# Linux Spi Documentation

## The kernel development community

**Jul 14, 2020**

# CONTENTS

CHAPTER

ONE

# OVERVIEW OF LINUX KERNEL SPI SUPPORT

02-Feb-2012

## 1.1 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.

## 1.2 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.

## 1.3 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.

## 1.4 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.

## 1.5 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.

### 1.5.1 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);
```

(continues on next page)

(continued from previous page)

```
                /* 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.

### 1.5.2 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!

### 1.5.3 Non-static Configurations

Developer boards often play by different rules than product boards, and one example is the potential need to hotplug SPI devices and/or controllers.

For those cases you might need to use spi_busnum_to_master() to look up the spi bus master, and will likely need spi_new_device() to provide the board info based on the board that was hotplugged. Of course, you'd later call at least spi_unregister_device() when that board is removed.

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.

## 1.6 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_usecs setting (this delay can be the only protocol effect, if the buffer length is zero);
  - 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.

## 1.7 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().

### 1.7.1 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).

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

### 1.7.3 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.

### 1.7.4 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).

## 1.8 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

CHAPTER

TWO

# SPI USERSPACE API

SPI devices have a limited userspace API, supporting basic half-duplex read() and write() access to SPI slave devices. Using ioctl() requests, full duplex transfers and device I/O configuration are also available.

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/types.h>
#include <linux/spi/spidev.h>
```

Some reasons you might want to use this programming interface include:

- Prototyping in an environment that's not crash-prone; stray pointers in userspace won't normally bring down any Linux system.
- Developing simple protocols used to talk to microcontrollers acting as SPI slaves, which you may need to change quite often.

Of course there are drivers that can never be written in userspace, because they need to access kernel interfaces (such as IRQ handlers or other layers of the driver stack) that are not accessible to userspace.

## 2.1 DEVICE CREATION, DRIVER BINDING

The simplest way to arrange to use this driver is to just list it in the spi_board_info for a device as the driver it should use: the "modalias" entry is "spidev", matching the name of the driver exposing this API. Set up the other device characteristics (bits per word, SPI clocking, chipselect polarity, etc) as usual, so you won't always need to override them later.

(Sysfs also supports userspace driven binding/unbinding of drivers to devices. That mechanism might be supported here in the future.)

When you do that, the sysfs node for the SPI device will include a child device node with a "dev" attribute that will be understood by udev or mdev. (Larger systems will have "udev". Smaller ones may configure "mdev" into busybox; it's less featureful, but often enough.) For a SPI device with chipselect C on bus B, you should see:

**/dev/spidevB.C** ⋯ character special device, major number 153 with a dynamically chosen minor device number. This is the node that userspace programs will open, created by "udev" or "mdev".

**/sys/devices/···/spiB.C** ··· as usual, the SPI device node will be a child of its SPI master controller.

**/sys/class/spidev/spidevB.C** ··· created when the "spidev" driver binds to that device. (Directory or symlink, based on whether or not you enabled the "deprecated sysfs files" Kconfig option.)

Do not try to manage the /dev character device special file nodes by hand. That's error prone, and you'd need to pay careful attention to system security issues; udev/mdev should already be configured securely.

If you unbind the "spidev" driver from that device, those two "spidev" nodes (in sysfs and in /dev) should automatically be removed (respectively by the kernel and by udev/mdev). You can unbind by removing the "spidev" driver module, which will affect all devices using this driver. You can also unbind by having kernel code remove the SPI device, probably by removing the driver for its SPI controller (so its spi_master vanishes).

Since this is a standard Linux device driver – even though it just happens to expose a low level API to userspace – it can be associated with any number of devices at a time. Just provide one spi_board_info record for each such SPI device, and you'll get a /dev device node for each device.

## 2.2 BASIC CHARACTER DEVICE API

Normal open() and close() operations on /dev/spidevB.D files work as you would expect.

Standard read() and write() operations are obviously only half-duplex, and the chipselect is deactivated between those operations. Full-duplex access, and composite operation without chipselect de-activation, is available using the SPI_IOC_MESSAGE(N) request.

Several ioctl() requests let your driver read or override the device's current settings for data transfer parameters:

**SPI_IOC_RD_MODE, SPI_IOC_WR_MODE** ··· pass a pointer to a byte which will return (RD) or assign (WR) the SPI transfer mode. Use the constants SPI_MODE_0..SPI_MODE_3; or if you prefer you can combine SPI_CPOL (clock polarity, idle high iff this is set) or SPI_CPHA (clock phase, sample on trailing edge iff this is set) flags. Note that this request is limited to SPI mode flags that fit in a single byte.

**SPI_IOC_RD_MODE32, SPI_IOC_WR_MODE32** ··· pass a pointer to a uin32_t which will return (RD) or assign (WR) the full SPI transfer mode, not limited to the bits that fit in one byte.

**SPI_IOC_RD_LSB_FIRST, SPI_IOC_WR_LSB_FIRST** ··· pass a pointer to a byte which will return (RD) or assign (WR) the bit justification used to transfer SPI words. Zero indicates MSB-first; other values indicate the less common LSB-first encoding. In both cases the specified value is right-justified in each word, so that unused (TX) or undefined (RX) bits are in the MSBs.

**SPI_IOC_RD_BITS_PER_WORD, SPI_IOC_WR_BITS_PER_WORD …**
pass a pointer to a byte which will return (RD) or assign (WR) the number of bits in each SPI transfer word. The value zero signifies eight bits.

**SPI_IOC_RD_MAX_SPEED_HZ, SPI_IOC_WR_MAX_SPEED_HZ …**
pass a pointer to a u32 which will return (RD) or assign (WR) the maximum SPI transfer speed, in Hz. The controller can't necessarily assign that specific clock speed.

NOTES:

- At this time there is no async I/O support; everything is purely synchronous.
- There's currently no way to report the actual bit rate used to shift data to/from a given device.
- From userspace, you can't currently change the chip select polarity; that could corrupt transfers to other devices sharing the SPI bus. Each SPI device is deselected when it's not in active use, allowing other drivers to talk to other devices.
- There's a limit on the number of bytes each I/O request can transfer to the SPI device. It defaults to one page, but that can be changed using a module parameter.
- Because SPI has no low-level transfer acknowledgement, you usually won't see any I/O errors when talking to a non-existent device.

## 2.3 FULL DUPLEX CHARACTER DEVICE API

See the spidev_fdx.c sample program for one example showing the use of the full duplex programming interface. (Although it doesn't perform a full duplex transfer.) The model is the same as that used in the kernel spi_sync() request; the individual transfers offer the same capabilities as are available to kernel drivers (except that it's not asynchronous).

The example shows one half-duplex RPC-style request and response message. These requests commonly require that the chip not be deselected between the request and response. Several such requests could be chained into a single kernel request, even allowing the chip to be deselected after each response. (Other protocol options include changing the word size and bitrate for each transfer segment.)

To make a full duplex request, provide both rx_buf and tx_buf for the same transfer. It's even OK if those are the same buffer.

CHAPTER

THREE

# SPI_BUTTERFLY - PARPORT-TO-BUTTERFLY ADAPTER DRIVER

This is a hardware and software project that includes building and using a parallel port adapter cable, together with an "AVR Butterfly" to run firmware for user interfacing and/or sensors. A Butterfly is a $US20 battery powered card with an AVR microcontroller and lots of goodies: sensors, LCD, flash, toggle stick, and more. You can use AVR-GCC to develop firmware for this, and flash it using this adapter cable.

You can make this adapter from an old printer cable and solder things directly to the Butterfly. Or (if you have the parts and skills) you can come up with something fancier, providing ciruit protection to the Butterfly and the printer port, or with a better power supply than two signal pins from the printer port. Or for that matter, you can use similar cables to talk to many AVR boards, even a breadboard.

This is more powerful than "ISP programming" cables since it lets kernel SPI protocol drivers interact with the AVR, and could even let the AVR issue interrupts to them. Later, your protocol driver should work easily with a "real SPI controller", instead of this bitbanger.

The first cable connections will hook Linux up to one SPI bus, with the AVR and a DataFlash chip; and to the AVR reset line. This is all you need to reflash the firmware, and the pins are the standard Atmel "ISP" connector pins (used also on non-Butterfly AVR boards). On the parport side this is like "sp12" programming cables.

| Signal | Butterfly | Parport (DB-25) |
|---|---|---|
| SCK | J403.PB1/SCK | pin 2/D0 |
| RESET | J403.nRST | pin 3/D1 |
| VCC | J403.VCC_EXT | pin 8/D6 |
| MOSI | J403.PB2/MOSI | pin 9/D7 |
| MISO | J403.PB3/MISO | pin 11/S7,nBUSY |
| GND | J403.GND | pin 23/GND |

Then to let Linux master that bus to talk to the DataFlash chip, you must (a) flash new firmware that disables SPI (set PRR.2, and disable pullups by clearing PORTB.[0-3]); (b) configure the mtd_dataflash driver; and (c) cable in the chipselect.

| Signal | Butterfly | Parport (DB-25) |
|---|---|---|
| VCC | J400.VCC_EXT | pin 7/D5 |
| SELECT | J400.PB0/nSS | pin 17/C3,nSELECT |
| GND | J400.GND | pin 24/GND |

Or you could flash firmware making the AVR into an SPI slave (keeping the DataFlash in reset) and tweak the spi_butterfly driver to make it bind to the driver for your custom SPI-based protocol.

The "USI" controller, using J405, can also be used for a second SPI bus. That would let you talk to the AVR using custom SPI-with-USI firmware, while letting either Linux or the AVR use the DataFlash. There are plenty of spare parport pins to wire this one up, such as:

| Signal | Butterfly | Parport (DB-25) |
|---|---|---|
| SCK | J403.PE4/USCK | pin 5/D3 |
| MOSI | J403.PE5/DI | pin 6/D4 |
| MISO | J403.PE6/DO | pin 12/S5,nPAPEROUT |
| GND | J403.GND | pin 22/GND |
| IRQ | J402.PF4 | pin 10/S6,ACK |
| GND | J402.GND(P2) | pin 25/GND |

CHAPTER

FOUR

# PXA2XX SPI ON SSP DRIVER HOWTO

This a mini howto on the pxa2xx_spi driver. The driver turns a PXA2xx synchronous serial port into a SPI master controller (see Documentation/spi/spi-summary.rst). The driver has the following features

- Support for any PXA2xx SSP
- SSP PIO and SSP DMA data transfers.
- External and Internal (SSPFRM) chip selects.
- Per slave device (chip) configuration.
- Full suspend, freeze, resume support.

The driver is built around a "spi_message" fifo serviced by workqueue and a tasklet. The workqueue, "pump_messages", drives message fifo and the tasklet (pump_transfer) is responsible for queuing SPI transactions and setting up and launching the dma/interrupt driven transfers.

## 4.1 Declaring PXA2xx Master Controllers

Typically a SPI master is defined in the arch/···/mach-/board-.c as a "platform device". The master configuration is passed to the driver via a table found in include/linux/spi/pxa2xx_spi.h:

```
struct pxa2xx_spi_controller {
      u16 num_chipselect;
      u8 enable_dma;
};
```

The "pxa2xx_spi_controller.num_chipselect" field is used to determine the number of slave device (chips) attached to this SPI master.

The "pxa2xx_spi_controller.enable_dma" field informs the driver that SSP DMA should be used. This caused the driver to acquire two DMA channels: rx_channel and tx_channel. The rx_channel has a higher DMA service priority the tx_channel. See the "PXA2xx Developer Manual" section "DMA Controller".

## 4.2 NSSP MASTER SAMPLE

Below is a sample configuration using the PXA255 NSSP:

```
static struct resource pxa_spi_nssp_resources[] = {
      [0] = {
              .start  = __PREG(SSCR0_P(2)), /* Start address of NSSP */
              .end    = __PREG(SSCR0_P(2)) + 0x2c, /* Range of registers */
              .flags  = IORESOURCE_MEM,
      },
      [1] = {
              .start  = IRQ_NSSP, /* NSSP IRQ */
              .end    = IRQ_NSSP,
              .flags  = IORESOURCE_IRQ,
      },
};

static struct pxa2xx_spi_controller pxa_nssp_master_info = {
      .num_chipselect = 1, /* Matches the number of chips attached to NSSP␣
↪*/
      .enable_dma = 1, /* Enables NSSP DMA */
};

static struct platform_device pxa_spi_nssp = {
      .name = "pxa2xx-spi", /* MUST BE THIS VALUE, so device match driver␣
↪*/
      .id = 2, /* Bus number, MUST MATCH SSP number 1..n */
      .resource = pxa_spi_nssp_resources,
      .num_resources = ARRAY_SIZE(pxa_spi_nssp_resources),
      .dev = {
              .platform_data = &pxa_nssp_master_info, /* Passed to driver␣
↪*/
      },
};

static struct platform_device *devices[] __initdata = {
      &pxa_spi_nssp,
};

static void __init board_init(void)
{
      (void)platform_add_device(devices, ARRAY_SIZE(devices));
}
```

## 4.3 Declaring Slave Devices

Typically each SPI slave (chip) is defined in the arch/…/mach-/board-.c using the “spi_board_info” structure found in “linux/spi/spi.h”. See “Documentation/spi/spi-summary.rst” for additional information.

Each slave device attached to the PXA must provide slave specific configuration information via the structure “pxa2xx_spi_chip” found in “include/linux/spi/pxa2xx_spi.h”. The pxa2xx_spi master controller driver will uses the configuration whenever the driver communicates with the slave device. All

fields are optional.

```
struct pxa2xx_spi_chip {
      u8 tx_threshold;
      u8 rx_threshold;
      u8 dma_burst_size;
      u32 timeout;
      u8 enable_loopback;
      void (*cs_control)(u32 command);
};
```

The “pxa2xx_spi_chip.tx_threshold” and “pxa2xx_spi_chip.rx_threshold” fields are used to configure the SSP hardware fifo. These fields are critical to the performance of pxa2xx_spi driver and misconfiguration will result in rx fifo overruns (especially in PIO mode transfers). Good default values are:

```
.tx_threshold = 8,
.rx_threshold = 8,
```

The range is 1 to 16 where zero indicates “use default”.

The “pxa2xx_spi_chip.dma_burst_size” field is used to configure PXA2xx DMA engine and is related the “spi_device.bits_per_word” field. Read and understand the PXA2xx “Developer Manual” sections on the DMA controller and SSP Controllers to determine the correct value. An SSP configured for byte-wide transfers would use a value of 8. The driver will determine a reasonable default if dma_burst_size == 0.

The “pxa2xx_spi_chip.timeout” fields is used to efficiently handle trailing bytes in the SSP receiver fifo. The correct value for this field is dependent on the SPI bus speed (“spi_board_info.max_speed_hz”) and the specific slave device. Please note that the PXA2xx SSP 1 does not support trailing byte timeouts and must busy-wait any trailing bytes.

The “pxa2xx_spi_chip.enable_loopback” field is used to place the SSP porting into internal loopback mode. In this mode the SSP controller internally connects the SSPTX pin to the SSPRX pin. This is useful for initial setup testing.

The “pxa2xx_spi_chip.cs_control” field is used to point to a board specific function for asserting/deasserting a slave device chip select. If the field is NULL, the pxa2xx_spi master controller driver assumes that the SSP port is configured to use SSPFRM instead.

NOTE: the SPI driver cannot control the chip select if SSPFRM is used, so the chipselect is dropped after each spi_transfer. Most devices need chip select asserted around the complete message. Use SSPFRM as a GPIO (through cs_control) to accommodate these chips.

## 4.4 NSSP SLAVE SAMPLE

The pxa2xx_spi_chip structure is passed to the pxa2xx_spi driver in the "spi_board_info.controller_data" field. Below is a sample configuration using the PXA255 NSSP.

```
/* Chip Select control for the CS8415A SPI slave device */
static void cs8415a_cs_control(u32 command)
{
      if (command & PXA2XX_CS_ASSERT)
              GPCR(2) = GPIO_bit(2);
      else
              GPSR(2) = GPIO_bit(2);
}

/* Chip Select control for the CS8405A SPI slave device */
static void cs8405a_cs_control(u32 command)
{
      if (command & PXA2XX_CS_ASSERT)
              GPCR(3) = GPIO_bit(3);
      else
              GPSR(3) = GPIO_bit(3);
}

static struct pxa2xx_spi_chip cs8415a_chip_info = {
      .tx_threshold = 8, /* SSP hardward FIFO threshold */
      .rx_threshold = 8, /* SSP hardward FIFO threshold */
      .dma_burst_size = 8, /* Byte wide transfers used so 8 byte bursts */
      .timeout = 235, /* See Intel documentation */
      .cs_control = cs8415a_cs_control, /* Use external chip select */
};

static struct pxa2xx_spi_chip cs8405a_chip_info = {
      .tx_threshold = 8, /* SSP hardward FIFO threshold */
      .rx_threshold = 8, /* SSP hardward FIFO threshold */
      .dma_burst_size = 8, /* Byte wide transfers used so 8 byte bursts */
      .timeout = 235, /* See Intel documentation */
      .cs_control = cs8405a_cs_control, /* Use external chip select */
};

static struct spi_board_info streetracer_spi_board_info[] __initdata = {
      {
              .modalias = "cs8415a", /* Name of spi_driver for this device␣
↪*/
              .max_speed_hz = 3686400, /* Run SSP as fast a possbile */
              .bus_num = 2, /* Framework bus number */
              .chip_select = 0, /* Framework chip select */
              .platform_data = NULL; /* No spi_driver specific config */
              .controller_data = &cs8415a_chip_info, /* Master chip config␣
↪*/
              .irq = STREETRACER_APCI_IRQ, /* Slave device interrupt */
      },
      {
              .modalias = "cs8405a", /* Name of spi_driver for this device␣
↪*/
              .max_speed_hz = 3686400, /* Run SSP as fast a possbile */
```

(continues on next page)

(continued from previous page)

```
                .bus_num = 2, /* Framework bus number */
                .chip_select = 1, /* Framework chip select */
                .controller_data = &cs8405a_chip_info, /* Master chip config␣
↪*/
                .irq = STREETRACER_APCI_IRQ, /* Slave device interrupt */
        },
};

static void __init streetracer_init(void)
{
        spi_register_board_info(streetracer_spi_board_info,
                                ARRAY_SIZE(streetracer_spi_board_info));
}
```

## 4.5 DMA and PIO I/O Support

The pxa2xx_spi driver supports both DMA and interrupt driven PIO message transfers. The driver defaults to PIO mode and DMA transfers must be enabled by setting the "enable_dma" flag in the "pxa2xx_spi_controller" structure. The DMA mode supports both coherent and stream based DMA mappings.

The following logic is used to determine the type of I/O to be used on a per "spi_transfer" basis:

```
if !enable_dma then
        always use PIO transfers

if spi_message.len > 8191 then
        print "rate limited" warning
        use PIO transfers

if spi_message.is_dma_mapped and rx_dma_buf != 0 and tx_dma_buf != 0 then
        use coherent DMA mode

if rx_buf and tx_buf are aligned on 8 byte boundary then
        use streaming DMA mode

otherwise
        use PIO transfer
```

## 4.6 THANKS TO

David Brownell and others for mentoring the development of this driver.

CHAPTER

FIVE

# SPI_LM70LLP : LM70-LLP PARPORT-TO-SPI ADAPTER

Supported board/chip:

- National Semiconductor LM70 LLP evaluation board

  Datasheet: http://www.national.com/pf/LM/LM70.html

**Author:** Kaiwan N Billimoria <kaiwan@designergraphix.com>

## 5.1 Description

This driver provides glue code connecting a National Semiconductor LM70 LLP temperature sensor evaluation board to the kernel's SPI core subsystem.

This is a SPI master controller driver. It can be used in conjunction with (layered under) the LM70 logical driver (a "SPI protocol driver"). In effect, this driver turns the parallel port interface on the eval board into a SPI bus with a single device, which will be driven by the generic LM70 driver (drivers/hwmon/lm70.c).

## 5.2 Hardware Interfacing

The schematic for this particular board (the LM70EVAL-LLP) is available (on page 4) here:

http://www.national.com/appinfo/tempsensors/files/LM70LLPEVALmanual.pdf

The hardware interfacing on the LM70 LLP eval board is as follows:

| Parallel | | | LM70 LLP |
|---|---|---|---|
| Port | . | Direction | JP2 Header |
| D0 | 2 | • | • |
| D1 | 3 | –> | V+ 5 |
| D2 | 4 | –> | V+ 5 |
| D3 | 5 | –> | V+ 5 |
| D4 | 6 | –> | V+ 5 |
| D5 | 7 | –> | nCS 8 |
| D6 | 8 | –> | SCLK 3 |
| D7 | 9 | –> | SI/O 5 |
| GND | 25 | • | GND 7 |
| Select | 13 | <– | SI/O 1 |

Note that since the LM70 uses a “3-wire” variant of SPI, the SI/SO pin is connected to both pin D7 (as Master Out) and Select (as Master In) using an arrangement that lets either the parport or the LM70 pull the pin low. This can’t be shared with true SPI devices, but other 3-wire devices might share the same SI/SO pin.

The bitbanger routine in this driver (lm70_txrx) is called back from the bound “hwmon/lm70” protocol driver through its sysfs hook, using a spi_write_then_read() call. It performs Mode 0 (SPI/Microwire) bitbanging. The lm70 driver then inteprets the resulting digital temperature value and exports it through sysfs.

A “gotcha”: National Semiconductor’s LM70 LLP eval board circuit schematic shows that the SI/O line from the LM70 chip is connected to the base of a transistor Q1 (and also a pullup, and a zener diode to D7); while the collector is tied to VCC.

Interpreting this circuit, when the LM70 SI/O line is High (or tristate and not grounded by the host via D7), the transistor conducts and switches the collector to zero, which is reflected on pin 13 of the DB25 parport connector. When SI/O is Low (driven by the LM70 or the host) on the other hand, the transistor is cut off and the voltage tied to it’s collector is reflected on pin 13 as a High level.

So: the getmiso inline routine in this driver takes this fact into account, inverting the value read at pin 13.

## 5.3 Thanks to

- David Brownell for mentoring the SPI-side driver development.
- Dr.Craig Hollabaugh for the (early) “manual” bitbanging driver version.
- Nadir Billimoria for help interpreting the circuit schematic.

CHAPTER

SIX

# KERNEL DRIVER SPI-SC18IS602

Supported chips:

- NXP SI18IS602/602B/603

  Datasheet: http://www.nxp.com/documents/data_sheet/SC18IS602_602B_603.pdf

**Author:** Guenter Roeck <linux@roeck-us.net>

## 6.1 Description

This driver provides connects a NXP SC18IS602/603 I2C-bus to SPI bridge to the kernel’s SPI core subsystem.

The driver does not probe for supported chips, since the SI18IS602/603 does not support Chip ID registers. You will have to instantiate the devices explicitly. Please see Documentation/i2c/instantiating-devices.rst for details.

## 6.2 Usage Notes

This driver requires the I2C adapter driver to support raw I2C messages. I2C adapter drivers which can only handle the SMBus protocol are not supported.

The maximum SPI message size supported by SC18IS602/603 is 200 bytes. Attempts to initiate longer transfers will fail with -EINVAL. EEPROM read operations and similar large accesses have to be split into multiple chunks of no more than 200 bytes per SPI message (128 bytes of data per message is recommended). This means that programs such as “cp” or “od”, which automatically use large block sizes to access a device, can not be used directly to read data from EEPROM. Programs such as dd, where the block size can be specified, should be used instead.