Overview of Linux kernel SPI support

02-Feb-2012

What is SPI?

The "Serial Peripheral Interface" (SPI) is a synchronous four wire serial link used to connect microcontrollers to sensors, memory, and peripherals. It's a simple "de facto" standard, not complicated enough to acquire a standardization body. SPI uses a master/slave configuration.

The three signal wires hold a clock (SCK, often on the order of 10 MHz), and parallel data lines with "Master Out, Slave In" (MOSI) or "Master In, Slave Out" (MISO) signals. (Other names are also used.) There are four clocking modes through which data is exchanged; mode-0 and mode-3 are most commonly used. Each clock cycle shifts data out and data in; the clock doesn't cycle except when there is a data bit to shift. Not all data bits are used though; not every protocol uses those full duplex capabilities.

SPI masters use a fourth "chip select" line to activate a given SPI slave device, so those three signal wires may be connected to several chips in parallel. All SPI slaves support chipselects; they are usually active low signals, labeled nCSx for slave 'x' (e.g. nCS0). Some devices have other signals, often including an interrupt to the master.

Unlike serial busses like USB or SMBus, even low level protocols for SPI slave functions are usually not interoperable between vendors (except for commodities like SPI memory chips).

- SPI may be used for request/response style device protocols, as with touchscreen sensors and memory chips.
- It may also be used to stream data in either direction (half duplex), or both of them at the same time (full duplex).
- Some devices may use eight bit words. Others may use different word lengths, such as streams of 12-bit or 20-bit digital samples.
- Words are usually sent with their most significant bit (MSB) first, but sometimes the least significant bit (LSB) goes first instead.
- Sometimes SPI is used to daisy-chain devices, like shift registers.

In the same way, SPI slaves will only rarely support any kind of automatic discovery/enumeration protocol. The tree of slave devices accessible from a given SPI master will normally be set up manually, with configuration tables.

SPI is only one of the names used by such four-wire protocols, and most controllers have no problem handling "MicroWire" (think of it as half-duplex SPI, for request/response protocols), SSP ("Synchronous Serial Protocol"), PSP ("Programmable Serial Protocol"), and other related protocols.

Some chips eliminate a signal line by combining MOSI and MISO, and limiting themselves to half-duplex at the hardware level. In fact some SPI chips have this signal mode as a strapping option. These can be accessed using the same programming interface as SPI, but of course they won't handle full duplex transfers. You may find such chips described as using "three wire" signaling: SCK, data, nCSx. (That data line is sometimes called MOMI or SISO.)

Microcontrollers often support both master and slave sides of the SPI protocol. This document (and Linux) supports both the master and slave sides of SPI interactions.

Who uses it? On what kinds of systems?

Linux developers using SPI are probably writing device drivers for embedded systems boards. SPI is used to control external chips, and it is also a protocol supported by every MMC or SD memory card. (The older "DataFlash" cards, predating MMC cards but using the same connectors and card shape, support only SPI.) Some PC hardware uses SPI flash for BIOS code.

SPI slave chips range from digital/analog converters used for analog sensors and codecs, to memory, to peripherals like USB controllers or Ethernet adapters; and more.

Most systems using SPI will integrate a few devices on a mainboard. Some provide SPI links on expansion connectors; in cases where no dedicated SPI controller exists, GPIO pins can be used to create a low speed "bitbanging" adapter. Very few systems will "hotplug" an SPI controller; the reasons to use SPI focus on low cost and simple operation, and if dynamic reconfiguration is important, USB will often be a more appropriate low-pincount peripheral bus.

Many microcontrollers that can run Linux integrate one or more I/O interfaces with SPI modes. Given SPI support, they could use MMC or SD cards without needing a special purpose MMC/SD/SDIO controller.

I'm confused. What are these four SPI "clock modes"?

It's easy to be confused here, and the vendor documentation you'll find isn't necessarily helpful. The four modes combine two mode bits:

- CPOL indicates the initial clock polarity. CPOL=0 means the clock starts low, so the first (leading) edge is rising, and the second (trailing) edge is falling. CPOL=1 means the clock starts high, so the first (leading) edge is falling.

- CPHA indicates the clock phase used to sample data; CPHA=0 says sample on the leading edge, CPHA=1 means the trailing edge.

  Since the signal needs to stablize before it's sampled, CPHA=0 implies that its data is written half a clock before the first clock edge. The chipselect may have made it become available.

Chip specs won't always say "uses SPI mode X" in as many words, but their timing diagrams will make the CPOL and CPHA modes clear.

In the SPI mode number, CPOL is the high order bit and CPHA is the low order bit. So when a chip's timing diagram shows the clock starting low (CPOL=0) and data stabilized for sampling during the trailing clock edge (CPHA=1), that's SPI mode 1.

Note that the clock mode is relevant as soon as the chipselect goes active. So the master must set the clock to inactive before selecting a slave, and the slave can tell the chosen polarity by sampling the clock level when its select line goes active. That's why many devices support for example both modes 0 and 3: they don't care about polarity, and always clock data in/out on rising clock edges.

How do these driver programming interfaces work?

The <linux/spi/spi.h> header file includes kerneldoc, as does the main source code, and you should certainly read that chapter of the kernel API document. This is just an overview, so you get the big picture before those details.

SPI requests always go into I/O queues. Requests for a given SPI device are always executed in FIFO order, and complete asynchronously through completion callbacks. There are also some simple synchronous wrappers for those calls, including ones for common transaction types like writing a command and then reading its response.

There are two types of SPI driver, here called:

Controller drivers ...
: controllers may be built into System-On-Chip processors, and often support both Master and Slave roles. These drivers touch hardware registers and may use DMA. Or they can be PIO bitbangers, needing just GPIO pins.

Protocol drivers ...
: these pass messages through the controller driver to communicate with a Slave or Master device on the other side of an SPI link.

So for example one protocol driver might talk to the MTD layer to export data to filesystems stored on SPI flash like DataFlash; and others might control audio interfaces, present touchscreen sensors as input interfaces, or monitor temperature and voltage levels during industrial processing. And those might all be sharing the same controller driver.

A "struct spi_device" encapsulates the controller-side interface between those two types of drivers.

There is a minimal core of SPI programming interfaces, focussing on using the driver model to connect controller and protocol drivers using device tables provided by board specific initialization code. SPI shows up in sysfs in several locations:

```
/sys/devices/.../CTLR ... physical node for a given SPI controller

/sys/devices/.../CTLR/spiB.C ... spi_device on bus "B",
     chipselect C, accessed through CTLR.

/sys/bus/spi/devices/spiB.C ... symlink to that physical
     .../CTLR/spiB.C device

/sys/devices/.../CTLR/spiB.C/modalias ... identifies the driver
     that should be used with this device (for hotplug/coldplug)

/sys/bus/spi/drivers/D ... driver for one or more spi*.* devices

/sys/class/spi_master/spiB ... symlink (or actual device node) to
     a logical node which could hold class related state for the SPI
     master controller managing bus "B".  All spiB.* devices share one
     physical SPI bus segment, with SCLK, MOSI, and MISO.

/sys/devices/.../CTLR/slave ... virtual file for (un)registering the
     slave device for an SPI slave controller.
     Writing the driver name of an SPI slave handler to this file
     registers the slave device; writing "(null)" unregisters the slave
     device.
     Reading from this file shows the name of the slave device ("(null)"
     if not registered).

/sys/class/spi_slave/spiB ... symlink (or actual device node) to
     a logical node which could hold class related state for the SPI
     slave controller on bus "B".  When registered, a single spiB.*
     device is present here, possible sharing the physical SPI bus
     segment with other SPI slave devices.
```

Note that the actual location of the controller's class state depends on whether you enabled CONFIG_SYSFS_DEPRECATED or

not. At this time, the only class-specific state is the bus number ("B" in "spiB"), so those /sys/class entries are only useful to quickly identify busses.

How does board-specific init code declare SPI devices?

Linux needs several kinds of information to properly configure SPI devices. That information is normally provided by board-specific code, even for chips that do support some of automated discovery/enumeration.

Declare Controllers

The first kind of information is a list of what SPI controllers exist. For System-on-Chip (SOC) based boards, these will usually be platform devices, and the controller may need some platform_data in order to operate properly. The "struct platform_device" will include resources like the physical address of the controller's first register and its IRQ.

Platforms will often abstract the "register SPI controller" operation, maybe coupling it with code to initialize pin configurations, so that the arch/.../mach-*/board*-.c files for several boards can all share the same basic controller setup code. This is because most SOCs have several SPI-capable controllers, and only the ones actually usable on a given board should normally be set up and registered.

So for example arch/.../mach-*/board*-.c files might have code like:

```
#include <mach/spi.h>   /* for mysoc_spi_data */

/* if your mach-* infrastructure doesn't support kernels that can
 * run on multiple boards, pdata wouldn't benefit from "__init".
 */
static struct mysoc_spi_data pdata __initdata = { ... };

static __init board_init(void)
{
        ...
        /* this board only uses SPI controller #2 */
        mysoc_register_spi(2, &pdata);
        ...
}
```

And SOC-specific utility code might look something like:

```
#include <mach/spi.h>

static struct platform_device spi2 = { ... };

void mysoc_register_spi(unsigned n, struct mysoc_spi_data *pdata)
{
        struct mysoc_spi_data *pdata2;

        pdata2 = kmalloc(sizeof *pdata2, GFP_KERNEL);
        *pdata2 = pdata;
        ...
        if (n == 2) {
                spi2->dev.platform_data = pdata2;
                register_platform_device(&spi2);

                /* also: set up pin modes so the spi2 signals are
                 * visible on the relevant pins ... bootloaders on
                 * production boards may already have done this, but
                 * developer boards will often need Linux to do it.
                 */
        }
        ...
}
```

Notice how the platform_data for boards may be different, even if the same SOC controller is used. For example, on one board SPI might use an external clock, where another derives the SPI clock from current settings of some master clock.

Declare Slave Devices

The second kind of information is a list of what SPI slave devices exist on the target board, often with some board-specific data needed for the driver to work correctly.

Normally your arch/.../mach-*/board*-.c files would provide a small table listing the SPI devices on each board. (This would typically be only a small handful.) That might look like:

```
static struct ads7846_platform_data ads_info = {
        .vref_delay_usecs       = 100,
        .x_plate_ohms           = 580,
        .y_plate_ohms           = 410,
};
```

```
static struct spi_board_info spi_board_info[] __initdata = {
{
        .modalias       = "ads7846",
        .platform_data  = &ads_info,
        .mode           = SPI_MODE_0,
        .irq            = GPIO_IRQ(31),
        .max_speed_hz   = 120000 /* max sample rate at 3V */ * 16,
        .bus_num        = 1,
        .chip_select    = 0,
},
};
```

Again, notice how board-specific information is provided; each chip may need several types. This example shows generic constraints like the fastest SPI clock to allow (a function of board voltage in this case) or how an IRQ pin is wired, plus chip-specific constraints like an important delay that's changed by the capacitance at one pin.

(There's also "controller_data", information that may be useful to the controller driver. An example would be peripheral-specific DMA tuning data or chipselect callbacks. This is stored in spi_device later.)

The board_info should provide enough information to let the system work without the chip's driver being loaded. The most troublesome aspect of that is likely the SPI_CS_HIGH bit in the spi_device.mode field, since sharing a bus with a device that interprets chipselect "backwards" is not possible until the infrastructure knows how to deselect it.

Then your board initialization code would register that table with the SPI infrastructure, so that it's available later when the SPI master controller driver is registered:

```
spi_register_board_info(spi_board_info, ARRAY_SIZE(spi_board_info));
```

Like with other static board-specific setup, you won't unregister those.

The widely used "card" style computers bundle memory, cpu, and little else onto a card that's maybe just thirty square centimeters. On such systems, your `arch/.../mach-.../board-*.c` file would primarily provide information about the devices on the mainboard into which such a card is plugged. That certainly includes SPI devices hooked up through the card connectors!

Non-static Configurations

When Linux includes support for MMC/SD/SDIO/DataFlash cards through SPI, those configurations will also be dynamic. Fortunately, such devices all support basic device identification probes, so they should hotplug normally.

How do I write an "SPI Protocol Driver"?

Most SPI drivers are currently kernel drivers, but there's also support for userspace drivers. Here we talk only about kernel drivers.

SPI protocol drivers somewhat resemble platform device drivers:

```
static struct spi_driver CHIP_driver = {
        .driver = {
                .name           = "CHIP",
                .owner          = THIS_MODULE,
                .pm             = &CHIP_pm_ops,
        },

        .probe          = CHIP_probe,
        .remove         = CHIP_remove,
};
```

The driver core will automatically attempt to bind this driver to any SPI device whose board_info gave a modalias of "CHIP". Your probe() code might look like this unless you're creating a device which is managing a bus (appearing under /sys/class/spi_master).

```
static int CHIP_probe(struct spi_device *spi)
{
        struct CHIP                     *chip;
        struct CHIP_platform_data       *pdata;

        /* assuming the driver requires board-specific data: */
        pdata = &spi->dev.platform_data;
        if (!pdata)
                return -ENODEV;

        /* get memory for driver's per-chip state */
        chip = kzalloc(sizeof *chip, GFP_KERNEL);
        if (!chip)
                return -ENOMEM;
        spi_set_drvdata(spi, chip);

        ... etc
        return 0;
}
```

As soon as it enters probe(), the driver may issue I/O requests to the SPI device using "struct spi_message". When remove() returns, or after probe() fails, the driver guarantees that it won't submit any more such messages.

- An spi_message is a sequence of protocol operations, executed as one atomic sequence. SPI driver controls include:
    - when bidirectional reads and writes start ... by how its sequence of spi_transfer requests is arranged;
    - which I/O buffers are used ... each spi_transfer wraps a buffer for each transfer direction, supporting full duplex (two pointers, maybe the same one in both cases) and half duplex (one pointer is NULL) transfers;
    - optionally defining short delays after transfers ... using the spi_transfer.delay.value setting (this delay can be the only protocol effect, if the buffer length is zero) ... when specifying this delay the default spi_transfer.delay.unit is microseconds, however this can be adjusted to clock cycles or nanoseconds if needed;
    - whether the chipselect becomes inactive after a transfer and any delay ... by using the spi_transfer.cs_change flag;
    - hinting whether the next message is likely to go to this same device ... using the spi_transfer.cs_change flag on the last transfer in that atomic group, and potentially saving costs for chip deselect and select operations.
- Follow standard kernel rules, and provide DMA-safe buffers in your messages. That way controller drivers using DMA aren't forced to make extra copies unless the hardware requires it (e.g. working around hardware errata that force the use of bounce buffering).

  If standard dma_map_single() handling of these buffers is inappropriate, you can use spi_message.is_dma_mapped to tell the controller driver that you've already provided the relevant DMA addresses.
- The basic I/O primitive is spi_async(). Async requests may be issued in any context (irq handler, task, etc) and completion is reported using a callback provided with the message. After any detected error, the chip is deselected and processing of that spi_message is aborted.
- There are also synchronous wrappers like spi_sync(), and wrappers like spi_read(), spi_write(), and spi_write_then_read(). These may be issued only in contexts that may sleep, and they're all clean (and small, and "optional") layers over spi_async().
- The spi_write_then_read() call, and convenience wrappers around it, should only be used with small amounts of data where the cost of an extra copy may be ignored. It's designed to support common RPC-style requests, such as writing an eight bit command and reading a sixteen bit response -- spi_w8r16() being one its wrappers, doing exactly that.

Some drivers may need to modify spi_device characteristics like the transfer mode, wordsize, or clock rate. This is done with spi_setup(), which would normally be called from probe() before the first I/O is done to the device. However, that can also be called at any time that no message is pending for that device.

While "spi_device" would be the bottom boundary of the driver, the upper boundaries might include sysfs (especially for sensor readings), the input layer, ALSA, networking, MTD, the character device framework, or other Linux subsystems.

Note that there are two types of memory your driver must manage as part of interacting with SPI devices.

- I/O buffers use the usual Linux rules, and must be DMA-safe. You'd normally allocate them from the heap or free page pool. Don't use the stack, or anything that's declared "static".
- The spi_message and spi_transfer metadata used to glue those I/O buffers into a group of protocol transactions. These can be allocated anywhere it's convenient, including as part of other allocate-once driver data structures. Zero-init these.

If you like, spi_message_alloc() and spi_message_free() convenience routines are available to allocate and zero-initialize an spi_message with several transfers.

How do I write an "SPI Master Controller Driver"?

An SPI controller will probably be registered on the platform_bus; write a driver to bind to the device, whichever bus is involved.

The main task of this type of driver is to provide an "spi_master". Use spi_alloc_master() to allocate the master, and spi_master_get_devdata() to get the driver-private data allocated for that device.

```
struct spi_master       *master;
struct CONTROLLER       *c;

master = spi_alloc_master(dev, sizeof *c);
if (!master)
        return -ENODEV;
```

```
c = spi_master_get_devdata(master);
```

The driver will initialize the fields of that spi_master, including the bus number (maybe the same as the platform device ID) and three methods used to interact with the SPI core and SPI protocol drivers. It will also initialize its own internal state. (See below about bus numbering and those methods.)

After you initialize the spi_master, then use spi_register_master() to publish it to the rest of the system. At that time, device nodes for the controller and any predeclared spi devices will be made available, and the driver model core will take care of binding them to drivers.

If you need to remove your SPI controller driver, spi_unregister_master() will reverse the effect of spi_register_master().

Bus Numbering

Bus numbering is important, since that's how Linux identifies a given SPI bus (shared SCK, MOSI, MISO). Valid bus numbers start at zero. On SOC systems, the bus numbers should match the numbers defined by the chip manufacturer. For example, hardware controller SPI2 would be bus number 2, and spi_board_info for devices connected to it would use that number.

If you don't have such hardware-assigned bus number, and for some reason you can't just assign them, then provide a negative bus number. That will then be replaced by a dynamically assigned number. You'd then need to treat this as a non-static configuration (see above).

SPI Master Methods

`master->setup(struct spi_device *spi)`

This sets up the device clock rate, SPI mode, and word sizes. Drivers may change the defaults provided by board_info, and then call spi_setup(spi) to invoke this routine. It may sleep.

Unless each SPI slave has its own configuration registers, don't change them right away ... otherwise drivers could corrupt I/O that's in progress for other SPI devices.

> **Note**
>
> BUG ALERT: for some reason the first version of many spi_master drivers seems to get this wrong. When you code setup(), ASSUME that the controller is actively processing transfers for another device.

`master->cleanup(struct spi_device *spi)`

Your controller driver may use spi_device.controller_state to hold state it dynamically associates with that device. If you do that, be sure to provide the cleanup() method to free that state.

`master->prepare_transfer_hardware(struct spi_master *master)`

This will be called by the queue mechanism to signal to the driver that a message is coming in soon, so the subsystem requests the driver to prepare the transfer hardware by issuing this call. This may sleep.

`master->unprepare_transfer_hardware(struct spi_master *master)`

This will be called by the queue mechanism to signal to the driver that there are no more messages pending in the queue and it may relax the hardware (e.g. by power management calls). This may sleep.

`master->transfer_one_message(struct spi_master *master, struct spi_message *mesg)`

The subsystem calls the driver to transfer a single message while queuing transfers that arrive in the meantime. When the driver is finished with this message, it must call spi_finalize_current_message() so the subsystem can issue the next message. This may sleep.

`master->transfer_one(struct spi_master *master, struct spi_device *spi, struct spi_transfer *transfer)`

The subsystem calls the driver to transfer a single transfer while queuing transfers that arrive in the meantime. When the driver is finished with this transfer, it must call spi_finalize_current_transfer() so the subsystem can issue the next transfer. This may sleep. Note: transfer_one and transfer_one_message are mutually exclusive; when both are set, the generic subsystem does not call your transfer_one callback.

Return values:

- negative errno: error
- 0: transfer is finished
- 1: transfer is still in progress

`master->set_cs_timing(struct spi_device *spi, u8 setup_clk_cycles, u8 hold_clk_cycles, u8 inactive_clk_cycles)`

This method allows SPI client drivers to request SPI master controller for configuring device specific CS setup, hold and inactive timing requirements.

Deprecated Methods

`master->transfer(struct spi_device *spi, struct spi_message *message)`

> This must not sleep. Its responsibility is to arrange that the transfer happens and its complete() callback is issued. The two will normally happen later, after other transfers complete, and if the controller is idle it will need to be kickstarted. This method is not used on queued controllers and must be NULL if transfer_one_message() and (un)prepare_transfer_hardware() are implemented.

SPI Message Queue

If you are happy with the standard queueing mechanism provided by the SPI subsystem, just implement the queued methods specified above. Using the message queue has the upside of centralizing a lot of code and providing pure process-context execution of methods. The message queue can also be elevated to realtime priority on high-priority SPI traffic.

Unless the queueing mechanism in the SPI subsystem is selected, the bulk of the driver will be managing the I/O queue fed by the now deprecated function transfer().

That queue could be purely conceptual. For example, a driver used only for low-frequency sensor access might be fine using synchronous PIO.

But the queue will probably be very real, using message->queue, PIO, often DMA (especially if the root filesystem is in SPI flash), and execution contexts like IRQ handlers, tasklets, or workqueues (such as keventd). Your driver can be as fancy, or as simple, as you need. Such a transfer() method would normally just add the message to a queue, and then start some asynchronous transfer engine (unless it's already running).

THANKS TO

Contributors to Linux-SPI discussions include (in alphabetical order, by last name):

- Mark Brown
- David Brownell
- Russell King
- Grant Likely
- Dmitry Pervushin
- Stephen Street
- Mark Underwood
- Andrew Victor
- Linus Walleij
- Vitaly Wool