# 硬件说明

spi(Serial Peripheral interface)串行外设接口是一种全双工、同步通信方式，采用一主多从模式，spi占用4根引脚，多个从设备则需要更多片选引脚，分别为：

MOSI:主设备输出/从设备输入数据线

MISO:主设备输入/从设备输出数据线

CLK:串行时钟，由主机产生

CS:片选信号，由主机控制

Master

CLK

MOSI

MISO

CS1

CS2

CS3

Slave

CLK MOSI MISO CS

Slave

CLK MOSI MISO CS

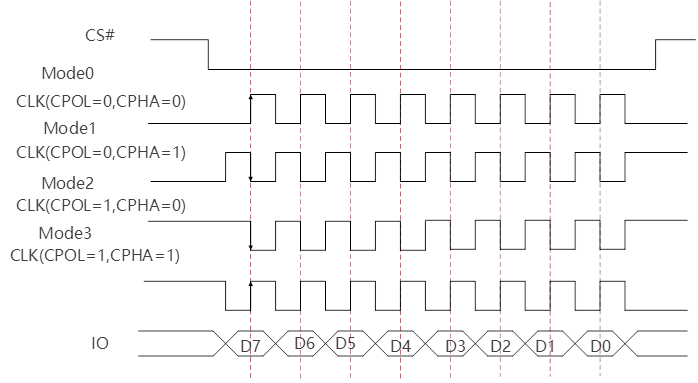
Slave

CLK MOSI MISO CS

## 时序说明

spi的时序由CPOL(Clock Polarity 时钟极性)、CPHA(Clock Phase时钟相位)决定，这两个参数就会产生4中工作模式。CPOL决定clk时钟信号在空闲状态的电平(0-低电平 1-高电平)，CPHA决定了数据在clk时钟信号的第几个边沿(无论是上升沿还是下降沿)被采样(0-第一个 1-第二个)。

4种工作模式下的时序如图：



在操作spi设备时，需要根据设备手册正确设置CPOL和CPHA才能正确读写数据。

# 驱动框架

应用程序

spidev

spi设备驱动

spi核心

spi master

spi控制器

spi设备

spi设备

spi设备

用户空间

内核空间

硬件

spi\_bus\_type

spi的驱动架构几乎与i2c驱动框架一样，用户空间可以直接通过spidev.c提供的通用接口访问设备，也可以通过spi设备驱动访问设备，设备的访问都是通过spi master提供的通信方法，每个spi master就对应着硬件上的spi控制器，每个spi控制器可以外挂多个spi设备，直接通过设备驱动访问设备时，需要设备与驱动进行匹配，一个设备驱动可以支持多个同类型的spi设备。

# spi控制器驱动

内核使用spi\_master结构来描述一个spi控制器:

struct spi\_master {

struct device dev;//控制器驱动对应的设备

struct list\_head list;//链接到全局spi\_master链表

s16 bus\_num;//控制器编号，决定了/dev/下spidevx.x的编号，第一个x就是控制器编号，第二个x为cs号

u16 num\_chipselect;//片选数量

u16 dma\_alignment;

u16 mode\_bits;//支持的设备模式

u32 bits\_per\_word\_mask;//传输设置的bits\_per\_word的掩码

u32 min\_speed\_hz;//最小传输速度

u32 max\_speed\_hz;//最大传输速度

u16 flags;//用于某些约束条件的标志

size\_t (\*max\_transfer\_size)(struct spi\_device \*spi);//返回最大传输大小，如未定义，则使用SIZE\_MAX

size\_t (\*max\_message\_size)(struct spi\_device \*spi);//返回最大消息大小，如未定义，则使用SIZE\_MAX

struct mutex io\_mutex;//物理总线访问的互斥锁

spinlock\_t bus\_lock\_spinlock;

struct mutex bus\_lock\_mutex;

bool bus\_lock\_flag;//总线锁定标志

int (\*setup)(struct spi\_device \*spi);//根据spi设备更新工作模式、时钟等配置

int (\*transfer)(struct spi\_device \*spi, struct spi\_message \*mesg);//添加消息到控制器的消息队列，此函数不可睡眠

void (\*cleanup)(struct spi\_device \*spi);//在spidev\_release函数中被调用去释放由spi\_master提供的内存

bool (\*can\_dma)(struct spi\_master \*master, struct spi\_device \*spi, struct spi\_transfer \*xfer);//返回spi master是否支持dma，如果支持dma，则会在transfer\_one() 前进行dma的映射

//内核实现spi master传输队列机制使用到的参数与函数，如果上面的transfer()未定义则使用此机制

bool queued;//是否提供内部消息队列

struct kthread\_worker kworker;//用于管理消息队列的工作队列线程

struct task\_struct \*kworker\_task;

struct kthread\_work pump\_messages;//用于注册到kworker上的kthread\_work，其回调真正实现数据传输

spinlock\_t queue\_lock;//工作队列锁

struct list\_head queue;//该控制器的消息队列，所有等待传输的消息挂在该链表下

struct spi\_message \*cur\_msg;//当前正在处理的消息

bool idling;//表示设备进入空闲状态

bool busy;

bool running;

bool rt;

bool auto\_runtime\_pm;

bool cur\_msg\_prepared;

bool cur\_msg\_mapped;

struct completion xfer\_completion;

size\_t max\_dma\_len;

int (\*prepare\_transfer\_hardware)(struct spi\_master \*master);//准备硬件资源，正式发起传输前调用

int (\*transfer\_one\_message)(struct spi\_master \*master, struct spi\_message \*mesg);//传输单个消息，如果未定义则使用内核实现的transfer\_one\_message()

int (\*unprepare\_transfer\_hardware)(struct spi\_master \*master);//释放硬件资源

int (\*prepare\_message)(struct spi\_master \*master, struct spi\_message \*message);//消息开始传输前的准备工作

int (\*unprepare\_message)(struct spi\_master \*master, struct spi\_message \*message);//与prepare\_message对应

int (\*spi\_flash\_read)(struct spi\_device \*spi, struct spi\_flash\_read\_message \*msg);//提供从闪存设备读取的加速接口

bool (\*flash\_read\_supported)(struct spi\_device \*spi);//是否支持flash读

//如果使用内核实现的通用传输接口transfer\_one\_message()则会用到如下函数

void (\*set\_cs)(struct spi\_device \*spi, bool enable);//设置cs的逻辑电平

int (\*transfer\_one)(struct spi\_master \*master, struct spi\_device \*spi, struct spi\_transfer \*transfer);//传输一个spi\_transfer

void (\*handle\_err)(struct spi\_master \*master,struct spi\_message \*message);//错误处理回调，如果transfer\_one\_message()发生错误

int \*cs\_gpios;

struct spi\_statistics statistics;

/\* DMA channels for use with core dmaengine helpers \*/

struct dma\_chan \*dma\_tx;

struct dma\_chan \*dma\_rx;

/\* dummy data for full duplex devices \*/

void \*dummy\_rx;

void \*dummy\_tx;

int (\*fw\_translate\_cs)(struct spi\_master \*master, unsigned cs);

};

我们想要对struct spi\_master成员进行赋值，需要先向内核申请：

struct spi\_master \*spi\_alloc\_master(struct device \*dev, unsigned size);

dev:一般为platform\_device 中的 dev 成员

size：私有数据的大小

此函数不仅申请了一个struct spi\_master，还申请了私有数据的内存，所以此函数一般在控制器驱动开始时就需要调用，然后对spi\_master进行赋值，然后通过spi\_master\_get\_devdata(master)得到私有数据指针对私有数据进行赋值。

对应的释放函数为：

void spi\_master\_put(struct spi\_master \*master);

对spi\_master初始化完成后使用如下函数进行注册：

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

对应的注销函数为：

void spi\_unregister\_master(struct spi\_master \*master);

# spi设备驱动

内核使用spi\_device来描述一个spi从设备：

struct spi\_device {

struct device dev;//设备结构

struct spi\_master \*master;//从设备使用的spi控制器

u32 max\_speed\_hz;//时钟最大频率

u8 chip\_select;//片选

u8 bits\_per\_word;//每个字长的bit数，默认为8

u16 mode;//模式

int irq;

void \*controller\_state;//控制器状态

void \*controller\_data;//控制器数据

char modalias[SPI\_NAME\_SIZE];//设备驱动名称

int cs\_gpio;

};

spi\_driver结构则描述了一个spi设备驱动：

struct spi\_driver {

const struct spi\_device\_id \*id\_table;//驱动支持的spi设备列表

int (\*probe)(struct spi\_device \*spi);

int (\*remove)(struct spi\_device \*spi);

void (\*shutdown)(struct spi\_device \*spi);

struct device\_driver driver;

};

同样驱动和设备匹配成功后就会执行probe函数，设备移除时执行remove函数。

spi\_driver初始化完成后向内核注册:

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

对应的注销函数为：

void spi\_unregister\_driver(struct spi\_driver \*sdrv);

与i2c一样内核也提供了一个宏能同时完成spi设备的注册与注销,展开后其实也是调用了上面的注册与注销函数，展开过程与module\_i2c\_driver的展开类似：

module\_spi\_driver(spi\_driver);

# spi核心

spi核心spi.c提供了spi控制器和设备的注册方法，以及数据传输方法，并提供了通用的数据传输队列机制。

## spi数据传输

### 相关结构

spi\_message结构描述了spi消息，一条消息可以包含多个spi\_transfer事务，spi\_message无论包含几个spi\_transfer，整条spi\_message的传输都不会被其他spi\_message打断:

struct spi\_message {

struct list\_head transfers;//包含一个或多个spi\_transfer的列表

struct spi\_device \*spi;//指向产生进入队列的事务的spi device

unsigned is\_dma\_mapped:1;//？？？

void (\*complete)(void \*context);//完成该message下所有spi\_transfer后调用

void \*context;//complete的参数

unsigned frame\_length;//消息中的字节总数

unsigned actual\_length;//实际传输成功的字节总数

int status;//消息发送结果，成功返回0，否则为负的错误码

struct list\_head queue;//用于把结构挂到spi\_master的queue上

void \*state;

struct list\_head resources;

};

spi\_transfer结构描述了spi传输的最小单元：

struct spi\_transfer {

const void \*tx\_buf;//发送缓冲区

void \*rx\_buf;//接收缓冲区

unsigned len;//缓冲区长度

dma\_addr\_t tx\_dma;//dma模式下的发送缓冲区地址

dma\_addr\_t rx\_dma;//dma模式下的接收缓冲区地址

struct sg\_table tx\_sg;

struct sg\_table rx\_sg;

unsigned cs\_change:1;//当前spi\_transfer完成后cs是否需要变化

unsigned tx\_nbits:3;

unsigned rx\_nbits:3;

#define SPI\_NBITS\_SINGLE 0x01 /\* 1bit transfer \*/

#define SPI\_NBITS\_DUAL 0x02 /\* 2bits transfer \*/

#define SPI\_NBITS\_QUAD 0x04 /\* 4bits transfer \*/

u8 bits\_per\_word;//每个字长的比特数，为0则使用spi\_device中的默认值8

u16 delay\_usecs;//发送完spi\_transfer后的延迟时间(ms)

u32 speed\_hz;//传输速度，为0则默认使用spi\_device的速度

struct list\_head transfer\_list;//用于链接到spi\_message的transfers上

};

spi\_master、spi\_message、spi\_transfer结构之间的关系如下：

spi\_master

bus\_num

queue

...

spi\_message

queue

transfers

...

spi\_transfer

\*tx\_buf

transfer\_list

...

spi\_transfer

\*tx\_buf

transfer\_list

...

spi\_message

queue

transfers

...

spi\_transfer

\*tx\_buf

transfer\_list

...

spi\_transfer

\*tx\_buf

transfer\_list

...

### API说明

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

初始化spi\_message结构。

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

把一个spi\_transfer加入到一个spi\_message中。

void spi\_transfer\_del(struct spi\_transfer \*t);

将spi\_transfer移除。

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

spi\_message\_init和spi\_message\_add\_tail的组合，初始化spi\_message并将num\_xfers个spi\_transfer加入到spi\_message

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

spi\_message传输，只是将message添加到spi\_master的传输队列中，并不阻塞等待传输完成。

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

spi\_message传输，阻塞等待传输完成。

数据传输的步骤都为：

初始化消息spi\_message->将spi\_transfer加入spi\_message->执行读写操作

所以接口调用顺序为：

spi\_message\_init->spi\_message\_add\_tail->spi\_sync/spi\_async

内核也提供了能同时完成上述三步操作的API：

int spi\_write(struct spi\_device \*spi, const void \*buf, size\_t len);

同步写。

int spi\_read(struct spi\_device \*spi, void \*buf, size\_t len);

同步读。

int spi\_write\_then\_read(struct spi\_device \*spi, const void \*txbuf, unsigned n\_tx, void \*rxbuf, unsigned n\_rx);

同步写完读。

### 传输过程分析

spi的数据传输过程可以分为两步，第一步是将spi\_message添加到spi\_master的队列中，第二步则为从队列中将任务取出然后真正完成数据的传输。

#### 工作队列初始化

在注册控制器驱动时会初始化spi\_master的工作队列与工作线程：

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

{

//如果定义了transfer则表示不使用内核提供的通用传输队列机制

if (master->transfer)

dev\_info(dev, "master is unqueued, this is deprecated\n");

else {

status = spi\_master\_initialize\_queue(master);

}

}

初始化通用传输队列机制的队列与工作线程：

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

{

//通用的master->transfer为队列化函数spi\_queued\_transfer

master->transfer = spi\_queued\_transfer;

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

//如果控制器未实现此方法，则使用内核提供的默认方法

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

//初始化队列与工作线程

ret = spi\_init\_queue(master);

master->queued = true;

//启动工作线程

ret = spi\_start\_queue(master);

return 0;

}

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

{

kthread\_init\_worker(&master->kworker);

//kthread\_worker\_fn作用就是判断master->kworker的work\_list是否为空，不为空则将kthread\_work从链表中取出并执行其work->func，否则进入睡眠

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

&master->kworker, "%s",

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

//将work->func指定为spi\_pump\_messages

kthread\_init\_work(&master->pump\_messages, spi\_pump\_messages);

}

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

{

//将master->pump\_messages加入master->kworker，最后唤醒master->kworker->task，也就是上面的kthread\_worker\_fn

kthread\_queue\_work(&master->kworker, &master->pump\_messages);

}

总结一下就是kthread\_worker\_fn判断master->kworker是否有kthread\_work，kthread\_work其实就是master->pump\_messages，有则执行spi\_pump\_messages，无则睡眠等待唤醒。spi\_pump\_messages中完成了真正的spi数据传输，那么谁完成唤醒操作呢？我们接着分析。

#### 向工作队列中添加任务

kthread\_queue\_work会完成唤醒，所以看谁执行了此函数就可以知道谁做出了唤醒操作：

异步传输过程如下，可以看到异步传输只是将消息添加到工作队列，然后唤醒工作线程，之后并不等待执行结果，直接就会返回：

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

{

ret = \_\_spi\_async(spi, message);

//master->transfer已被初始化为spi\_queued\_transfer

master->transfer(spi, message);

}

int spi\_queued\_transfer(struct spi\_device \*spi, struct spi\_message \*msg)

{

//将msg->queue链接到master的工作队列上

list\_add\_tail(&msg->queue, &master->queue);

if (!master->busy && need\_pump)

kthread\_queue\_work(&master->kworker, &master->pump\_messages);

}

//当一个spi\_message完成后，会执行以下函数，也会唤醒线程去传输下一个spi\_message

void spi\_finalize\_current\_message(struct spi\_master \*master)

{

kthread\_queue\_work(&master->kworker, &master->pump\_messages);

if (mesg->complete)

mesg->complete(mesg->context);

}

#### 数据传输

实际的数据传输过程在spi\_pump\_messages中完成：

void spi\_pump\_messages(struct kthread\_work \*work)

{

//取出master->queue队列中的第一个spi\_message

master->cur\_msg = list\_first\_entry(&master->queue, struct spi\_message, queue);

//master->transfer\_one\_message被初始化为spi\_transfer\_one\_message

ret = master->transfer\_one\_message(master, master->cur\_msg);

}

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

{

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

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

//控制器驱动中实现的实际数据传输过程

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

//在介绍spi\_master结构时已说过，master->transfer\_one返回值为1则表示执行未完成，则需要等待控制器完成，控制器完成后会执行spi\_finalize\_current\_transfer()通知已完成传输。

if (ret > 0) {

wait\_for\_completion\_timeout(&master->xfer\_completion, msecs\_to\_jiffies(ms));

}

spi\_finalize\_current\_message(master);

}

同步传输过程如下，可以看到不仅要将消息添加到队列，而且会立即执行，并且需要等待执行完毕。

int spi\_sync(struct spi\_device \*spi, struct spi\_message \*message)

{

//初始化完成量

DECLARE\_COMPLETION\_ONSTACK(done);

message->complete = spi\_complete;

message->context = &done;

//消息添加到队列

status = \_\_spi\_queued\_transfer(spi, message, false);

//执行数据传输

\_\_spi\_pump\_messages(master, false);

//等待执行完成

wait\_for\_completion(&done);

}

# 使用gpio作为cs

## 设备树更改

以am3352为例，每个控制器最多可以支持四个从设备，但是只有两个cs，所以如果想挂载超过2个从设备就需要使用gpio作为cs。

设备树需要添加cs-gpios属性，其中的0则表示使用硬件片选，如果为gpio则表示使用gpio作为片选，设备还是通过reg来选择片选：

&spi1 {

status = "okay";

pinctrl-names = "default";

pinctrl-0 = <&spi1\_pins\_default>;

ti,pindir-d0-out-d1-in;

cs-gpios = <0>, <0>, <&gpio0 29 0>, <&gpio1 5 0>;

device@2{

reg = <2>;//使用GPIO0\_29作为片选

compatible = "rohm,dh2228fv";

spi-max-frequency = <1000000>;//最大时钟频率

};

device@3{

reg = <3>;//使用GPIO1\_5作为片选

compatible = "rohm,dh2228fv";

spi-max-frequency = <1000000>;//最大时钟频率

};

};

## 代码分析

spi master注册时会解析设备树信息以及将spi设备注册到spi bus上

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

{

//解析设备树信息

status = of\_spi\_register\_master(master);

//遍历spi设备并将设备注册到spi bus上

of\_register\_spi\_devices(master);

}

spi master注册时会获取gpio:

static int of\_spi\_register\_master(struct spi\_master \*master)

{

//得到cs-gpios中gpio数目

nb = of\_gpio\_named\_count(np, "cs-gpios");

master->cs\_gpios = cs;

//获取gpio

for (i = 0; i < nb; i++)

cs[i] = of\_get\_named\_gpio(np, "cs-gpios", i);

}

spi设备的注册过程如下：

static struct spi\_device \*

of\_register\_spi\_device(struct spi\_master \*master, struct device\_node \*nc)

{

//spi device申请

spi = spi\_alloc\_device(master);

//将spi 设备添加到spi bus

spi\_add\_device(spi);

if (master->cs\_gpios)

spi->cs\_gpio = master->cs\_gpios[spi->chip\_select];

//设置spi模式与频率

status = spi\_setup(spi);

//设备的添加

status = device\_add(&spi->dev);

}

然后spi数据传输前后设置cs时就会用到gpio:

static void spi\_set\_cs(struct spi\_device \*spi, bool enable)

{

if (spi->mode & SPI\_CS\_HIGH)

enable = !enable;

//如果cs为gpio则控制gpio作为cs

if (gpio\_is\_valid(spi->cs\_gpio))

gpio\_set\_value(spi->cs\_gpio, !enable);

//否则使用硬件cs

else if (spi->master->set\_cs)

spi->master->set\_cs(spi, !enable);

}

但是注意使用gpio作为cs时，spi核心并不会帮我们申请gpio，所以实现spi master驱动时需要判断cs是否为gpio然后进行申请，一般会在master->setup中完成，如spi-omap2-mcspi.c中:

static int omap2\_mcspi\_setup(struct spi\_device \*spi)

{

if (gpio\_is\_valid(spi->cs\_gpio)) {

//gpio申请

ret = gpio\_request(spi->cs\_gpio, dev\_name(&spi->dev));

//初始化gpio电平

gpio\_direction\_output(spi->cs\_gpio, !(spi->mode & SPI\_CS\_HIGH));

}

}

# spi通用设备

## 说明

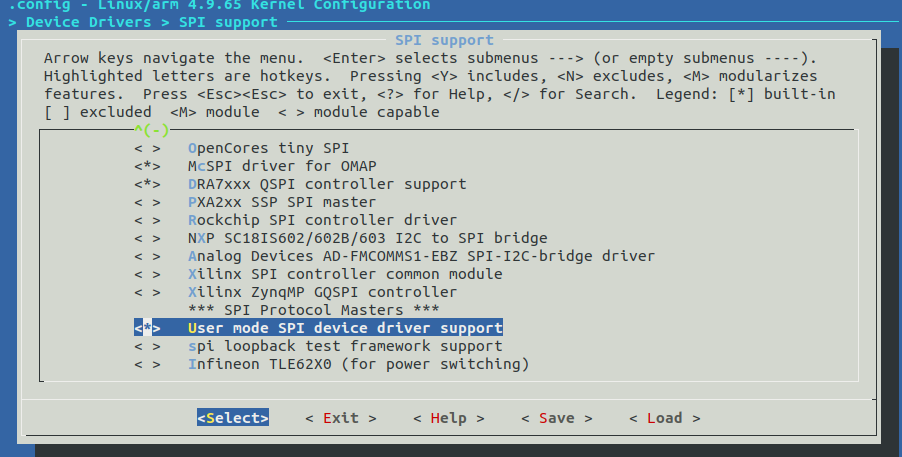
spidev.c实现的通用spi设备驱动为用户层提供了直接操作spi设备的接口，分析spidev的实现也有助于编写spi设备驱动。

同样是提供了通用设备驱动，i2c和spi的通用设备驱动区别如下：

1. i2c-dev.c通过通知链方式在i2c adapter注册到i2c总线上时自动创建i2c-x设备，而spidev.c并不能自动创建设备，需要匹配设备树节点才能进行设备创建。
2. i2c的通用设备与其i2c adapter关联，然后能访问该控制器下的所有设备，但是spi的通用设备是与挂载到spi控制器下的某个spi设备绑定的，只能访问指定设备。换句话说就是当我们将某个spi设备与通用spi设备驱动绑定后，我们就不能再将这个spi设备与自己实现的spi设备驱动绑定了，而i2c并不一样，i2c的通用设备驱动与自己实现的i2c设备驱动是并存的。

## 内核配置

想要使用spi通用设备驱动，内核需要添加如下配置：



设备树中也要在spi控制器节点下添加如下节点信息：

&spi1 {

device@0{

reg = <0>;//片选号

compatible = "rohm,dh2228fv";

spi-max-frequency = <1000000>;//最大时钟频率

};

};

## 代码分析

设备与驱动匹配成功后执行：

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

{

//创建spidevx.x设备，第一个x就是控制器编号，第二个x为片选号

dev = device\_create(spidev\_class, &spi->dev, spidev->devt, spidev, "spidev%d.%d", spi->master->bus\_num, spi->chip\_select);

}

当打开spidevx.x设备后，我们就可以调用read/write/ioctl接口函数了，但是read、write只能完成单条的单方向读和写操作，并不太好用，还是重点关注ioctl，通过ioctl不仅可以完成数据传输还能设置数据传输参数。

命令说明：

SPI\_IOC\_RD\_MODE/ SPI\_IOC\_WR\_MODE

参数为单字节指针，获取/设置spi传输模式，对于时序配置直接使用SPI\_MODE\_0..SPI\_MODE\_3，当然也可以自己组合SPI\_CPOL和SPI\_CPHA，更多模式请参考spidev.h中的定义，注意这个命令只能获取/设置8位的模式。

SPI\_IOC\_RD\_MODE32/SPI\_IOC\_WR\_MODE32

参数为32bit的指针，相对上面的命令能获取/设置超过8位的模式。

SPI\_IOC\_RD\_LSB\_FIRST/SPI\_IOC\_WR\_LSB\_FIRST

参数为单字节指针，获取/设置位序，0-MSB 其他-LSB。

SPI\_IOC\_RD\_BITS\_PER\_WORD/SPI\_IOC\_WR\_BITS\_PER\_WORD

参数为单字节指针，获取/设置每个字长的bit数，0代表0位。

SPI\_IOC\_RD\_MAX\_SPEED\_HZ/SPI\_IOC\_WR\_MAX\_SPEED\_HZ

参数为32bit的指针，获取/设置最大传输速度。

ioctl将除以上命令以外的命令都认为是数据传输，数据传输的参数使用spi\_ioc\_transfer结构表示：

struct spi\_ioc\_transfer {

\_\_u64 tx\_buf;//写数据缓冲区

\_\_u64 rx\_buf;//读数据缓冲区

\_\_u32 len;//缓冲区长度：字节

\_\_u32 speed\_hz;//通信速度

\_\_u16 delay\_usecs;//两个spi\_ioc\_transfer之间的延迟

\_\_u8 bits\_per\_word;//字长

\_\_u8 cs\_change;//传输下一个spi\_ioc\_transfer前是否更改片选

\_\_u8 tx\_nbits;

\_\_u8 rx\_nbits;

\_\_u16 pad;

};

数据传输过程如下

static struct spi\_ioc\_transfer \* spidev\_get\_ioc\_message(unsigned int cmd, struct spi\_ioc\_transfer \_\_user \*u\_ioc, unsigned \*n\_ioc)

{

//判断命令格式

if (\_IOC\_TYPE(cmd) != SPI\_IOC\_MAGIC || \_IOC\_NR(cmd) != \_IOC\_NR(SPI\_IOC\_MESSAGE(0)) || \_IOC\_DIR(cmd) != \_IOC\_WRITE)

return ERR\_PTR(-ENOTTY);

//获取数据大小

tmp = \_IOC\_SIZE(cmd);

//如果不为spi\_ioc\_transfer的整数倍，则认为参数无效

if ((tmp % sizeof(struct spi\_ioc\_transfer)) != 0)

return ERR\_PTR(-EINVAL);

//得到spi\_ioc\_transfer个数

\*n\_ioc = tmp / sizeof(struct spi\_ioc\_transfer);

//申请内存并将用户空间的数据copy过来

ioc = kmalloc(tmp, GFP\_KERNEL);

\_\_copy\_from\_user(ioc, u\_ioc, tmp);

}

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;

spi\_message\_init(&msg);

k\_xfers = kcalloc(n\_xfers, sizeof(\*k\_tmp), GFP\_KERNEL);

for (n = n\_xfers, k\_tmp = k\_xfers, u\_tmp = u\_xfers;n;n--, k\_tmp++, u\_tmp++) {

k\_tmp->rx\_buf = rx\_buf;

k\_tmp->tx\_buf = tx\_buf;

...

spi\_message\_add\_tail(k\_tmp, &msg);

}

spidev\_sync(spidev, &msg);

spi\_sync(spi, message);

}

从数据传输过程中也可以看到剥离掉驱动本身的一些处理，传输过程就是spi数据传输中讲过的三步：初始化消息spi\_message->将spi\_transfer加入spi\_message->执行读写操作。

内核提供了spidev.c的测试程序spidev\_test.c，分析spidev\_test.c就可以知道spi通用设备用户层接口使用方法。

# 示例说明

测试程序test\_spidev.c利用spi通用设备接口完成spi数据的传输，代码只是择取了spidev\_test.c的部分，只是为了演示spi通用设备接口的使用。