Popular

- Home
- Gumstix and Yocto
- Gumstix kernel dev
- Gumstix u-boot dev
- Gumstix I2C
- Gumstix GPIO
- Gumstix PWM
- Gumstix network boot
- Gumstix USB networking
- Writing a Linux SPI driver
- Gumstix developer tips
- Gumstix ADC
- Gumstix Buttons and LEDs

# Writing a Linux SPI driver - Part 2

Last Updated on Friday, 13 January 2012 19:08

SPI message basics

## Contents



The project code can be found at http://github.com/scottellis/spike

You can use the Github 'Switch Branches' menu in the upper left if you just want to browse the the code for a particular section.

## Linux SPI Communication Basics

The way you read or write data on the SPI bus is by packaging one or more **spi_transfer** records into a **spi_message** structure and then submitting the spi_message to the SPI system.

Here is what a spi_transfer structure looks like

```
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 bits_per_word;
        u16 delay_usecs;
        u32 speed_hz;
        struct list_head transfer_list;
};
```

**tx_buf** and **rx_buf** are dma safe buffers that you provide.

**len** is always the number of 8-bit bytes to clock, could be send, receive or both

**tx_dma** and **rx_dma** can be initialized by you with dma_map_single() or you can leave them alone and the controller driver will use these fields. If you do the mapping yourself, you should also set the spi_message.is_dma_mapped field to 1 informing the controller that you have done the work. If you map the dma buffers, then you are also responsible for unmapping them. Not all transfers use DMA. For the omap2_mcspi controller, transfers less then 8 bytes use PIO mode.

**cs_change** when set to 1 causes the CS line to go high between transfers in a spi_message series. Normally the CS goes low before the first transfer and remains there until the last transfer in this spi_message series. This toggling behavior is useful when communicating with some devices.

**bits_per_word** can be used to override the struct spi_device.bits_per_word field for a single transfer. Set to zero for no override.

**delay_usecs** will cause a fixed delay after a transfer and before a CS state change or the next transfer in this spi_message series.

**speed_hz** allows this transfer to run at a different rate then specified in struct spi_device.max_speed_hz. Set to zero for no override.

**transfer_list** is for internal use to maintain the transfers in this spi_message series.

Frequently, the only fields in a spi_transfer that you will be modifying are len, tx_buf and rx_buf. The remaining fields should be zero initialized.

A single spi_transfer can represent a read, a write or both since SPI is a full-duplex bus. In the underlying SPI controller driver, operations are always full-duplex. The appearance of a write-only operation is accomplished by passing NULL for the rx_buf in a spi_transfer record. The controller driver will then discard data coming in on the MISO line. Likewise, a read-only operation is done by passing NULL for the tx_buf in a spi_transfer. In this case, the controller driver will send zeros out the MOSI line when it clocks the bus. The spi_transfer len field always represents how many bytes to clock the bus.

Here is a spi_message structure

```
struct spi_message {
        struct list_head transfers;
        struct spi_device *spi;
        unsigned is_dma_mapped:1;
        void (*complete)(void *context);
```

```
        void *context;
        unsigned actual_length;
        int status;
        struct list_head queue;
        void *state;
};
```

**transfers** is the list of spi_transfers for this spi_message series.

**is_dma_mapped** indicates that the spi_transfer tx_dma and rx_dma are already dma mapped.

**complete** is the callback function you provide

**context** is optional data you provide which will then be given back to you as the argument to complete()

**actual_length** is a result field that holds the total number of bytes that were transfered by this spi_message series

**status** is a result field, zero if there were no errors for this spi_message series or an error code otherwise

**queue** and **state** are for internal use by the controller driver

There is a helper function you should use to initialize a spi_message structure before each use

```
void spi_message_init(struct spi_message *m);
```

To add transfers to a spi_message, use the spi_message_add_tail() function

```
void spi_message_add_tail(struct spi_transfer *t, struct spi_message *m);
```

And when you are ready with the spi_message, you can submit it for execution. The core method for submitting any message to the SPI system is spi_async().

```
int spi_async(struct spi_device *spi, struct spi_message *message);
```

The spi_async() call returns immediately. The function is guaranteed not to sleep and is safe to call from an interrupt handler or any code that cannot sleep. We'll take advantage of that in a later section.

Using spi_async() requires that you wait for the completion callback before you access any of the data buffers in the spi_transfer records you submitted. You also cannot free the memory for those buffers while the SPI system owns them. Once your completion callback is called, the buffers are yours again.

Waiting for the spi completion callback adds some complexity for simple reads and writes, so for this first example, we'll use one of the helper functions the Linux SPI framework provides. In this case, the spi_sync() function.

spi_sync has the same signature as spi_async()

```
int spi_sync(struct spi_device *spi, struct spi_message *message);
```

When you use spi_sync(), you do not have to provide a completion callback. When the function returns it is safe to look at the data buffers. If you look at the implementation of spi_sync() in drivers/spi/spi.c, you'll see that it uses spi_async() and then puts our calling thread to sleep until the completion callback is called. Since we'll be calling spi_sync() from a user thread, it is safe for us to sleep while the transfers complete.

## Code for Part2

Part2 will be a simple loopback test of the SPI bus. You should jumper the MOSI line (pin 5) to the MISO line (pin7) on the Overo expansion board. Now when when you do a read of the /dev/spike device, the driver will perform a full-duplex transfer, writing 4 bytes and simultaneously reading 4 bytes.

The changes from the Part 1 code are the addition of a structure to hold a spi_message and a spi_transfer structure and some allocated buffers for the data.

```
struct spike_control {
        struct spi_message msg;
        struct spi_transfer transfer;
        u8 *tx_buff;
        u8 *rx_buff;
};

static struct spike_control spike_ctl;
```

There are two new functions (https://gist.github.com/715712)

```
1  static void spike_prepare_spi_message(void)
2  {
3          spi_message_init(&spike_ctl.msg);
4
5          /* put some changing values in tx_buff for testing */
6          spike_ctl.tx_buff[0] = spike_dev.test_data++;
7          spike_ctl.tx_buff[1] = spike_dev.test_data++;
8          spike_ctl.tx_buff[2] = spike_dev.test_data++;
9          spike_ctl.tx_buff[3] = spike_dev.test_data++;
10
11         memset(spike_ctl.rx_buff, 0, SPI_BUFF_SIZE);
12
13         spike_ctl.transfer.tx_buf = spike_ctl.tx_buff;
14         spike_ctl.transfer.rx_buf = spike_ctl.rx_buff;
15         spike_ctl.transfer.len = 4;
16
17         spi_message_add_tail(&spike_ctl.transfer, &spike_ctl.msg);
```

```
18 }
```

gistfile1.txt hosted with ❤ by GitHub view raw

and (https://gist.github.com/715716)

```
1  static int spike_do_one_message(void)
2  {
3          int status;
4
5          if (down_interruptible(&spike_dev.spi_sem))
6                  return -ERESTARTSYS;
7
8          if (!spike_dev.spi_device) {
9                  up(&spike_dev.spi_sem);
10                 return -ENODEV;
11         }
12
13         spike_prepare_spi_message();
14
15         /* this will put us to sleep until the transfers are complete */
16         status = spi_sync(spike_dev.spi_device, &spike_ctl.msg);
17
18         up(&spike_dev.spi_sem);
19
20         return status;
21 }
```

gistfile1.txt hosted with ❤ by GitHub view raw

Continuing from part1 of this article, here is how to get and build the part2 code.

```
$ cd spike
$ git checkout -b part2 origin/part2
$ source overo-source-me.txt
$ make clean
$ make
```

Copy spike.ko to your system and load it. Unload any previous version if loaded.

```
root@overo:~# rmmod spike
root@overo:~# insmod spike.ko
```

Then test the write/read loopback be doing a read of the /dev/spike device. Make sure pins 5 and 7 of the Overo expansion board are jumpered.

```
root@overo:~# cat /dev/spike
Status: 0
TX: 0 1 2 3
RX: 0 1 2 3
root@overo:~# cat /dev/spike
Status: 0
TX: 4 5 6 7
RX: 4 5 6 7
root@overo:~# cat /dev/spike
Status: 0
TX: 8 9 10 11
RX: 8 9 10 11
root@overo:~# cat /dev/spike
Status: 0
TX: 12 13 14 15
RX: 12 13 14 15
```

The next section will use spi_async() directly to improve performance Part 3 - Asynchronous writes

## Notes

The git command

```
$ git checkout -b <new_branch_name> <start_point>
```

is only needed the first time you work with a new branch.
After that, you can use the command

```
$ git checkout <branch_name>
```

For example, to go back to part1 and the 'master' branch it would be

```
$ git checkout master
```

And then to come back to the 'part2' branch

```
$ git checkout part2
```

Be sure to run 'make clean' after switching branches.