# spi bitbang实现原理分析

最近研究spi的bit bang,学自网络，不敢独享,特做此文档。
此文档是关于SPI的，读者对象为对linux 驱动如SPI作为platform有一定基础的programer。
个人只见，难免有误，欢迎大家批评指正。
此文档基于linux2.6.32内核

SPI bit bang是什么？我把他理解成用GPIO模拟SPI口，跟普通单片机没什么大的区别，下面我们将一步一步掀开bit bang的面纱.

## 一. SPI接口时序详细解

关于这部分，可以参考如下资料
，http://www.52rd.com/Blog/Detail_RD.Blog_yuwenxin_21678.html?#Flag_Comment
如下也摘抄于此，感谢博主 yuwenxin 分享如此好的一片文档
SPI接口有四种不同的数据传输时序，取决于CPOL和CPHL这两位的组合。图1中表现了这四种时序，
时序与CPOL、CPHL的关系也可以从图中看出。

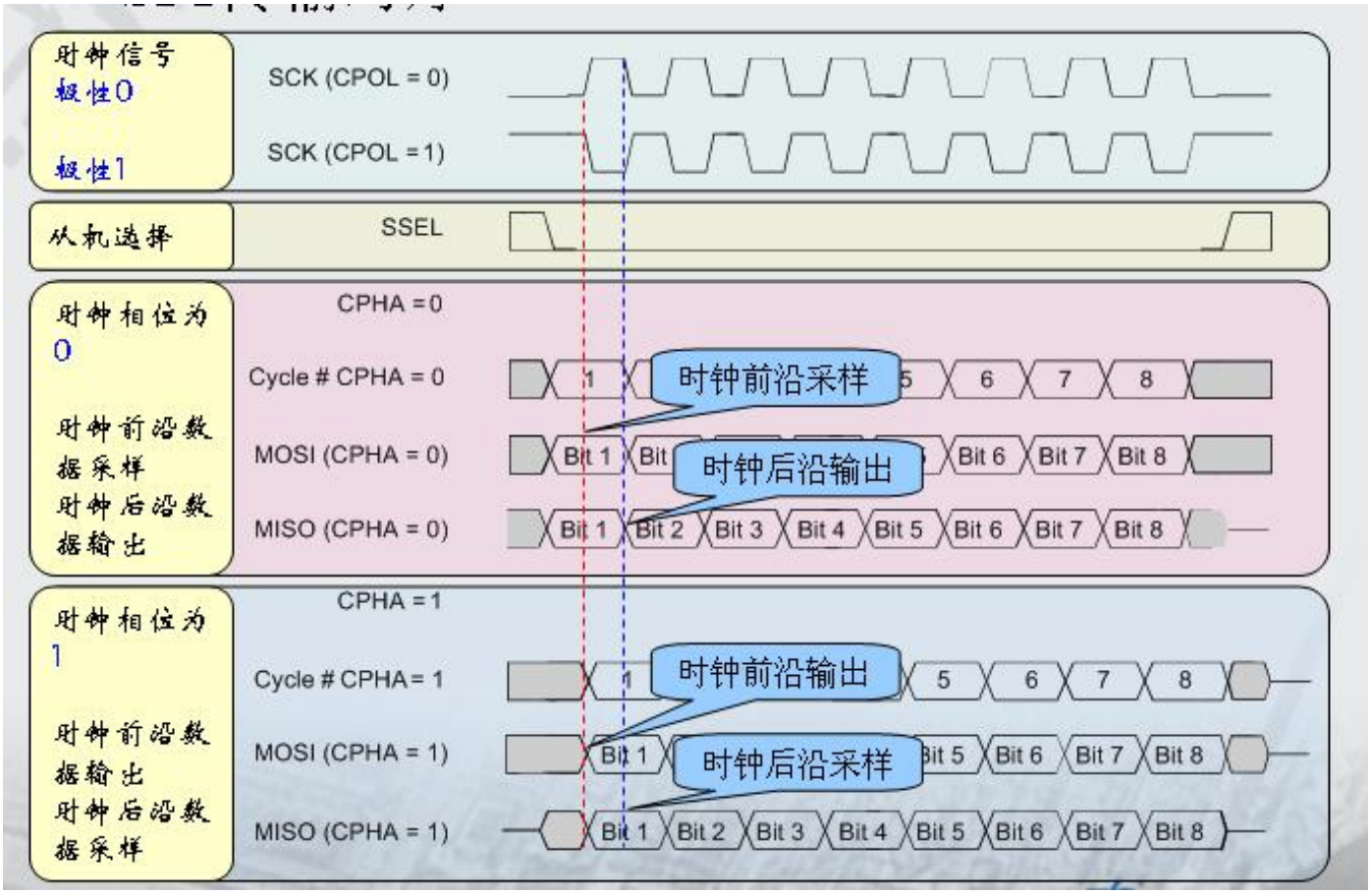


图1

注意:
CPOL是用来决定SCK时钟信号空闲时的电平，CPOL＝0，空闲电平为低电平，CPOL＝1时，

空闲电平为高电平。CPHA是用来决定采样时刻的，CPHA=0，在每个周期的第一个时钟沿采样，

CPHA＝1，在每个周期的第二个时钟沿采样。

二, linux中的spi驱动模型
linux的SPI模型中重要的有如下几个结构体,位置include/linux/spi/spi.h
struct spi_device {}  //Master side proxy for an SPI slave device
struct spi_driver {}  //Host side "protocol" driver
struct spi_master {}  //interface to SPI master controller
struct spi_transfer{}  //a read/write buffer pair
在这几个结构体中，我们只注意一下device结构体

```
struct spi_device
{
.....
#define SPI_CPHA 0x01 /* clock phase 同步*/
#define SPI_CPOL 0x02 /* clock polarity */
#define SPI_MODE_0 (0|0) /* (original MicroWire) */
#define SPI_MODE_1 (0|SPI_CPHA)
#define SPI_MODE_2 (SPI_CPOL|0)
#define SPI_MODE_3 (SPI_CPOL|SPI_CPHA)
.....
}
```

注意如上定义的 SPI_CPHA和SPI_CPOL
这个和一中的CPHA,以及CPOL是对应的,
然后在次基础上定义了MODE?,到此，你是否能相像出SPI接口的数据传输过程?
接着卖我吗继续研究
spi bitbang，也有一个重要的结构体位置为inlcud/linux/spi/spi_bitbang.h

```
struct spi_bitbang {
.....
 struct workqueue_struct *workqueue;
 struct work_struct work;
......
int (*txrx_bufs)(struct spi_device *spi, struct spi_transfer *t);
 u32 (*txrx_word[4])(struct spi_device *spi,unsigned nsecs, u32 word, u
```

```
8 bits);
......
}
```

我们只对上面四个东西感兴趣
struct work_struct work;
这个让我们想入菲菲--bit bang是按照workqueue队列的形式来工作的吗？

worqueue这个是什么？简单地说，他就是一个队列，里面的每一个work节点代表着一个需要调度的工作。
具体可以百度，这个和tasklist有点相似的。
既然是worqueue，那么我们可以猜想，spi是在每一个work中实现bit bang的。
事实上确实如此，在/driver/spi/spi_bitbang.c#L267中，我们可以看到如下函数
static void bitbang_work(struct work_struct *work){},
在/driver/spi/spi_bitbang.c中，我们可以发现
int spi_bitbang_start(struct spi_bitbang *bitbang);
这个函数的主要工作是完成如下工作
workqueue的初始化INIT_WORK(&bitbang->work, bitbang_work);
类寄存器的赋值
数据发送方法的初始化
bitbang->setup_transfer =spi_bitbang_setup_transfer;
注意这个函数(L=/driver/spi/spi_bitbang.c)

```
int spi_bitbang_setup_transfer(struct spi_device *spi, struct spi_trans
fer *t)
{
  ..............

    /* spi_transfer level calls that work per-word */
    if (!bits_per_word)
        bits_per_word = spi->bits_per_word;
    if (bits_per_word <= 8)
        cs->txrx_bufs = bitbang_txrx_8;
    else if (bits_per_word <= 16)
        cs->txrx_bufs = bitbang_txrx_16;
    else if (bits_per_word <= 32)
        cs->txrx_bufs = bitbang_txrx_32;
    else
```

```
        return -EINVAL;


    .....
}
```

追踪这个函数bitbang_txrx_xx，以为bitbang_txrx_16为例(L=ver/spi/spi_bitbang.c)

```
static unsigned bitbang_txrx_16(struct spi_device *spi,
 u32 (*txrx_word)(struct spi_device *spi, unsigned nsecs, u32 word, u8
bits),
 unsigned ns,struct spi_transfer *t)
{
.....
  word = txrx_word(spi, ns, word, bits);
.....
 return t->len - count;
}
```

那么txrx_word这个东西在哪里实现的呢？
/driver/spi/spi_gpio.c中的spi_gpio_probe函数中实现的！

```
static int __init spi_gpio_probe(struct platform_device *pdev)
{
.....
spi_gpio->bitbang.txrx_word[SPI_MODE_0] = spi_gpio_txrx_word_mode0;
spi_gpio->bitbang.txrx_word[SPI_MODE_1] = spi_gpio_txrx_word_mode1;
 spi_gpio->bitbang.txrx_word[SPI_MODE_2] = spi_gpio_txrx_word_mode2;
 spi_gpio->bitbang.txrx_word[SPI_MODE_3] = spi_gpio_txrx_word_mode3;
 spi_gpio->bitbang.setup_transfer = spi_bitbang_setup_transfer;
....
}
```

止于此，我们渐渐明朗：
spi_gpio_txrx_word_modex就是实现bitbang的基本，x对应上面定义的SPI的哥哥模式
以spi_gpio_txrx_word_mode0为例研究这个函数
spi_gpio_txrx_word_mode0()调用了bitbang_txrx_be_cpha0 该函数位
inlcude/linux/spi/spi_bitbang.h

```
static inline u32
```

```
bitbang_txrx_be_cpha0(struct spi_device *spi,unsigned nsecs, unsigned c
pol,u32 word, u8 bits)
{
 /* if (cpol == 0) this is SPI_MODE_0; else this is SPI_MODE_2 */
 /* clock starts at inactive polarity */
 for (word <<= (32 - bits); likely(bits); bits--) {
  /* setup MSB (to slave) on trailing edge */
  setmosi(spi, word & (1 << 31));
  spidelay(nsecs); /* T(setup) */
  setsck(spi, !cpol);
  spidelay(nsecs);
  /* sample MSB (from slave) on leading edge */
  word <<= 1;
  word |= getmiso(spi);
  setsck(spi, cpol);
 }
 return word;
}
```

注意其中的for循环，是不是单片机下的串口模拟SPI时序？