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.

DEVICE CREATION, DRIVER BINDING

The spidev driver contains lists of SPI devices that are supported for the different hardware topology representations.

The following are the SPI device tables supported by the spidev driver:

- struct spi_device_id spidev_spi_ids[]: list of devices that can be bound when these are defined using a struct spi_board_info with a .modalias field matching one of the entries in the table.
- struct of_device_id spidev_dt_ids[]: list of devices that can be bound when these are defined using a Device Tree node that has a compatible string matching one of the entries in the table.
- struct acpi_device_id spidev_acpi_ids[]: list of devices that can be bound when these are defined using a ACPI device object with a _HID matching one of the entries in the table.

You are encouraged to add an entry for your SPI device name to relevant tables, if these don't already have an entry for the device. To do that, post a patch for spidev to the linux-spi@vger.kernel.org mailing list.

It used to be supported to define an SPI device using the "spidev" name. For example, as .modalias = "spidev" or compatible = "spidev". But this is no longer supported by the Linux kernel and instead a real SPI device name as listed in one of the tables must be used.

Not having a real SPI device name will lead to an error being printed and the spidev driver failing to probe.

Sysfs also supports userspace driven binding/unbinding of drivers to devices that do not bind automatically using one of the tables above. To make the spidev driver bind to such a device, use the following:

> echo spidev > /sys/bus/spi/devices/spiB.C/driver_override echo spiB.C > /sys/bus/spi/drivers/spidev/bind

When the spidev driver is bound to a SPI device, the sysfs node for the device will include a child device node with a "dev" attribute that will be understood by udev or mdev (udev replacement from 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.

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.

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.