Writing a Linux SPI driver - Part 3

Last Updated on Monday, 24 January 2011 17:09

SPI asynchronous writes

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.

Overview

This section will implement asynchronous writes on the SPI bus. It's a small change from part 2 so to make the example somewhat useful, a kernel high-resolution timer will be introduced to trigger the writes at a user specified frequency.

An overview of hrtimers can be found on the LWN.net site - The high-resolution timer API

The spike driver will accept the write frequency as a kernel module parameter.

The /dev/spike driver will respond to two commands, start and stop, using the file write interface.

Reads of the spike device will return some stats from the driver.

After receiving the start command, the driver will continuously write two bytes to the SPI busy at the frequency that was specified.

SPI Async

The spi_async function is the interface for submitting messages to the SPI system. All other methods for submitting spi_messages are built upon spi_async.

As the name implies, spi_async is asynchronous, meaning control returns back to the caller immediately. Internally, the spi controller driver adds the spi_message it receives to an internal work queue. The work queue is processed on another kernel work thread at some point in the future outside of your control.

When you use spi_async, you must provide a callback function in the spi_message structure or spi_async will reject it with an error.

The callback has a simple signature, a void return with a single void * argument. The argument is the value you provide in the context field of the spi_message. It can be NULL. The controller driver never looks at it.

The completion callback is called from a kernel thread that should not sleep. This limits what you can do in the callback. Some of the things you cannot do are allocate memory, access user memory or use a semaphore since they can all sleep.

There are standard strategies to deal with this. It is the same problem interrupt handlers have. For the example in this section I am avoiding the issue by keeping the implementation very simple.

Here it is (https://gist.github.com/729666)

```
static void spike_completion_handler(void *arg)
{
        spike_ctl.spi_callbacks++;
        spike_ctl.busy = 0;
}
```

gistfile1.txt hosted with ❤ by GitHub view raw

If you wanted you could make it even simpler and do absolutely nothing. You still have to provide the callback function though.

In the next sections when some real devices are introduced, the completion callback will get more sophisticated.

Code for Part3

The changes are fairly simple from the Part2 code.

The spike_control structure got a busy flag field and a few counters. The rx_buf was removed since it isn't used here.

(https://gist.github.com/729673)

```
struct spike_control {
        struct spi_message msg;
        struct spi_transfer transfer;
        u32 busy;
        u32 spi_callbacks;
        u32 busy_counter;
        u8 *tx_buff;
};
```

gistfile1.txt hosted with ❤ by **GitHub** **view raw**

The busy field is used as an indicator that the spi_message has been submitted, but the completion callback has not been received.

There is a new function to package a spi_transfer and submit a spi_message using spi_async.

(https://gist.github.com/729675)

```
static int spike_queue_spi_write(void)
{
        int status;
        unsigned long flags;

        spi_message_init(&spike_ctl.msg);

        /* this gets called when the spi_message completes */
        spike_ctl.msg.complete = spike_completion_handler;
        spike_ctl.msg.context = NULL;

        /* write some toggling bit patterns, doesn't really matter */
        spike_ctl.tx_buff[0] = 0xAA;
        spike_ctl.tx_buff[1] = 0x55;

        spike_ctl.transfer.tx_buf = spike_ctl.tx_buff;
        spike_ctl.transfer.rx_buf = NULL;
        spike_ctl.transfer.len = 2;

        spi_message_add_tail(&spike_ctl.transfer, &spike_ctl.msg);

        spin_lock_irqsave(&spike_dev.spi_lock, flags);

        if (spike_dev.spi_device)
                status = spi_async(spike_dev.spi_device, &spike_ctl.msg);
        else
                status = -ENODEV;

        spin_unlock_irqrestore(&spike_dev.spi_lock, flags);

        if (status == 0)
                spike_ctl.busy = 1;

        return status;
}
```

gistfile1.txt hosted with ❤ by **GitHub** **view raw**

A spinlock is used instead of a semaphore to guard the spi_device pointer access. This is because spike_queue_spi_write() is called in the timer callback which should not sleep.

A kernel hrtimer and a couple of time fields were added to the spike_dev structure. The second and nanosecond fields are just for convenience.

(https://gist.github.com/729676)

```
struct spike_dev {
        spinlock_t spi_lock;
        struct semaphore fop_sem;
        dev_t devt;
        struct cdev cdev;
        struct class *class;
        struct spi_device *spi_device;
        struct hrtimer timer;
        u32 timer_period_sec;
        u32 timer_period_ns;
        u32 running;
        char *user_buff;
};
```

gistfile1.txt hosted with ❤ by **GitHub** **view raw**

A running flag was added to spike_dev to keep track of the driver state.

The spike_dev.timer is initialized in the module init function spike_init()

```
...
```

```
hrtimer_init(&spike_dev.timer, CLOCK_MONOTONIC, HRTIMER_MODE_REL);
spike_dev.timer.function = spike_timer_callback;
...
```

The timer callback looks like this (https://gist.github.com/730170)

```
static enum hrtimer_restart spike_timer_callback(struct hrtimer *timer)
{
        if (!spike_dev.running) {
                return HRTIMER_NORESTART;
        }

        /* busy means the previous message has not completed */
        if (spike_ctl.busy) {
                spike_ctl.busy_counter++;
        }
        else if (spike_queue_spi_write() != 0) {
                return HRTIMER_NORESTART;
        }

        hrtimer_forward_now(&spike_dev.timer,
                ktime_set(spike_dev.timer_period_sec,
                        spike_dev.timer_period_ns));

        return HRTIMER_RESTART;
}
```

gistfile1.txt hosted with ❤ by **GitHub** **view raw**

The timer callback returns HRTIMER_NORESTART if the driver was stopped (spike_dev.running == 0) or if there was an error with spi_queue_spi_write(). If spike_ctl.busy is set, the timer is restarted but no new spi_message is submitted.

A file write handler was added for handling the start/stop commands and the read handler modified to return some state information.

If you want to verify that the driver is actually sending data for this part, you will need an oscope or signal analyzer. You can watch the CLK, MOSI and CS lines, Pins 3, 5 and 8 for CS1 on the Gumstix Overo expansion boards.

See part1 on how to get the code and checkout branches.

To build the part3 driver

```
$ cd spike
$ git checkout -b part3 origin/part3
$ source overo-source-me.txt
$ make clean
$ make
```

Copy spike.ko to your system and load it, unloading any previous version.

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

Reading from the spike device will return stats. The output is run-state|spi-callbacks|busy-counter.

```
root@overo:~# cat /dev/spike
Stopped|0|0
```

Here is how to test the two commands, start and stop.

```
root@overo:~# echo start > /dev/spike

root@overo:~# cat /dev/spike
Running|714|0

root@overo:~# cat /dev/spike
Running|1278|0

root@overo:~# echo stop > /dev/spike

root@overo:~# cat /dev/spike
Stopped|2025|0
```

The default write_frequency is 100 Hz. If you want to change it, do so using the write_frequency module parameter at load time.

```
root@overo:~# rmmod spike
root@overo:~# insmod spike.ko write_frequency=2000

root@overo:~# echo start > /dev/spike
```

```
root@overo:~# cat /dev/spike
Running|9560|3

root@overo:~# cat /dev/spike
Running|15618|7

root@overo:~# echo stop > /dev/spike

root@overo:~# cat /dev/spike
Stopped|30474|9
```

The busy-counter is incremented every time the timer interrupt finds that the previous spi_message still hasn't completed. As the write frequency increases, some messages aren't getting out before the next timer interrupt. Experimenting with the frequency values you can see the non-determinancy of the Linux SPI system.

The next section will be asynchronous reads using some ADCs to provide data.