**SPI**

The Serial Peripheral Interface (SPI) circuit is a synchronous serial data link that provides communication with external devices in Master or Slave Mode. It also enables communication between processors if an external processor is connected to the system.

The Serial Peripheral Interface is essentially a shift register that serially transmits data bits to other SPIs. During a data transfer, one SPI system acts as the “master”' which controls the data flow, while the other devices act as “slaves'' which have data shifted into and out by the master. Different CPUs can take turn being masters (Multiple Master Protocol opposite to Single Master Protocol where one CPU is always the master while all of the others are always slaves) and one master may simultaneously shift data into multiple slaves. However, only one slave may drive its output to write data back to the master at any given time.

A slave device is selected when the master asserts its NSS signal. If multiple slave devices exist, the master generates a separate slave select signal for each slave (NPCS).

The SPI system consists of two data lines and two control lines:

• **Master Out Slave In (MOSI):** This data line supplies the output data from the master shifted into the input(s) of the slave(s).

• **Master In Slave Out (MISO):** This data line supplies the output data from a slave to the input of the master. There may be no more than one slave transmitting data during any particular transfer.

• **Serial Clock (SPCK):** This control line is driven by the master and regulates the flow of the data bits. The master may transmit data at a variety of baud rates; the SPCK line cycles once for each bit that is transmitted.

• **Slave Select (NSS):** This control line allows slaves to be turned on and off by hardware.

**Operation**

The SPI bus can operate with a single master device and with one or more slave devices.

If a single slave device is used, the SS pin *may* be fixed to [logic low](http://en.wikipedia.org/wiki/Logic_level) if the slave permits it. Some slaves require the [falling edge](http://en.wikipedia.org/wiki/Falling_edge) (high→low transition) of the slave select to initiate an action such as the [Maxim](http://en.wikipedia.org/wiki/Maxim_Integrated_Products) MAX1242 [ADC](http://en.wikipedia.org/wiki/Analog-to-digital_converter), which starts conversion on said transition. With multiple slave devices, an independent SS signal is required from the master for each slave device.Most slave devices have [tri-state outputs](http://en.wikipedia.org/wiki/Tri-state_output) so their MISO signal becomes [high impedance](http://en.wikipedia.org/wiki/High_impedance) ("disconnected") when the device is not selected. Devices without tri-state outputs can't share SPI bus segments with other devices; only one such slave could talk to the master, and only its chip select could be activated.

**Modes of Operation**

The SPI operates in Master Mode or in Slave Mode. Operation in Master Mode is programmed by writing at 1 the MSTR bit in the Mode Register. The pins NPCS0 to NPCS3 are all configured as outputs, the SPCK pin is driven, and the MISO line is wired on the receiver input and the MOSI line driven as an output by the transmitter. If the MSTR bit is written at 0, the SPI operates in Slave Mode. The MISO line is driven by the transmitter output, the MOSI line is wired on the receiver input, the SPCK pin is driven by the transmitter to synchronize the receiver. The NPCS0 pin becomes an input, and is used as a Slave Select signal (NSS). The pins NPCS1 to NPCS3 are not driven and can be used for other purposes. The data transfers are identically programmable for both modes of operations. The baud rate generator is activated only in Master Mode.

**Data Transfer**

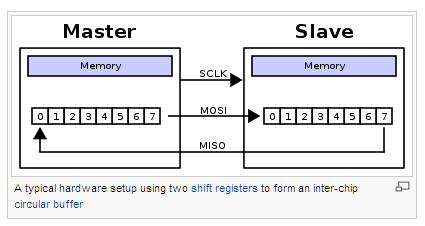
To begin a communication, the master first configures the clock, using a frequency less than or equal to the maximum frequency the slave device supports. Such frequencies are commonly in the range of 1-70 MHz.

The master then pulls the slave select low for the desired chip. If a waiting period is required (such as for analog-to-digital conversion) then the master must wait for at least that period of time before starting to issue clock cycles.

During each SPI clock cycle, a [full duplex](http://en.wikipedia.org/wiki/Full_duplex) data transmission occurs:

* the master sends a bit on the MOSI line; the slave reads it from that same line
* the slave sends a bit on the MISO line; the master reads it from that same line

Transmissions normally involve two shift registers of some given word size, such as eight bits, one in the master and one in the slave; they are connected in a ring. Data is usually shifted out with the most significant bit first, while shifting a new least significant bit into the same register. After that register has been shifted out, the master and slave have exchanged register values. Then each device takes that value and does something with it, such as writing it to memory. If there is more data to exchange, the shift registers are loaded with new data and the process repeats.



Transmissions may involve any number of clock cycles. When there are no more data to be transmitted, the master stops toggling its clock. Normally, it then deselects the slave.

Transmissions often consist of 8-bit words, and a master can initiate multiple such transmissions if it wishes/needs. However, other word sizes are also common, such as 16-bit words for touchscreen controllers or audio codecs, like the TSC2101 from [Texas Instruments](http://en.wikipedia.org/wiki/Texas_Instruments); or 12-bit words for many digital-to-analog or analog-to-digital converters.

Every slave on the bus that hasn't been activated using its slave select line must disregard the input clock and MOSI signals, and must not drive MISO. The master must select only one slave at a time.

**Clock Polarity-Phase/Generation**

The SPI Baud rate clock is generated by dividing the Master Clock (MCK) , by a value between 1 and 255. This allows a maximum operating baud rate at up to Master Clock and a minimum operating baud rate of MCK divided by 255.

SPCK Baudrate = *MCK /SCBR*

*(MCK = MASTER CLOCK)*

Programming the SCBR field (of chip select register) at 0 is forbidden. Triggering a transfer while SCBR is at 0 can lead to unpredictable results. At reset, SCBR is 0 and the user has to program it at a valid value before performing the first transfer. The divisor can be defined independently for each chip select, as it has to be programmed in the SCBR field of the Chip Select Registers. This allows the SPI to automatically adapt the baud rate for each interfaced peripheral without reprogramming.

In addition to setting the clock frequency, the master must also configure the clock polarity and phase with respect to the data. Freescale's SPI Block Guide [[1]](http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus#cite_note-0) names these two options as CPOL and CPHA respectively, and most vendors have adopted that convention.

**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.

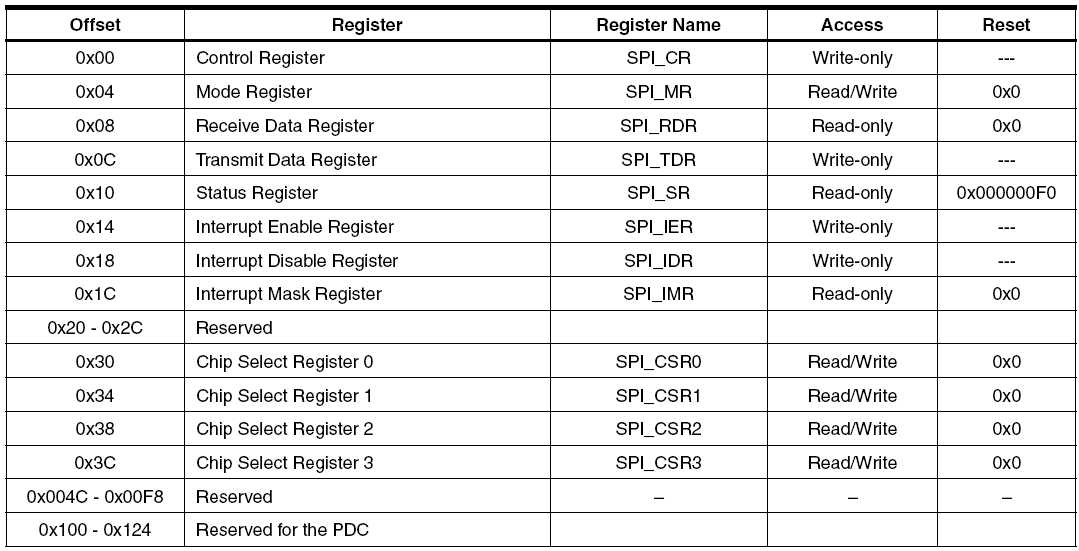
**CPHA** indicates the clock phase used to sample data; CPHA=0 says sample on the leading edge, CPHA=1 means the trailing edge.

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

The combinations of polarity and phases are often referred to as modes which are commonly numbered according to the following convention, with CPOL as the high order bit and CPHA as the low order bit:

|  |  |  |
| --- | --- | --- |
| **Mode** | **CPOL** | **CPHA** |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 1 | 0 |
| 3 | 1 | 1 |

**Important Registers and their Bits fields**

****

**In control Register,SPIEN** i.e SPI enable/disable bit for Tx/Rx.And an important field i.e LASTXFER,if this is set,the current CS will be deselected after the character has been written in Holding register has been transferred.

**In Mode Register**,fields are Master/Slave Mode,Chip select decode from decoder,fixed peripheral chip select if there, and Delay between chip selects i.e doing activate & deactivate with some delay.

**In Receive data register**,RD-data received by the SPI interface is stored in this register.

**In Transmit data register**,TD-Data to be transmitted by the SPI Interface is stored in this register. And an LASTXFER if set stated above.

**In status Register**,Checking with Receive/Trnasmit data register full/empty, Overrun error,End of Tx/TX buffer,SPI enable/disable bit.

The above status register information will get from the **interrupt registers** i.e Interrupt Enable register(IER),interrupt disable register(IDR) & Interrupt mask Register(IMR).

The most important register i.e **Chip select Register.**Here clock polarity/phase,Bits per transfer,baud rate,delay before SPCK,Delay between transfers and CSSAT bit if cleared peripheral chip select rises after last transfer(deactive).

**Peripheral Selection**

The serial peripherals are selected through the assertion of the NPCS0 to NPCS3 signals. By default, all the NPCS signals are high before and after each transfer. The peripheral selection can be performed in two different ways:

* Fixed Peripheral Select: SPI exchanges data with only one peripheral
* Variable Peripheral Select: Data can be exchanged with more than one peripheral

Fixed Peripheral Select is activated by writing the PS bit to zero in SPI\_MR (Mode Register). In this case, the current peripheral is defined by the PCS field in SPI\_MR and the PCS field in the SPI\_TDR has no effect. Variable Peripheral Select is activated by setting PS bit to one. The PCS field in SPI\_TDR is used to select the current peripheral. This means that the peripheral selection can be defined for each new data.

**Peripheral Deselecting**

When operating normally, as soon as the transfer of the last data written in SPI\_TDR is completed, the NPCS lines all rise. This might lead to runtime error if the processor is too long in responding to an interrupt, and thus might lead to difficulties for interfacing with some serial peripherals requiring the chip select line to remain active during a full set of transfers.

To facilitate interfacing with such devices, the Chip Select Register can be programmed with the CSAAT bit (Chip Select Active After Transfer) at 1. This allows the chip select lines to remain in their current state (low = active) until transfer to another peripheral is required.

**Peripheral Chip Select Decoding**

The user can program the SPI to operate with up to 15 peripherals by decoding the four Chip Select lines, NPCS0 to NPCS3 with an external logic. This can be enabled by writing the PCS-DEC bit at 1 in the Mode Register (SPI\_MR). When operating without decoding, the SPI makes sure that in any case only one chip select line is activated, i.e. driven low at a time. If two bits are defined low in a PCS field, only the lowest numbered chip select is driven low.

When operating with decoding, the SPI directly outputs the value defined by the PCS field of either the Mode Register or the Transmit Data Register (depending on PS). As the SPI sets a default value of 0xF on the chip select lines (i.e. all chip select lines at 1) when not processing any transfer, only 15 peripherals can be decoded. The SPI has only four Chip Select Registers, not 15. As a result, when decoding is activated, each chip select defines the characteristics of up to four peripherals. As an example, SPI\_CRS0 defines the characteristics of the externally decoded peripherals 0 to 3, corresponding to the PCS values 0x0 to 0x3. Thus, the user has to make sure to connect compatible peripherals on the decoded chip select lines 0 to 3, 4 to 7, 8 to 11 and 12 to 14.

**Master Mode Operation**

When configured in Master Mode, the SPI operates on the clock generated by the internal programmable baud rate generator. It fully controls the data transfers to and from the slave(s) connected to the SPI bus. The SPI drives the chip select line to the slave and the serial clock signal (SPCK). The SPI features two holding registers, the Transmit Data Register and the Receive Data Register, and a single Shift Register. The holding registers maintain the data flow at a constant rate.

***Here is the flow***

* Select the desired peripheral CS
* Enable SPI
* Set the master clock
* When a data transfer begins when the processor writes to the SPI\_TDR (Transmit Data Register). The written data is immediately transferred in the Shift Register and transfer on the SPI bus starts. While the data in the Shift Register is shifted on the MOSI line, the MISO line is sampled and shifted in the Shift Register. Transmission cannot occur without reception.
* If new data is written in SPI\_TDR during the transfer, it stays in it until the current transfer is completed. Then, the received data is transferred from the Shift Register to SPI\_RDR, then new data in SPI\_TDR is loaded in the Shift Register and a new transfer starts.
* The transfer of a data written in SPI\_TDR in the Shift Register is indicated by the TDRE bit (Transmit Data Register Empty) in the Status Register (SPI\_SR). When new data is written in SPI\_TDR, this bit is cleared.
* The end of transfer is indicated by the TXEMPTY flag in the SPI\_SR register. If a transfer delay (DLYBCT) is greater than 0 for the last transfer, TXEMPTY is set after the completion of said delay. The master clock (MCK) can be switched off at this time.
* The transfer of received data from the Shift Register in SPI\_RDR is indicated by the RDRF bit (Receive Data Register Full) in the Status Register (SPI\_SR). When the received data is read, the RDRF bit is cleared.
* If the SPI\_RDR (Receive Data Register) has not been read before new data is received, the Overrun Error bit (OVRES) in SPI\_SR is set. As long as this flag is set, data is loaded in SPI\_RDR. The user has to read the status register to clear the OVRES bit.

**SPI slave Mode**

Here is the flow when operating in slave mode

* When operating in Slave Mode, the SPI processes data bits on the clock provided on the SPI clock pin (SPCK). The SPI waits for NSS to go active before receiving the serial clock from an external master.
* When NSS(CS) falls, the clock is validated on the serializer, which processes the number of bits defined by the BITS field of the Chip Select Register 0 (SPI\_CSR0). These bits are processed following a phase and a polarity defined respectively by the NCPHA and CPOL bits of the SPI\_CSR0. Note that BITS, CPOL and NCPHA of the other Chip Select Registers have no effect when the SPI is programmed in Slave Mode. The bits are shifted out on the MISO line and sampled on the MOSI line.
* When all the bits are processed, the received data is transferred in the Receive Data Register and the RDRF bit rises. If the SPI\_RDR (Receive Data Register) has not been read before new data is received, the Overrun Error bit (OVRES) in SPI\_SR is set. As long as this flag is set, data is loaded in SPI\_RDR. The user has to read the status register to clear the OVRES bit.
* When a transfer starts, the data shifted out is the data present in the Shift Register. If no data has been written in the Transmit Data Register (SPI\_TDR), the last data received is transferred.
* If no data has been received since the last reset, all bits are transmitted low, as the Shift Register resets at 0.
* When a first data is written in SPI\_TDR, it is transferred immediately in the Shift Register and the TDRE bit rises. If new data is written, it remains in SPI\_TDR until a transfer occurs, i.e. NSS falls and there is a valid clock on the SPCK pin.
* When the transfer occurs, the last data written in SPI\_TDR is transferred in the Shift Register and the TDRE bit rises. This enables frequent updates of critical variables with single transfers.
* Then, a new data is loaded in the Shift Register from the Transmit Data Register. In case no character is ready to be transmitted, i.e. no character has been written in SPI\_TDR since the last load from SPI\_TDR to the Shift Register, the Shift Register is not modified and the last received character is retransmitted.

**Linux Kernel architecture wrt SPI**

There are two types of SPI driver, here called:

**Controller drivers**: controllers may be built in to 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 bit bangers, 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.

The most important data structures used for any slave driver development is declared in **include/linux/spi/spi.h**

Here are the important structures

**Struct spi\_device**

**{**

struct device dev; //***Driver model representation of the device***

struct spi\_master \*master; //***SPI controller used with the device***

u32 max\_speed\_hz; //***Maximum clock rate to be used with this chip***

u8 chip\_select;// ***Chipselect, distinguishing chips handled by "master"***

u8 mode; //***The spi mode defines how data is clocked out and in.***

#define SPI\_CPHA 0x01 /\* clock phase \*/

#define SPI\_CPOL 0x02 /\* clock polarity \*/

#define SPI\_MODE\_0 (0|0) /\* (original MicroWire) \*/

#define SPI\_MODE\_1 (0|SPI\_CPHA)

#define SPI\_MODE\_2 (SPI\_CPOL|0)

#define SPI\_MODE\_3 (SPI\_CPOL|SPI\_CPHA)

#define SPI\_CS\_HIGH 0x04 /\* chipselect active high? \*/

#define SPI\_LSB\_FIRST 0x08 /\* per-word bits-on-wire \*/

u8 bits\_per\_word; //***Data transfers involve one or more words; word sizes*** ***like eight or 12 bits are common.***

int irq;

void \*controller\_state;

void \*controller\_data; //***Board-specific definitions for controller, such as FIFO initialization parameters; from board\_info.controller\_data***

const char \*modalias;

}

So the above structure used to interchange data between an SPI slave and CPU memory.

The next structure is I/O interface between SPI controller and protocol drivers.

**Struct spi\_transfer** - Protocol drivers use a queue of spi\_messages, each transferring data between the controller and memory buffers.The spi\_messages themselves consist of a series of read+write transfer

segments. Those segments always read the same number of bits as they write; but one or the other is easily ignored by passing a null buffer pointer. (This is unlike most types of I/O API, because SPI hardware is full duplex.)

NOTE: Allocation of spi\_transfer and spi\_message memory is entirely up to the protocol driver, which guarantees the integrity of both (as well as the data buffers) for as long as the message is queued.

**Struct spi\_transfer{**

const void \*tx\_buf;// ***data to be written (dma-safe memory), or NULL***

void \*rx\_buf;// ***data to be read (dma-safe memory), or NULL***

unsigned [len](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=len);// ***size of rx and tx buffers (in bytes)***

[dma\_addr\_t](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=dma_addr_t) [tx\_dma](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=tx_dma);// ***DMA address of tx\_buf, if spi\_message.is\_dma\_mapped***

[dma\_addr\_t](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=dma_addr_t) [rx\_dma](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=rx_dma);// ***DMA address of rx\_buf, if spi\_message.is\_dma\_mapped***

unsigned cs\_change:1;// ***affects chipselect after this transfer completes***

[u8](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=u8) bits\_per\_word;// ***select a bits\_per\_word other then the device default*** ***for this transfer. If 0 the default (from spi\_device) is used.***

[u16](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=u16) delay\_usecs;// ***microseconds to delay after this transfer before*** ***(optionally) changing the chipselect status, then starting*** ***the next transfer or completing this spi\_message.***

[u32](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=u32) speed\_hz;// ***Select a speed other then the device default for this*** ***transfer. If 0 the default (from spi\_device) is used.***

struct [list\_head](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=list_head) transfer\_list;// ***transfers are sequenced through spi\_message.transfers***

};

Note : All SPI transfers start with the relevant chip select active. Normally it stays selected until after the last transfer in a message. Drivers can affect the chip select signal using cs\_change:

* If the transfer isn't the last one in the message, this flag is used to make the chip select briefly go inactive in the middle of the message. Toggling chip select in this way may be needed to terminate a chip command, letting a single spi\_message perform all of group of chip transactions together.
* When the transfer is the last one in the message, the chip may stay selected until the next transfer. This is purely a performance hint; the controller driver may need to select a different device for the next message.

**struct spi\_message**

The next structure is **struct spi\_message** which executes an atomic sequence of data transfers, each represented by a struct spi\_transfer. The sequence is "atomic" in the sense that no other spi\_message may use that SPI bus until that sequence completes. On some systems, many such sequences can execute as as single programmed DMA transfer. On all systems, these messages are queued, and might complete after transactions to other devices. Messages sent to a given spi\_device are alway executed in FIFO order.

struct [spi\_message](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi_message) {

struct [list\_head](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=list_head) transfers;//***list of transfer segments in this transaction***

struct [spi\_device](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi_device) \*[spi](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi);//***SPI device to which the transaction is queued***

unsigned is\_dma\_mapped:1;// ***if true, the caller provided both dma and cpu virtual*** ***addresses for each transfer buffer***

void (\*[complete](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=complete))(void \*[context](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=context));// ***called to report transaction completions***

void \*[context](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=context);// ***the argument to complete() when it's called***

unsigned actual\_length;// ***the total number of bytes that were transferred in all successful segments***

int [status](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=status);// ***zero for success, else negative errno***

struct [list\_head](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=list_head) [queue](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=queue);// ***for use by whichever driver currently owns the message***

void \*[state](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=state);// ***for use by whichever driver currently owns the message***

};

**Some important functions in these header file are**

Spi\_setup(struct [spi\_device](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi_device) \*[spi](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi))

***Sets up SPI mode and clock rate***

spi: the device whose settings are being modified

SPI protocol drivers may need to update the transfer mode if the device doesn't work with the mode 0 default. They may likewise need to update clock rates or word sizes from initial values. This function changes those settings, and must be called from a context that can sleep. The changes take effect the next time the device is selected and data is transferred to or from it.

[spi\_async](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi_async)(struct [spi\_device](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi_device) \*[spi](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi), struct [spi\_message](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi_message) \*[message](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=message))

***spi: device with which data will be exchanged***

***message: describes the data transfers, including completion callback***

This call may be used in\_irq and other contexts which can't sleep, as well as from task contexts which can sleep. The completion callback is invoked in a context which can't sleep. Before that invocation, the value of message->status is undefined. When the callback is issued, message->status holds either zero (to indicate complete success) or a negative error code.

After that callback returns, the driver which issued the transfer request may deallocate the associated memory; it's no longer in use by any SPI core or controller driver code. Note that although all messages to a spi\_device are handled in FIFO order, messages may go to different devices in other orders. Some device might be higher priority, or have various "hard" access time requirements, for example.

On detection of any fault during the transfer, processing of the entire message is aborted, and the device is de selected. Until returning from the associated message completion callback, no other spi\_message queued to that device will be processed.(This rule applies equally to all the synchronous transfer calls, which are wrappers around this core asynchronous primitive.)

[spi\_sync](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi_sync)(struct [spi\_device](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi_device) \*[spi](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi), struct [spi\_message](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi_message) \*[message](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=message));

All these synchronous SPI transfer routines are utilities layered over the core async transfer primitive. Here, "synchronous" means they will sleep uninterruptedly until the async transfer completes.

[spi\_write](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi_write)(struct [spi\_device](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi_device) \*[spi](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi), const [u8](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=u8) \*[buf](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=buf), [size\_t](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=size_t) [len](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=len))

***spi: device to which data will be written***

***buf: data buffer***

***len: data buffer size***

This writes the buffer and returns zero or a negative error code. Callable only from contexts that can sleep.

[spi\_read](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi_read)(struct [spi\_device](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi_device) \*[spi](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi), [u8](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=u8) \*[buf](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=buf), [size\_t](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=size_t) [len](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=len))

Description as above

Now one more important structure is

struct [spi\_board\_info](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=spi_board_info){ // ***board-specific information about each SPI device***

char modalias[[KOBJ\_NAME\_LEN](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=KOBJ_NAME_LEN)];//name

// ***platform\_data goes to spi\_device.dev.platform\_data,***

***controller\_data goes to spi\_device.controller\_data,*** ***irq is copied too***

const void \*[platform\_data](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=platform_data);

void \*controller\_data;

int [irq](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=irq);

[u32](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=u32) max\_speed\_hz; //speed of spi slave

[u16](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=u16) bus\_num; //which port of spi

[u16](http://enpc3240.eas.asu.edu/lxr/linux/http/ident?v=2.6.17.10;i=u16) chip\_select; //which CS pin by schematics

};

This information we have to declare in arch/arm/mact-xx/board\_file.c

For e.g

static struct spi\_board\_info ek\_spi\_devices[] = {

{ /\* 7 Segment Display 0 \*/

.modalias = "as1107",

.chip\_select = 0,

.mode = SPI\_MODE\_0,

.max\_speed\_hz = 10\*1000\*1000,

.bus\_num = 0,

},

{ /\* 7 Segment Display 1 \*/

.modalias = "as1107",

.chip\_select = 1,

.mode = SPI\_MODE\_0,

.max\_speed\_hz = 10\*1000\*1000,

.bus\_num = 0,

},

{ /\* SPI to UART expander \*/

.modalias = "xr20m1170",//"optouartprinter",//"xr20m1170",//optouartprinter",

.chip\_select = 0,

.mode = SPI\_MODE\_0,

.max\_speed\_hz = 16\*1000\*1000, /\* speed of 5mbps \*/

.bus\_num = 1,

},

};

And this information we are passing in a function

at91\_add\_device\_spi(ek\_spi\_devices, ARRAY\_SIZE(ek\_spi\_devices));

The above function defines the spi resource structure(i.e the start and end addr as platform devices with IRQ flags and passing the this above structure in a function

spi\_register\_board\_info(devices, nr\_devices);

which registers SPI devices for a given board defined in drivers/spi/spi.c

\_\_init spi\_register\_board\_info(struct spi\_board\_info const \*info, unsigned n)

info: array of chip descriptors

n: how many descriptors are provided

Skeleton of writing a SPI slave driver

static struct spi\_driver CHIP\_driver = {

.driver = {

.name = "CHIP", //name should match whatever passed in mach-xx folder

.owner = THIS\_MODULE,

},

.probe = CHIP\_probe,

.remove = \_\_devexit\_p(CHIP\_remove),

.suspend = CHIP\_suspend,

.resume = CHIP\_resume,

};

The driver core will autmatically 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 \_\_devinit 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.

Note: In addition to all above there is New spi driver model in latest kernels called as spi-dev interface.

**Spi-dev interface**

In this interface,using ioctls the data transmission/receive happens. These are the header files

#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.

One small direct example is below

In user level

char \*BLUETOOTH\_DEVICE = "/dev/ttyXX";

static uint16\_t delay = 0;

static uint8\_t data\_bits = 8;

static uint32\_t Uart\_clk = (24 \* 1000 \*1000); // Clock

static uint32\_t msbfirst = 0;

static unsigned char tx[2] = { 0xff, 0xff};

static unsigned char rx[2] = {0x09,0x09};

static void bluetooth\_write(int fd, uint8\_t \*tx)

{

int ret;

unsigned int count=0;

struct spi\_ioc\_transfer tr = {

.tx\_buf = (unsigned long)tx,

.len = 2,

.delay\_usecs = delay,

.speed\_hz = Uart\_clk,

.bits\_per\_word = data\_bits,

.cs\_change = 0,

};

// printf("\n%s",tx);

ret = ioctl(fd, SPI\_IOC\_MESSAGE(1), &tr);

// printf("\nret = %d\n",ret);

if (ret == -1)

printf("can't send spi message");

for(count = 0; count < 1000 ; count++);

}

static void bluetooth\_read(int fd, uint8\_t \*tx)

{

int ret;

unsigned int count=0;

struct spi\_ioc\_transfer tr = {

.tx\_buf = (unsigned long)tx,

.rx\_buf = (unsigned long)rx,

.len = 2,

.delay\_usecs = delay,

.speed\_hz = Uart\_clk,

.bits\_per\_word = data\_bits,

.cs\_change = 0,

};

ret = ioctl(fd, SPI\_IOC\_MESSAGE(1), &tr);

if (ret == -1)

printf("can't send spi message");

sleep(1);

for(ret = 0;ret < 1;ret++)

printf("Rx = %c\n",rx[1]);

for(count = 0; count < 1000 ; count++);

}

Some important ioctls for the above interface as below

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.

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.

Disadvantages of using this interface:

* At this time there is no async I/O support; everything is purely synchronous(no interrupt also).
* 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.