\_device->chip\_select);

pdev = bus\_find\_device\_by\_name(spi\_device->dev.bus, NULL, buff);

if (pdev) {

/\* We are not going to use this spi\_device, so free it \*/

spi\_dev\_put(spi\_device);

/\*

\* There is already a device configured for this bus.cs

\* It is okay if it us, otherwise complain and fail.

\*/

if (pdev->driver && pdev->driver->name &&

strcmp(this\_driver\_name, pdev->driver->name)) {

printk(KERN\_ALERT

"Driver [%s] already registered for %s\n",

pdev->driver->name, buff);

status = -1;

}

} else {

spi\_device->max\_speed\_hz = SPI\_BUS\_SPEED;

spi\_device->mode = SPI\_MODE\_0;

spi\_device->bits\_per\_word = 8;

spi\_device->irq = -1;

spi\_device->controller\_state = NULL;

spi\_device->controller\_data = NULL;

strlcpy(spi\_device->modalias, this\_driver\_name, SPI\_NAME\_SIZE);

status = spi\_add\_device(spi\_device);

if (status < 0) {

spi\_dev\_put(spi\_device);

printk(KERN\_ALERT "spi\_add\_device() failed: %d\n",

status);

}

}

put\_device(&spi\_master->dev);

return status;

}

static struct spi\_driver spike\_driver = {

.driver = {

.name = this\_driver\_name,

.owner = THIS\_MODULE,

},

.probe = spike\_probe,

.remove = \_\_devexit\_p(spike\_remove),

};

static int \_\_init spike\_init\_spi(void)

{

int error;

error = spi\_register\_driver(&spike\_driver);

if (error < 0) {

printk(KERN\_ALERT "spi\_register\_driver() failed %d\n", error);

return error;

}

error = add\_spike\_device\_to\_bus();

if (error < 0) {

printk(KERN\_ALERT "add\_spike\_to\_bus() failed\n");

spi\_unregister\_driver(&spike\_driver);

return error;

}

return 0;

}

static const struct file\_operations spike\_fops = {

.owner = THIS\_MODULE,

.read = spike\_read,

.open = spike\_open,

};

static int \_\_init spike\_init\_cdev(void)

{

int error;

spike\_dev.devt = MKDEV(0, 0);

error = alloc\_chrdev\_region(&spike\_dev.devt, 0, 1, this\_driver\_name);

if (error < 0) {

printk(KERN\_ALERT "alloc\_chrdev\_region() failed: %d \n",

error);

return -1;

}

cdev\_init(&spike\_dev.cdev, &spike\_fops);

spike\_dev.cdev.owner = THIS\_MODULE;

error = cdev\_add(&spike\_dev.cdev, spike\_dev.devt, 1);

if (error) {

printk(KERN\_ALERT "cdev\_add() failed: %d\n", error);

unregister\_chrdev\_region(spike\_dev.devt, 1);

return -1;

}

return 0;

}

static int \_\_init spike\_init\_class(void)

{

spike\_dev.class = class\_create(THIS\_MODULE, this\_driver\_name);

if (!spike\_dev.class) {

printk(KERN\_ALERT "class\_create() failed\n");

return -1;

}

if (!device\_create(spike\_dev.class, NULL, spike\_dev.devt, NULL,

this\_driver\_name)) {

printk(KERN\_ALERT "device\_create(..., %s) failed\n",

this\_driver\_name);

class\_destroy(spike\_dev.class);

return -1;

}

return 0;

}

static int \_\_init spike\_init(void)

{

memset(&spike\_dev, 0, sizeof(spike\_dev));

sema\_init(&spike\_dev.spi\_sem, 1);

sema\_init(&spike\_dev.fop\_sem, 1);

if (spike\_init\_cdev() < 0)

goto fail\_1;

if (spike\_init\_class() < 0)

goto fail\_2;

if (spike\_init\_spi() < 0)

goto fail\_3;

return 0;

fail\_3:

device\_destroy(spike\_dev.class, spike\_dev.devt);

class\_destroy(spike\_dev.class);

fail\_2:

cdev\_del(&spike\_dev.cdev);

unregister\_chrdev\_region(spike\_dev.devt, 1);

fail\_1:

return -1;

}

module\_init(spike\_init);

static void \_\_exit spike\_exit(void)

{

spi\_unregister\_device(spike\_dev.spi\_device);

spi\_unregister\_driver(&spike\_driver);

device\_destroy(spike\_dev.class, spike\_dev.devt);

class\_destroy(spike\_dev.class);

cdev\_del(&spike\_dev.cdev);

unregister\_chrdev\_region(spike\_dev.devt, 1);

if (spike\_dev.user\_buff)

kfree(spike\_dev.user\_buff);

}

module\_exit(spike\_exit);

MODULE\_AUTHOR("Scott Ellis");

MODULE\_DESCRIPTION("spike module - an example SPI driver");

MODULE\_LICENSE("GPL");

MODULE\_VERSION("0.1");

安装spike模块后，出现了SPI0.1新的device 。0.2是SD卡， 0.3是上节建立的spidev.

[root@Loongson:/]#insmod spike.ko

[root@Loongson:/dev]#ls /sys/bus/spi/devices

spi0.1 spi0.2 spi0.3

查看设备，已经有了节点。

[root@Loongson:/]#cd dev

[root@Loongson:/dev]#ls

console mtd1 spike tty8

cpu\_dma\_latency mtd1ro tty tty9

dsp mtd2 tty0 ttyS0

fb0 mtd2ro tty1 ttyS1

full mtdblock0 tty10 ttyS2

i2c-0 mtdblock1 tty11 ttyS3

i2c-1 mtdblock2 tty12 ubi\_ctrl

i2c-2 network\_latency tty13 urandom

input network\_throughput tty14 vcs

kmsg null tty15 vcs1

ls1f-pwm ptmx tty2 vcsa

mem pts tty3 vcsa1

mmcblk0 random tty4 watchdog

mmcblk0p1 root tty5 zero

mtd0 rtc0 tty6

mtd0ro spidev0.3 tty7

## 5.7 编写测试程序

在内核目录\Documentation\spi下，有spidev\_test.c可以参考。以下程序作了修改，不停地发送字符 0x55。

/\*spidev\_test.c\*/

/\*

\* SPI testing utility (using spidev driver)

\*

\* Copyright (c) 2007 MontaVista Software, Inc.

\* Copyright (c) 2007 Anton Vorontsov <avorontsov@ru.mvista.com>

\*

\* This program is free software; you can redistribute it and/or modify

\* it under the terms of the GNU General Public License as published by

\* the Free Software Foundation; either version 2 of the License.

\*

\* Cross-compile with cross-gcc -I/path/to/cross-kernel/include

\*/

#include <stdint.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <getopt.h>

#include <fcntl.h>

#include <sys/ioctl.h>

#include <linux/types.h>

#include <linux/spi/spidev.h>

#define ARRAY\_SIZE(a) (sizeof(a) / sizeof((a)[0]))

static void pabort(const char \*s)

{

perror(s);

abort();

}

static const char \*device = "/dev/spidev0.3";

static uint8\_t mode;

static uint8\_t bits = 8;

static uint32\_t speed = 50000;

static uint16\_t delay;

unsigned char buf\_me[1] = {0x55};

static void transfer(int fd)

{

int ret;

uint8\_t tx[] = {

0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,

0x40, 0x00, 0x00, 0x00, 0x00, 0x95,

0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,

0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,

0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,

0xDE, 0xAD, 0xBE, 0xEF, 0xBA, 0xAD,

0xF0, 0x0D,

};

uint8\_t rx[ARRAY\_SIZE(tx)] = {0, };

struct spi\_ioc\_transfer tr = {

.tx\_buf = (unsigned long)tx,

.rx\_buf = (unsigned long)rx,

.len = ARRAY\_SIZE(tx),

.delay\_usecs = delay,

.speed\_hz = speed,

.bits\_per\_word = bits,

};

ret = ioctl(fd, SPI\_IOC\_MESSAGE(1), &tr);

if (ret < 1)

pabort("can't send spi message");

for (ret = 0; ret < ARRAY\_SIZE(tx); ret++) {

if (!(ret % 6))

puts("");

printf("%.2X ", rx[ret]);

}

puts("");

}

static void print\_usage(const char \*prog)

{

printf("Usage: %s [-DsbdlHOLC3]\n", prog);

puts(" -D --device device to use (default /dev/spidev1.1)\n"

" -s --speed max speed (Hz)\n"

" -d --delay delay (usec)\n"

" -b --bpw bits per word \n"

" -l --loop loopback\n"

" -H --cpha clock phase\n"

" -O --cpol clock polarity\n"

" -L --lsb least significant bit first\n"

" -C --cs-high chip select active high\n"

" -3 --3wire SI/SO signals shared\n");

exit(1);

}

static void parse\_opts(int argc, char \*argv[])

{

while (1) {

static const struct option lopts[] = {

{ "device", 1, 0, 'D' },

{ "speed", 1, 0, 's' },

{ "delay", 1, 0, 'd' },

{ "bpw", 1, 0, 'b' },

{ "loop", 0, 0, 'l' },

{ "cpha", 0, 0, 'H' },

{ "cpol", 0, 0, 'O' },

{ "lsb", 0, 0, 'L' },

{ "cs-high", 0, 0, 'C' },

{ "3wire", 0, 0, '3' },

{ "no-cs", 0, 0, 'N' },

{ "ready", 0, 0, 'R' },

{ NULL, 0, 0, 0 },

};

int c;

c = getopt\_long(argc, argv, "D:s:d:b:lHOLC3NR", lopts, NULL);

if (c == -1)

break;

switch (c) {

case 'D':

device = optarg;

break;

case 's':

speed = atoi(optarg);

break;

case 'd':

delay = atoi(optarg);

break;

case 'b':

bits = atoi(optarg);

break;

case 'l':

mode |= SPI\_LOOP;

break;

case 'H':

mode |= SPI\_CPHA;

break;

case 'O':

mode |= SPI\_CPOL;

break;

case 'L':

mode |= SPI\_LSB\_FIRST;

break;

case 'C':

mode |= SPI\_CS\_HIGH;

break;

case '3':

mode |= SPI\_3WIRE;

break;

case 'N':

mode |= SPI\_NO\_CS;

break;

case 'R':

mode |= SPI\_READY;

break;

default:

print\_usage(argv[0]);

break;

}

}

}

int main(int argc, char \*argv[])

{

int ret = 0;

int fd;

parse\_opts(argc, argv);

fd = open(device, O\_RDWR);

if (fd < 0)

pabort("can't open device");

/\*

\* spi mode

\*/

ret = ioctl(fd, SPI\_IOC\_WR\_MODE, &mode);

if (ret == -1)

pabort("can't set spi mode");

ret = ioctl(fd, SPI\_IOC\_RD\_MODE, &mode);

if (ret == -1)

pabort("can't get spi mode");

/\*

\* bits per word

\*/

ret = ioctl(fd, SPI\_IOC\_WR\_BITS\_PER\_WORD, &bits);

if (ret == -1)

pabort("can't set bits per word");

ret = ioctl(fd, SPI\_IOC\_RD\_BITS\_PER\_WORD, &bits);

if (ret == -1)

pabort("can't get bits per word");

/\*

\* max speed hz

\*/

ret = ioctl(fd, SPI\_IOC\_WR\_MAX\_SPEED\_HZ, &speed);

if (ret == -1)

pabort("can't set max speed hz");

ret = ioctl(fd, SPI\_IOC\_RD\_MAX\_SPEED\_HZ, &speed);

if (ret == -1)

pabort("can't get max speed hz");

printf("spi mode: %d\n", mode);

printf("bits per word: %d\n", bits);

printf("max speed: %d Hz (%d KHz)\n", speed, speed/1000);

// transfer(fd);

while(1){

write(fd,buf\_me,1);

}

close(fd);

return ret;

}

在虚拟机中用以下命令编译，出现错误：

root@ubuntu:/Workstation/examples/DrvProg/5.SPI# mipsel-linux-gcc -I /Workstation/tools/kernel/linux-3.0.82-openloongson/include/ spidev\_test.c -o spidev\_test

In file included from /opt/gcc-4.3-ls232/sysroot/usr/include/asm/types.h:15,

from /Workstation/tools/kernel/linux-3.0.82-openloongson/include/linux/types.h:4,

from spidev\_test.c:21:

/Workstation/tools/kernel/linux-3.0.82-openloongson/include/asm-generic/int-ll64.h:11:29: error: asm/bitsperlong.h: No such file or directory

In file included from spidev\_test.c:21:

/Workstation/tools/kernel/linux-3.0.82-openloongson/include/linux/types.h:13:2: warning: #warning "Attempt to use kernel headers from user space, see http://kernelnewbies.org/KernelHeaders"

root@ubuntu:/Workstation/examples/DrvProg/5.SPI# mipsel-linux-gcc -I /Workstation/tools/kernel/linux-3.0.82-openloongson/include/ spidev\_test.c -o spidev\_test

In file included from spidev\_test.c:21:

/Workstation/tools/kernel/linux-3.0.82-openloongson/include/linux/types.h:13:2: warning: #warning "Attempt to use kernel headers from user space, see http://kernelnewbies.org/KernelHeaders"

说明在int-ll64.h中找不到asm/bitsperlong.h。

下面修改内核目录/linux-3.0.82-openloongson/include/asm-generic/int-ll64.h 中的

#include <asm/bitsperlong.h>

改成：

#include <asm-generic/bitsperlong.h>

重新编译后成功，生成spidev\_test。拷备入开发板，运行正常。